A RECONFIGURATION ALGORITHM FOR A DOUBLE-LOOP TOKEN-RING LOCAL AREA NETWORK[1]

Raphael Rom
Nachum Shacham
SRI International
Menlo Park, California
July 1985

ABSTRACT

An algorithm to reconfigure a fault-tolerant double-loop token-ring local area network following topological changes is presented. The algorithm guarantees that at all operational times the network is organized to provide maximum possible connectivity among the nodes - either one loop that encompasses all nodes in the network or several subloops each of which operates as a separate token ring network. Tokens are generated or eliminated as necessary to result in one and only one token in each (sub)loop. The algorithm is formally specified and its properties are verified.

1.0 INTRODUCTION

Token ring is a well known architecture for local area networks (LANs) which is preferred by many over the bus architecture because of its inherent advantages:

1. High speed - a token ring can operate at speeds of hundreds of Mbps which are much higher than is practical for bus LANs.
2. Guaranteed delay - unlike the contention-based access to the bus, the token passing algorithm guarantees that each node will have an access to the communication medium at least once per token cycle thereby providing an upper bound on the delay.
3. Longer distances - whereas the performance of contention bus LAN is highly sensitive to the cable length, token ring network can operate over rather long distances.

The disadvantage of the token ring lies in its relative poor reliability: a single link or node failure disrupts the operation of the whole network. Several techniques have been proposed to overcome this deficiency. For example, the star-shaped ring[1] attempts to alleviate the above problem by concentrating all the switching elements in a central office thereby making the maintenance easier. Other schemes enhance the ring's reliability by adding more cables to it. Two famous architectures -- the daisy chain[2] and the double loop[3]--have each node connected to four unidirectional

---

1. This work was partially supported by FMC Corporation under SRI Project 7390

links, two incoming and two outgoing. Thus when one loop is broken the nodes can use the other one for transmission. Of the two architecture the double loop, shown in Figure 1, has a less complex topology and requires much simpler routing procedures which implies overall simplicity and lower cost nodes.

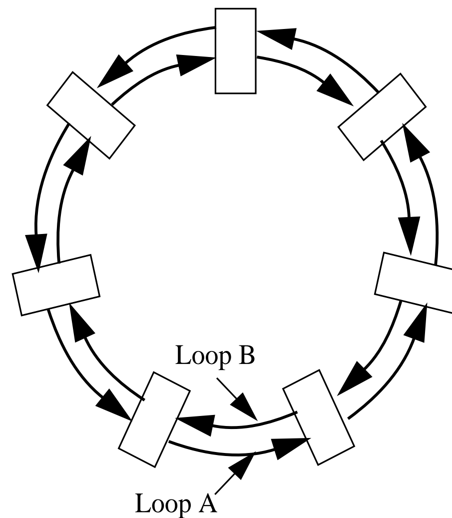

Figure 1: A Double-Loop LAN

Theding1 double loop network also provides for simpler addition and deletion of nodes and easier installation due to simpler wiring procedures. These features make the double loop LAN very attractive in situations where frequent topological changes are expected, such as in military environment. Its behavior under such changes is the topic of this paper.

During normal operating conditions, that is, when both loops are intact and all nodes are operational, the double loop network uses only one of the loops in a normal token passing protocol[4]. This statement is made to simplify the discussion below; the algorithms we present here work equally well if both loop are used whenever possible. When the loop on which the network operates breaks, the network switches to the other loop and continues to operate under the normal protocol. However when both loops are broken at one or more places the nodes cannot keep operating normally and network reconfiguration is called for.

Reconfiguration is done by means of “loopback” which amounts to reflecting the traffic received at a node from a neighbor back to that neighbor via the second loop instead of forwarding it to the next node on the same loop. As depicted in figure 2, this is done when a node discovers that it is disconnected from one of its two neighbors. In Figure 2(a) there is a double failure between two nodes and the loopback results in one large ring that spans all nodes over both cables. Figure 2(b) depicts the case where failures occur at two different spots of the LAN and on the two cables, in which case the most the network can do is reconfigure itself into two subloops, one containing nodes A,B, and C while the other contains the rest. This basic loopback operation, which is done on a local basis, provides the network with the capability to organize itself in various forms of loops that should yield the maximum possible connectivity in the network.

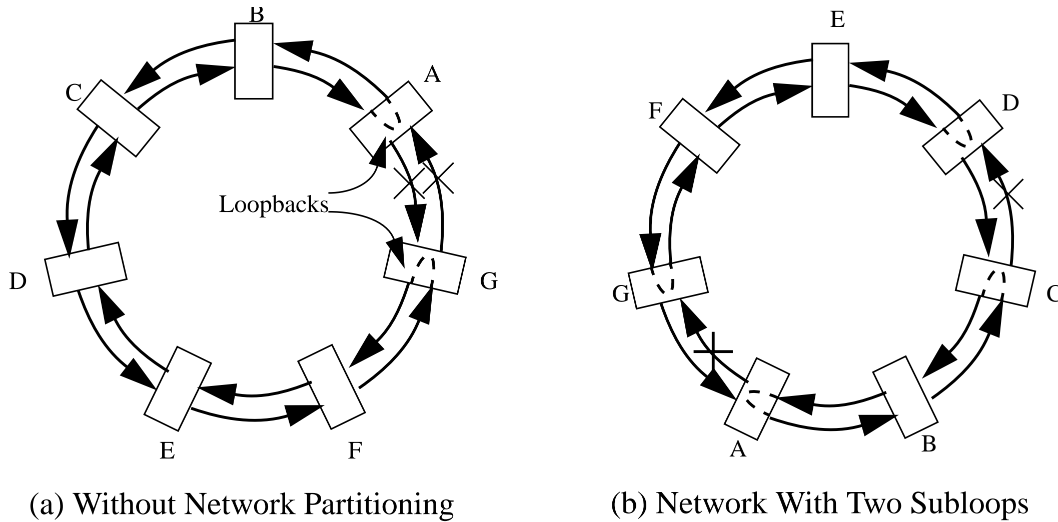

Figure 2: Reconfiguration Using Loopbacks

However, the network should exercise this feature with care, otherwise it may result in suboptimal operation conditions. For example, even if local connectivity indicates that a loopback may be necessary, loopback should not be carried out if one of the loops is intact and can serve the whole node population. Moreover, while carrying out the loopback operation or while removing loopback upon recovery, the number of subloops may change which implies the need to generate or eliminate tokens. If several such events occur within a small time interval, various messages and tokens may "float" in the network and may severely disrupt operation unless filtered out in an orderly manner.

These examples demonstrate that whereas the loopback operation is simple, its ramifications are not, and there is a need for a proper set of distributed algorithms that provide the right transitions upon topological changes and make sure that following those transitions the network has the right number of token that provide for proper operation. In this paper we present such algorithms which are rigorously specified and verified. In Section II we describe the model of the network nodes that execute the algorithms. Section III presents a description of the two major elements of the algorithms, provide a specification for them and prove their correctness.

2.0 MODEL

The system consists of nodes interconnected in a double loop structure. Every node is connected to each of its neighbors via two unidirectional links--an incoming and an outgoing link. The two *loops* thus formed are referred to as loop-A and loop-B (the nodes themselves are not aware of this labeling). Having two incoming links and two outgoing ones each node associates every incoming link with a single outgoing one. One such association is called *normal* and forms the double loop connection. The other association (made only when full connectivity cannot be achieved on either loop) is called *loopback*. Nodes making a loopback association are referred to as *loopback nodes*

or in the *loopback mode*. When one or more of the nodes are in the loopback mode, one or more *subloops* are formed (note that a subloop may contain all network nodes, such as in figure 2a).

A standard token passing scheme governs the access to the network[3][4][5]. Only a node that "owns" the token may transmits a packet of information. The transmitting node is responsible to remove the message from the network. Having transmitted a packet, the node must unconditionally pass the token (this avoids indefinite token capturing by any node). At any time the token can be either owned by a node (e.g., when it is transmitting a packet) or on its way between nodes. To "destroy" a token, which is sometimes required by the reconfiguration algorithm, it must be first owned by a node.

When not transmitting its own packet a node forwards every incoming packet on the outgoing link associated with the link from which the packet arrived. To achieve maximum connectivity a reconfiguration algorithm is implemented in the network. To execute the algorithm nodes exchange *messages* (as opposed to packets in the normal communication). Each node is capable of transferring a message to its next neighbor if the link between them is operational. Thus messages can pass along links over which packets do not (for example, the link marked 1 in figure 2b). The node is assumed to know on which link a message arrived. Messages are sent only from a node to its immediate neighbor and do not require a token. In effect, messages have priority over packets in that a node may abort an ongoing transmission of a packet if it needs to send a message to its neighbor. Because the links between nodes are unidirectional and because the sender does not necessarily know the status of a link, the sender does not know whether the message actually arrived.

The reconfiguration algorithm we present is an event driven algorithm and does not use timeouts. The relevant events are the failure or recovery of an incoming link or the receipt of a message. A node is assumed to be able to determine whether or not an incoming link is operational (this is usually done by sensing the clock on that link). We assume a node cannot sense the state of an outgoing link. Also, the reconfiguration algorithm is concerned with the link state transitions but not with the reason for the transition. It is only required that the events "link failed" and "link recovered" are defined within every node. It is possible, for example, for a node not to declare a link as operational until it is active for some time thereby avoiding flutter.

Every node processes events in the order it becomes aware of them. Each such processing may be arbitrary long but finite. It is assumed that a node can conduct a *local connectivity test* with its neighbor after which it can determine whether both links connecting the two are operational. We consider this test a single basic operation. Because of the potential disturbances it may cause, the use of this test is kept to a minimum--only in cases where there is no other way to acquire that same information. Each node has a unique ID stored within it which may be included in messages. In addition the node stores the following information:

- S_A and S_B. These are two records, one bit each, indicating whether the respective loop is perceived by this node to be connected ($S_A = 0$ indicates that the node perceives loop A to be broken).
- *MEMORY.* This is a memory cell in which a unique ID of some other node can be temporarily stored.

- *SCT_A*, *SCT_B*, *SLT*, *SLE*. These are auxiliary (state) variables, one bit each. Their exact function is described when they are first use.

3.0 RECONFIGURATION ALGORITHMS

The objective of the reconfiguration algorithm is to establish the maximum connectivity in every segment of the loop. This implies operating on a single cable that joins all the nodes, if such a connection is possible. Otherwise, subloops of maximum size must be formed. Upon cable severance, either the nodes switch to the other cable (if it is not broken) or a new subloop is established.

When a new subloop is established the residues of the previous state, such as control messages and token(s) are first cleared. Then, a new token is generated allowing the nodes on the subloop to operate as in an ordinary token-ring network.

Upon link recovery, either a fully connected loop results, in which case all nodes switch to the connected loop, or possibly, the number of subloops is reduced requiring the elimination of one of the existing tokens.

We present the algorithm gradually. The basic algorithm that is introduced first, updates the node's perception of the state of the two loops. The correctness of this algorithm is proven. Then the algorithm is extended to exchange the necessary information for looping back, and then the token control algorithm is presented. In the last subsection we indicate how all these pieces fit together to form a single unified algorithm.

3.1 The Basic Algorithm

The basic algorithm deals only with loop status. Its goal is to assure that after topological changes all nodes reach the same (correct) perception of the status of both loops. It does not deal with looping back or token control. To execute the algorithm neighboring nodes exchange the following messages:

FT indicates a Failure on This loop, i.e., on the loop on which this message is received. This message carries a link-status bit.

CT indicates a Connectivity Test. This message carries the ID of its originator.

CE indicates that Connectivity has been Established.

The idea behind the algorithm is as follows. A node that senses a link failure sends an FT message to its next neighbor along the loop that just failed thereby notifying everybody of the failure. This message is sent unconditionally, even if the loop was broken before the recent, failure since all previous failures might have recovered at the same time the new failure occurred (if the loop had been broken before this message reaches only those still connected to the originator). When a link

recovers, the node sensing it tries to find out if this recovery caused the entire loop to be connected. This is done by sending the CT message and forwarding it from node to node. Thus, like the FT message, the CT message propagates up to the next broken link. If there is no broken link the originator of the CT message (identified by the ID carried by the message) receives its own message back, knows the loop is connected, and notifies everybody by a CE message which is forwarded in the same manner.

Each node uses the state variables SCT_A, SCT_B, SCE_A, and SCE_B to denote that it has sent CT and CE messages on loop A and loop B respectively.

In the description of the algorithm we denote by X the relevant loop and by $\bar{X}$ the other loop. The description consists of the actions taken by a node in response to every possible event.

1. X-link failed
    - 1.a. Send FT on X
    - 1.b. $S_X \leftarrow 0$
    - 1.c. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$
2. X-link recovered
    - 2.a. Send CT on X
    - 2.b $SCT_X \leftarrow 1$ $SCE_X \leftarrow 0$
3. Receive FT on X
    - 3.a. Send FT on X
    - 3.b. $S_X \leftarrow 0$
    - 3.c. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$
4. Receive CT(ID) on X
    - 4.a. ID = MyID
        - 4.a.1. Send CE on X
        - 4.a.2. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 1$
    - 4.b. ID $\neq$ MyID
        - 4.b.1. Send CT(ID) on X
5. Receive CE on X
    - 5.a. if $SCE_X = 0$ send CE on X
    - 5.b. $S_X \leftarrow 1$
    - 5.c. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$

Notable in this algorithm is the mutual independence of the status bit of the two loops. This is evident because the setting of S_X is never conditioned upon $S_{\bar{X}}$. This means that subsequent proofs of properties need be done only with respect to one loop.

The following proposition proves the main property of the basic algorithm.

Proposition 1: Within a finite time after a topological change, either all nodes have the same status record (S_X) or another topological change occurs.

Proof: The proposition is interesting only in the case where no more topological changes occur and the system is allowed to settle. We prove by contradiction.

Suppose that after an arbitrarily long time there are two nodes α' and β' with different status bit of loop A. This means that there exist two neighboring nodes α and β with differing view of the A-loop status. Without loss of generality we assume that α preceded β in the direction of the loop and that at α $S_A = 0$ and at β $S_A = 1$. We consider the last event at β.

Failure of link on loop A. This is impossible since a failure in an incoming A-link would result in $S_A = 0$ at β. Subsequently, we shall make use of the fact the link from α to β is operational.

Recovery of link on loop A. Prior to the recovery, the link must have been in a failure mode and β must have had $S_A = 0$. β sends a CT message but does not receive anything (the link recovery was the last event at β), nothing could have caused β to set its S_A to 1.

Receipt of an FT message. This would have caused $S_A = 0$ (line 3.b of the algorithm).

Receipt of a CE message. Since the link from α to β is operational, α must have seen that message but since at α $S_A = 0$ this message was generated by α (otherwise it would have set its S_A to 1, step 5). Since the CE message did not make it back to α the loop must have been broken at some node γ after α's CT message passed there. γ should have generated an FT message (line 1a) which should have arrived at α, leading to a contradiction.

Receipt of a CT message containing the ID of some node γ. We first show that this message does not make it to γ. Suppose it does, so γ would then send a CE message (line 4.a.1) which does not arrive at β. As in the previous case, a link between γ and β must have failed after the CT message passed; but then β should have received a FT message (line 1.a). Not making it back to γ means that the loop is still broken and no CE message will be generated.

Since the CT message does not cause any change in the status bits, its occurrence can be ignored altogether, and all the above arguments can be applied recursively with respect to the message preceding the last CT. Because there is a finite number of users only a finite number of consecutive CT messages can be generated and thus the recursion is finite.

The last event to be considered is the receipt of a CT message with β's ID, which is responded with a CE message (line 4.a.1) which does not make it back, meaning that a failure occurred at some node γ after the CT message passed there and before the CE message did. But that should have resulted in an FT message from γ which should have arrived at β. □

The proposition assures synchronization only after a topological change occurs. If a network is initially operated in an unsynchronized state and no topological changes occur, the network will remain unsynchronized. Thus, it is necessary to start the network properly; for example, by having every node declare its link down at startup and then "recover" them.

Proposition 2: Within a finite time after a topological change, either all nodes have the correct status record or another topological change occurs.

Proof: By proposition 1, within a finite time after a topological change all nodes have the same status record. It is left to show that the algorithm converges to the correct value.

Suppose in all nodes $S_A = 1$ yet the loop is broken. This is a contradiction since there is at least one node whose incoming link is broken and whose S_A record must be therefore be set to 0.

Suppose in all nodes $S_A = 0$ yet the loop is connected. Since there was a topological change and since the loop is connected the last topological change must have been link recovery, possibly at several nodes. Consider one of those nodes whose incoming link recovered and who therefore sent a CT message. Since the loop is connected, that message makes it back to the originator who then sends a CE message which must cause $S_A \leftarrow 1$. □

In the basic algorithm the SCE_X variable is introduced for efficiency reasons only. It is tested only in step 5a but the same result could be obtained if the CE message had carried an ID (resulting in longer messages). Also, if several nodes concurrently generate CE messages, it is not necessary that each of these complete a round (as would be the case if the user ID would be the criterion); it is sufficient to ensure that each user receives one such message. Note that any event other then receiving somebody else's CT, causes the resetting of SCE_X.

Tracing the activities of the basic algorithm when recovery occurs in more than one place, reveals that all the CT messages travel once around the loop and several CE messages traverse each a portion of the loop so that every node sees a single CE message. This is somewhat of a waste since it would be sufficient for one CT message to complete round and cause its originator to generate a CE message that would go around the entire loop. To achieve this we make use of the ID carried by each CT message and the SCT_X flag (heretofore unused). When a node originates a CT message it will not forward other CT messages that carry an ID lower then its own. This is equivalent to choosing a leader in the manner described in[6] (where its correctness is proved).

To incorporate this into the basic algorithm requires only a change in step 4, that would now become:

4. Receive CT(ID) on X
    - 4.a. If $SCT_X = 0$
        - 4.a.1. Send CT(ID) on X
    - 4.b. If $SCT_X = 1$
        - 4.b.1 If ID = MyID
            - 4.b.1.a. Send CE on X
            - 4.b.1.b. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 1$
        - 4.b.2 If ID>MyID
            - 4.b.2.a. Send CT(ID) on X
            - 4.b.2.b. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$

We note that the SCE flag is still used to avoid the need for the CE message to carry the ID. Note also that when a leader is chosen (step 4.b.1) it sets its *SCE* flag to 1; because the leader is unique there is only one node with $SCE = 1$, a fact we shall subsequently use for token control.

3.2 Loopback Extension

When neither of the loops is connected, a loopback must be placed in every node adjacent to an inoperational link so that partial connectivity is achieved. The basic algorithm provides only synchronization with respect to loop status, and is not concerned with loopbacks. In fact, with the basic algorithm alone nodes may not know whether or not they need to install a loopback. The extension presented in this subsection deals with this.

The main problem is for a node to become aware of the status of its outgoing links. To do this two new messages are introduced:

FO indicates a Failure on the Other loop, i.e., not on the loop on which this message is received.

RO indicates a Recovery on the Other loop, i.e., not on the loop on which this message is received.

These two messages are not forwarded by their recipients. They are sent by a node sensing a failure or recovery of a link to its preceding neighbor for who this is an outgoing link. Because the status of the other link is not known by the sender of these messages, it does not know whether or not they arrived. Here the link status bit carried by the FT message is used; when a node sends an FT message it designates with this bit the status of its incoming link on the other loop ($l=1$ means operational). The algorithm would now look as follows:

1. X-link failed
    - 1.a. Send FT on X
    - 1.b. $S_X \leftarrow 0$
    - 1.c. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$
    - 1.d. Send FO on $\bar{X}$
    - 1.e. If $S_{\bar{X}} = 0$ loop back form $\bar{X}$ to X.
2. X-link recovered
    - 2.a. Send CT on X
    - 2.b. $SCT_X \leftarrow 1$ $SCE_X \leftarrow 0$
    - 2.c. Send RO on $\bar{X}$
    - 2.d. If $S_{\bar{X}} =0$ Conduct local connectivity test; remove loopbacks if successful
3. Receive FT(l) on X
    - 3.a. Send FT on X
    - 3.b. $S_X \leftarrow 0$
    - 3.c. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$
    - 3.d. If $S_{\bar{X}} = 0$
        - 3.d.1. If $l=0$ loop back form $\bar{X}$ to X.
        - 3.d.2. If $\bar{X}$-link inoperational loop back form X to $\bar{X}$.
4. Receive CT(ID) on X
    - 4.a. If $SCT_X = 0$
        - 4.a.1. Send CT(ID) on X
    - 4.b. If $SCT_X = 1$

4.b.1 If ID = MyID

4.b.1.a. Send CE on X

4.b.1.b. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 1$

4.b.2 If ID >MyID

4.b.2.a. Send CT(ID) on X

4.b.2.b. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$

5. Receive CE on X

5.a. if $SCE_X = 0$ send CE on X

5.b. $S_X \leftarrow 1$

5.c. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$

5.d. Undo any loopbacks.

6. Receive FO on X

6.a. $SCT_{\bar{X}} \leftarrow 0$ $SCE_{\bar{X}} \leftarrow 0$

6.b. If $S_X = 0$ loop back form $\bar{X}$ to X.

7. Receive RO on X

7.a Undo loopback

We note that in (the newly added) steps 6 and 7 the status bits of the loops is not changed, and therefore the synchronization results of propositions 1 and 2 still hold. What needs to be proven is that loopbacks are placed and dismantled in the right places at the right time.

Proposition 3: Within a finite time after a topological change loopback is performed iff necessary.

Proof. We recall that loopback need be performed at every node with either an incoming or outgoing link is broken, if and only if both loops are inoperational. Correct looping back is therefore equivalent to knowing the status of the adjacent links, and since the status of incoming links can be directly sensed, the status of outgoing links is our main concern.

Before starting the proof, we make notice of the following auxiliary lemma: “No superfluous loopbacks are installed”. This lemma is trivially correct since loopbacks are installed in steps 1e, 3d, and 6b, in each of which it is assured that both loops are inoperational and in each case the state of the relevant adjacent links is explicitly known. (In a similar way it can be shown that any loopback removal is necessary).

We prove the rest by contradiction.

Part 1: A loopback that should not exist does.

Assume the node a is looping back, diverting traffic away from node b. By the above lemma, a rightfully installed the loopback, i.e., both loops were broken and at least one of the links between a and b was broken. Two possible reasons could cause the loopback to be unnecessary:

Case 1: One of the loops became connected. By virtue of proposition 2 all nodes know it and dismantle loopbacks (step 5b).

Case 2: Both loops remain disconnected, both links between *a* and *b* are operational, however, *a* assumes that the link (*a*,*b*) is inoperational and is therefore looping back. One of the links between *a* and *b* must have recovered; we trace events after the recovery of the link that was last to recover. If it were *(b,a)* then *a* would successfully complete a connectivity test and subsequently remove the loopback (step 2d). If it were *(a,b)* then *b* would send *a* an RO message (step 2c) which would have caused *a* to remove the loopback (step 7).

Part 2: A loopback that should exist does not.

For such a circumstance to prevail there must exist a pair of adjacent nodes *a* and *b* in the following configuration:

1. One or both of the links between *a* and *b* are broken.
2. *a* assumes that both links are operational.
3. Both loops are broken (if this were not the case then loopback is unnecessary).

Suppose the link *(b,a)* is broken. When it broke, the other loop could either have been operational or broken. In the former case, the other loop broke some time later requiring a transition of S_X to 0 at *a*, which can be performed only at steps 1 and 3, but in both a loopback would have been installed. In the latter case, i.e., the other loop was broken, the failure of *(b,a)* would cause *a* to execute step 1 and install a loopback.

Thus *(b,a)* is operational, and therefore *(a,b)* is not. We observe the events after the failure of the link *(a,b)*. *b* sends an FO message to *a* (step 1d) which is not received by *a*, since, if it were received *a* would have looped back (step 6b). We thus conclude that *(b,a)* had been inoperational when *(a,b)* failed. Thus, some time later *(b,a)* recovered at which time *a* conducts a local connectivity test (step 2d) and realizes that *(a,b)* is inoperational, yielding a contradiction. □

It should be noted that step 2d is the only place where local connectivity test is performed. However, this is not disruptive to the regular operation of the subloops since a loopback exists there so that no packet flow is interrupted.

3.3 Token Control

The algorithm presented so far guarantees that following topological changes the network configures itself into the minimum number of subloops, thereby providing maximum possible connectivity. However, proper operation also requires that each loop or subloop has exactly one token, meaning that upon reconfiguration tokens may have to be generated or destroyed. In this subsection we present a token control algorithm whose objective is to organize the network so that in every connected loop or subloop a single token is present. As before, token control need be exerted only after topological changes. Our general approach is to destroy all existing tokens in any loop or subloop that undergoes reconfiguration, and then generate a single token in every such loop or subloop. We distinguish two separate cases depending on whether or not a subloop configuration exists after a topological change.

The case in which one of the loops becomes connected after a topological change is straightforward to handle. Connectivity is established by the CE message. Since there is only one such message the originator of that message generates a new token while all others destroy theirs (if they have one). To implement this, step 5 of the above algorithm is modified as follows:

5. Receive CE on X
- 5.a. if $SCE_X = 0$ send CE on X
  - 5.a.1. Destroy token
  - 5.a.2. Send CE on X
- 5.b. If $SCE_X = 1$ Generate Token
- 5.c. $S_X \leftarrow 1$
- 5.d. $SCT_X \leftarrow 0$ $SCE_X \leftarrow 0$
- 5.e. Undo any loopbacks.

To treat the other case let us restrict ourselves, for explanation purposes, to link severance and describe an algorithm for token control in this case. Focusing on one of the subloops we note that two nodes can be singled out--the two in the loopback mode. One of them will be selected to generate the token. The nodes that do not loopback do not actively participate in the execution except for message forwarding (in the following description only the activities of the looping back nodes are addressed).

The algorithm has two phases: a cleanup phase and a selection phase. The cleanup phase assures that any left over control messages and tokens from previous executions are eliminated. The selection phase assures that exactly one node generates a token. Two messages are used, similarly to those used for testing the entire loop connectivity:

LT is a Loop Test message performing the cleanup. This message carries the ID of the originator and a “bounce bit” (see below).

LE is a Loop Establish message indicating that a loop is established and a token generator is selected. This message carries the ID of its originator.

To explain the approach consider figure 3 which shows part of the entire system (the dashed lined indicate the existence of intermediate nodes). The link marked 1 had been broken when the link marked 2 broke. This necessitates to exert token control in the subloop that forms between nodes A and F. While this is going on the links marked 3 and 4 break. Token control is now required in three subloops: A-B, C-D, and E-F. Some message from the previous execution (of subloop A-F) are, however, still underway. Moreover, it is possible that such messages never arrive at their intended destination and need be cleaned up (we refer to those as *orphan* messages). Note also that the algorithm must handle the concurrent failure of links 3 and 4.

To assure the cleanup the LT message must go around the subloop until it returns to its originator. To stop orphan LT messages from an infinite loop a “bounce bit” is carried in the message. An LT message is originally sent with the bounce bit set to 0. When it arrives at a looping back node it is forwarded (on the other loop) with the bounce bit set to 1. An LT message with a bounce bit set to 1 is removed by a loopback node and forwarded by all others. When an LT message returns to its

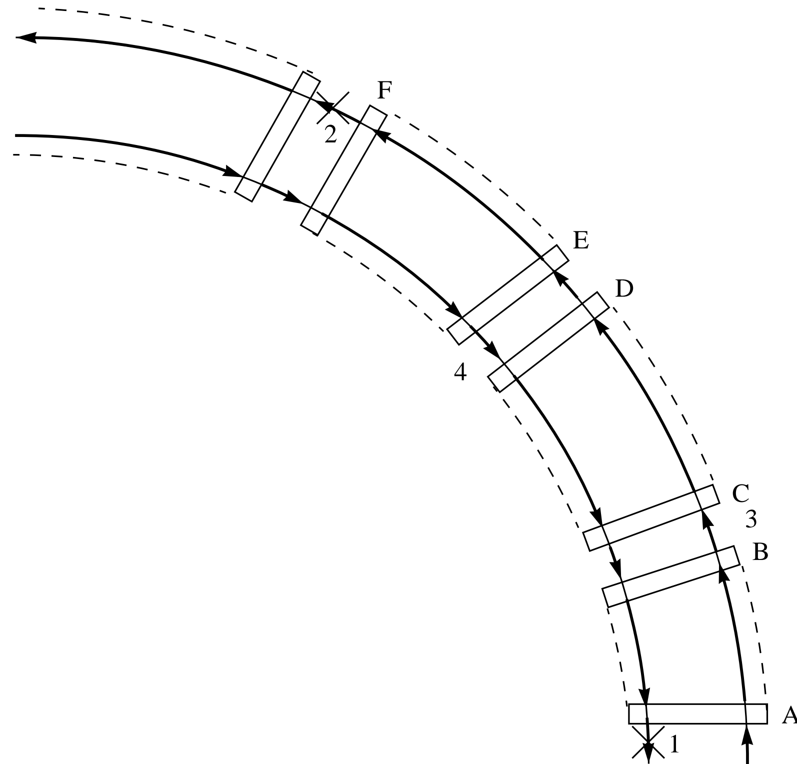

Figure 3: Token-Control Example

originator the subloop contains no more orphans. In fact, if the originator did not receive any other LT message since it sent its own, it can safely generate a token.

However, two LT message could have been received as a result of concurrent link severance, necessitating to arbitrate between the two originating nodes that are potential token generator. The LE message is then transmitted, carrying an ID; The node with the highest ID is selected to generate the token.

For every orphan message there is a node expecting it to return. In order to stop this node from waiting forever every node remembers, in a MEMORY cell the ID carried by the most recent LT message it encounters. If an LE message is received carrying the same ID as that stored in the MEMORY, it is an indication that another node succeeded in the cleanup operation, and hence our node will not receive the message it is waiting for and might as well give up.

We use two bits per node--SLT and SLE--to designate the fact that this node has sent an LT and an LE message respectively. The state diagram below describes the transitions in the execution of this algorithm by looping back nodes (the others only forward messages). In this diagram the transitions are marked “event/action” where the events are the receipt of the following messages: (1) *LT(ID=MyID,1)* (2) *LT(ID ≠ MyID,0)*, (3) *LT(ID ≠ MyID,1)*, (4) *LE(ID)*.

Several observation regarding this diagram are in order:

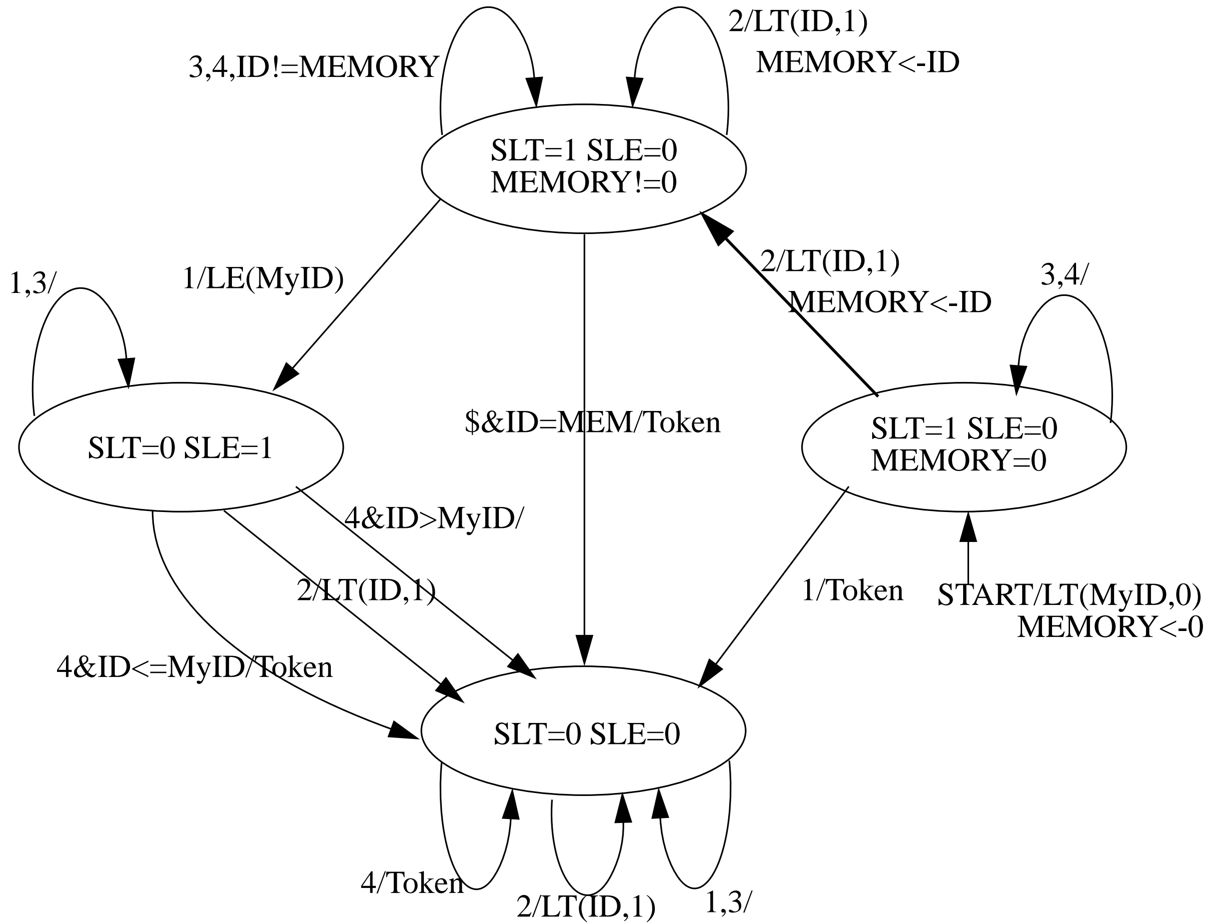

Figure 4: Token-Control State Diagram

- The algorithm must be “externally started.” This is done so that both failure and recovery can be accommodated. The nodes sensing the link failure or recovery cause the algorithm to be started, thereby assuring the execution in every newly generated subloop.
- Messages of type 3 are always tossed, since theses are the orphans.
- It is assumed the MEMORY cell can be empty, and recognized as such by the node.

3.3.1 Proof of Correctness

To prove the correctness of the algorithm we transform it from the state diagram above to the event/response notation. This results in the following specification:

8. Receive LT (ID,B) on X
    - 8.a. Destroy token
    - 8.b. If loopback
        - 8.b.1. If *ID=MyID and B=1*
            - 8.b.1.a. *SLE* $\leftarrow$ *0*
            - 8.b.1.b. If *SLT = 1*
                - 8.b.1.b.1. If MEMORY=0 Generate token

8.b.1.b.2. If MEMOR $\neq 0$

8.b.1.b.2.a. Send LE(ID) on $\overline{X}$

8.b.1.b.2.b. *SLE* $\leftarrow 1$

8.b.1.b.3 *SLT* $\leftarrow 0$

8.b.2. If *ID* $\neq$ *MyID and B=0*

8.b.2.a. Send LT(ID, 1) on $\overline{X}$

8.b.2.b. If *SLT = 1* MEMORY $\leftarrow$ ID

8.b.2.c. *SLE* $\leftarrow 0$

8.c If not loopback

8.c.1 If ID $\neq$MyID Send LT(ID,B) on X.

9. Receive LE(ID) on X

9.a. If loopback

9.a.1. If *SLT = 0*

9.a.1.a If *SLE =0* **or** *ID* $\leq$ *MyID* Generate Token

9.a.2.a. *SLE* $\leftarrow 0$

9.a.2. If *SLT = 1*

9.a.2.a. If ID=MEMORY

9.a.2.a.1. *SLT* $\leftarrow 0$

9.a.2.a.2. Generate Token

9.b If not loopback

9.b.1. If ID $\neq$MyID Send LE(ID) on X

10. START issued

10.a. If loopback

10.a.1 Send LT(ID, 1)

10.a.2 *SLT* $\leftarrow 0$

10.a.3 *SLE* $\leftarrow 0$

10.a.4 MEMORY $\leftarrow$ 0

To prove the correctness, we first show that, when the algorithm is invoked, tokens will be generated in every subloop and then show that, actually, exactly one is generated. We assume the algorithm is initiated by issuing a START at a looping-back node.

Proposition 4: Within a finite time after a START is given either a new token is generated or another START is given by a node that is aware of the original START.

Proof: Assume a START is given by node A and no other START (fulfilling the condition of the proposition) is given. Thus the LT(ID=A,0) sent as a result of the START will not become an orphan for otherwise a new START must have been given. Eventually an LT(ID=A,1) message is generated which, by to the same argument, will not become orphan and will return to node A. There are three possibilities at this point:

1. *SLT =1 and MEM=0*: a token is generated.
2. *SLT =1 and MEM* $\neq$*0*: an *LE* message is sent.
3. *SLT =0*.

Case 1 complies with the proposition so it is left to prove the other two cases.

Case 2. We trace the LE message when it arrives at another looping back node, say B (which must exist or a new START will be issued). If a token is not generated by node B (steps 9a1a or 9a2a2) then

1. *SLE =1* meaning that B itself sent an LE message that will arrive at node A where step 9a1a will be executed.
2. *SLT =1* meaning that node B sent LT(ID=B,0) after it has responded to node A's LT message, and is therefore considered a new START. (This will not lead to a deadlock since node A will not generate another LT message.)

Case 3. The arriving LT message is discarded, and the following is the sequence of events. When node A first sent the LT message it also set its *SLT* to 1, meaning that it was reset in the meantime, an event that can only happen (without generating a token) at step 8b1b3, i.e., after having received its own LT message. Repeatedly applying this arguments leads to a situation where node A received its own LT message when its *SLT* is set to 1, and one of the previous cases applies. □

Proposition 5: At most one token can be generated as a result of a START.

Proof: Since only the two looping back nodes can generate tokens, at most two tokens can be generated. This can be done either in step 8b1b1 or in step 9a. Suppose both have generated the token by executing 8b1b1; thus, both have received their own LT message back meaning that each has seen the other's message, meaning that in each $MEMORY \neq 0$ and a token will not be generated.

Suppose node A generated the token by executing step 8b1b1 while B generated it by executing step 9a. Node B received an LE message from node A which must have $MEMORY \neq 0$. But to generate a token in step 8b1 the node must have $MEMORY=0$, meaning that node A must have sent the LE message after generating a token, but doing so sets *SLT* to 0 (step 8b1b3) excluding the execution of step 8b1b1.

Suppose, finally, that both generated the token by executing step 9a. Both have received the other's LE message, so each had sent that message and must have its *SLE* set to 1 meaning that its *SLT* is 0 (step 8b1b3) so that step 9a2 is excluded and step 9a1a is executed because the ID relation is satisfied, but clearly this can be satisfied by only one of the two nodes. □

Proposition 6: During reconfiguration all old tokens are destroyed.

Proof: The algorithm is invoked by a START and terminated by a generation of a token (either in step 8 or step 9). Step 9 is always preceded by some node executing step 8 (since 8b1b2a is the only step in which an LE message is generated). Thus the generation of a token implies that at least one node executes 8b1. However, for a node to have received its own LT message with the bounce bit set to 1, all other nodes in the subloop must have received an LT message and by virtue of step 8a have destroyed all old tokens. □

As a result of these last three propositions it is guaranteed that when a loopback configuration remains exactly one token exists in every subloop.

3.3.2 Fitting It Together

The token control algorithm presented in the previous subsection must be integrated with the global reconfiguration algorithm. Our strategy is as follows: whenever a topological change takes place that may effect the subloop configuration, the subloop token control algorithm is invoked. In addition, to cover the possibility that a subloop configuration is no longer needed, the global loop token control algorithm is concurrently invoked whenever link recovery takes place. The integration therefore requires that:

1. A subloop token control algorithm be invoked whenever a change in subloop configuration takes place
2. The loop configuration algorithm override the subloop reconfiguration algorithm

Changes in the subloop topology are necessarily tied to loopback changes. Consequently, in compliance with the first requirement, nodes that modify their loopback are natural candidates to invoke the subloop token control algorithm. We now show exactly how this is performed.

We distinguish, as before, two cases--loopback introduction (as a result of link failure) and loopback removal (as a result of link recovery). To deal with the first case we require that every node introducing a loopback (steps 1e, 3d, or 6b) issue a START, i.e., send an LT message with its own ID and set its *SLT* variable to 1.

The case of loopback removal is more intricate since the subloop token control algorithm must be STARTed only by a node that is looping back. Unfortunately, the node sensing the recovery no longer loops back and can therefore not participate. To deal with this situation we make use of the fact that upon link recovery a CT message is sent (step 2a). If a subloop configuration remains, this CT message eventually arrives at a looping back node and instigates a START at it. If a subloop configuration does not remain, the CT message returns to the originator without causing a START to be generated anywhere. This modification is achieved by adding the following step:

4.c. if loopback
  4.c.1. Send LT on $\overline{X}$
  4.c.2. $SLT_{\overline{X}} \leftarrow 1$

To ensure fulfillment of requirement 2, a *CE* message must reset the *SLT* and *SLE* flags for both loops (to be done in step 5d). Once this is done orphan *LT* and *LE* messages that may be floating in the network must be destroyed; this is achieved by ensuring that steps 8c1 and 9b1 are executed only if neither status bit (S_A or S_B) are set to 1.

The above scheme while correct has a slight inefficiency. The CT message generated upon recovery may pass through subloops that do not require reconfiguration, unnecessarily invoking there the token control algorithm. To avoid this, an extra bit could be added to the CT message, ensuring that a CT message invokes only a single token control algorithm.

Finally a word about token destruction. The token generating algorithm generates a new token and requires that previous tokens, if any, be destroyed. To achieve this, the following rule is adopted. A

node owning the token and receiving either a CE or LT message destroys the token. To ensure that these messages do not “miss” the token (for example when the token and messages progress in opposite directions) we further require that nodes receiving a token when either *SCE =1* or *SLT =1* destroy the token. Since both CE and LT messages return to their originator, token destruction is guaranteed.

References

[1] J.H. Saltzer and K.T. Pogran. A Star Shaped Ring Network with High Maintainability. In *Proc. of the Local Area Network Symposium*, pp. 179–189, 1979.

[2] C.S. Raghvendra, M. Gerla, and A Avizienis. Reliable Loop Topologies for Large Local Computer Networks. *IEEE Trans. on Computers*, C–34(1):46–55, 1985.

[3] M.T. Liu, D.P. Tsay, C.P. Chou, and C.M. Li. Design of the Distributed Double Loop Computer Network (DDLCN). *Journal of Digital Systems*, 5(1–2):3–37, 1981.

[4] Y.J. Tang and S.G. Zaky. A Survey of Ring Networks: Topology and Protocols. In *Proceeding of IEEE Infocom 85*, pp. 318–325, Washington, D.C., 1985.

[5] J.R. Pierce. Network for Block Switching Data. In *Bell System Techincal Journal*, 51(6): 1133-1145, 1972

[6] A. Itai and M. Rodeh. Symmetry Breaking in Distributed Networks. In *Proc. of the 22nd Annual Symposium on the Foundations of Computer Science*, pp. 150–158. IEEE, Nashville, Tennessee, 1981.