September 1978 LIDS-P-848

# A RECOVERABLE PROTOCOL FOR LOOP-FREE DISTRIBUTED ROUTING

A. Segall*, P.M. Merlin* and R.G. Gallager**

## Abstract

An algorithm for adaptive routing in data-communication networks is presented. The algorithm uses distributed computation, provides loop-free routing for each destination in the network, adapts to changes in network flows and is completely failsafe. The latter means that after arbitrary failures and additions of nodes and links, the network recovers in finite time in the sense of providing routing paths between all physically connected nodes. Proofs of all these properties are provided in a separate paper.

---

The work of A. Segall and R.G. Gallager was supported in part by the Advanced Research Project Agency of the US Department of Defense (monitored by ONR) under Contract No. N00014-75-C-1183, and in part by Codex Corporation.

* Department of Electrical Engineering, Technion - Israel Institute of Technology, Haifa, Israel.

** Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., U.S.A.

# I. INTRODUCTION

Reliability and the ability to recover from topological changes are properties of utmost importance for smooth operation of data-communication networks. In today's data networks it happens occasionally, more or less often depending on the quality of the individual devices, that nodes and communication links fail and recover; also new nodes or links become operational and have to be added to an already operating network. The reliability of a computer-communication network, in the eyes of its users, depends on its ability to cope with these changes, meaning that no breakdown of the entire network or of large portions of it will be triggered by such changes and that in finite - and hopefully short - time after their occurrence, the remaining network will be able to operate normally. Unfortunately, recovery of the network under all conditions, namely after arbitrary number, timing and location of topological changes is a property that is very hard to insure and little successful analytical work has been done in this direction so far.

The above reliability and recovery problems are difficult whether one uses centralized or distributed routing control. With centralized routing, one has the problem of control node failure plus the chicken and egg problem of needing routes to obtain the network information required to establish routes. Our primary concern here is with distributed routing; here one has the problems of asynchronous computation of distributed status information and of designing algorithms which adapt to arbitrary changes in network topology in the absence of global knowledge of topology. In previous works [1], [2], minimum delay routing procedures using distributed computation were developed. If the topology of the network is fixed, these algorithms maintain a loop-free routing at each step, and if furthermore the input traffic requirements are stationary, the algorithms bring the network to the minimum delay routing.

The basic algorithm in this paper is similar to, but somewhat simpler than the algorithms of [1], [2]. Here we do not seek optimality, but are still interested in maintaining a loop-free distributed adaptive routing. One of the main contributions of the algorithm given in the present paper is to introduce features insuring recovery of the network from arbitrary topological changes. As such, the resulting algorithm to be presented in Section II is, to our knowledge, the first one for which all of the following properties hold and are rigorously proved:

(a) Distributed computation.

(b) Loop-freedom for each destination at all times.

(c) Independently of the sequence, location and quantity of topological changes, the network recovers in finite time.

The algorithm provides a protocol using distributed computation for building routing tables. For each destination, the routing table at each node $i$ consists of a preferred neighbor $p_i$ and an estimated minimum distance $d_i$ to the destination (the distance is measured in terms of possibly time-varying link weights). Property (b) above means that the links $(i,p_i)$ never form a loop. According to property (c), the algorithm insures that a finite time after (arbitrary) topological changes happen, all nodes physically connected to the destination will form a single tree defined by the relationship $(i,p_i)$ with the root at the destination. These properties are stated formally in Section III and rigorously proved in [11]. We may also note that, since we are concerned here only with building routing tables, the algorithm can be used in (actual or virtual) line switching, as well as in message or packet switching or any combination thereof. Finally, the algorithm has the property of not employing any time-out in its operation, a feature which greatly enhances its amenability to analysis.

In addition to the introduction of the algorithm and the proofs of its main properties, the paper provides contributions in the direction of modeling, analysis and validation of distributed algorithms. The operations required by the algorithm at each node are summarized as a finite-state machine, with transitions between states triggered by the arrival of updating messages. During the activity of the algorithm, messages are sent between neighbors, queued at the receiving node and then processed on a first-come-first-served basis. The processing of a message consists of its temporary storage in an appropriate memory location, followed by activation of the finite-state machine, which takes the necessary actions and performs the corresponding state transitions. In addition to its state, each node has a set of memory items (i.e. variables) that are used as "context" in the execution of the finite-state machine. Thus, predicates on the value of those variables can be used as conditions for transitions to occur, and the occurrence of transitions may change the value of the variables.

Methods for modeling and validation of various communication protocols are proposed in [3] - [6]. These methods are designed however to handle protocols involving either only two communicating entities or nodes connected by a fixed topology. The model we use to describe our algorithm is a combination of these known models, but is extended to allow us to study a fairly complex distributed protocol, where arbitrary failures and added links and nodes cause topological changes. The analysis and validation of the algorithm is performed by using a special type of induction to be described in Section III that allows us to prove in [11] global properties while essentially looking at local events.

Several routing algorithms possessing some of the properties indicated above have been previously indicated in the literature. In [9], a routing algorithm similar to the one used in the ARPA network, but with unity link

weights, is presented. It is shown that at the time the algorithm terminates, the resulting routing procedure is loop-free and provides the shortest-paths to each destination. As with the ARPA routing, however, the algorithm allows temporary loops to be formed during the evolution of the algorithm. The algorithm proposed in [10] ensures loop-free routing for individual messages. This property is achieved by requesting each node to send a probing message to the destination before each individual rerouting; the node is allowed to indeed perform the rerouting only after receiving an acknowledgement from the destination. Loop freedom for individual messages is a weaker property than loop freedom for each destination. For example, in a three-node network, sending traffic from node 3 to node 1 via node 2 and sending traffic from node 2 to node 1 via node 3 would be loopfree for individual messages, but not loopfree for each destination. See [12] for a more complete discussion of loop freedom.

## II. THE ALGORITHM

The algorithm is described in several steps. We first present the operations to be performed at a node in "normal" conditions, when no topological changes occur. Then we describe the addition to the algorithm in case of failures and finally the protocols for adding a link to the network. After the various features of the algorithm are introduced and explained, we proceed to present the formal algorithm for each node in the network.

### Informal Description of the Algorithm

The algorithm proceeds independently for each destination. Consequently, for the rest of the paper we fix the destination and present and analyse the algorithm for that given destination, which is denoted by SINK. In normal conditions, each node $i$ in the network has a routing table for this destination consisting of a preferred neighbor $p_i$, a node counter number $n_i$ and an estimated distance $d_i$ to the destination. Essentially, the algorithm is intended to update or establish these quantities at each step, to bring up links or nodes that are ready to be added to the network and to

eliminate links or nodes that have left the network. In addition to the quantities $p_i$, $n_i$, $d_i$, a node $i$ keeps a list of its current neighbors, named $LIST_i$, and for each node $k \in LIST_i$ it keeps two memory locations called $N_i(k)$ and $D_i(k)$, intended for storage of messages received from $k$. During the update activity, messages with format (SINK,m,d) are transmitted between neighbors, where, if $\ell$ is the sender, then $m = n_\ell$ and $d = d_\ell$. (Since we are looking at a particular SINK, we shall suppress this parameter from now on). After appending the identification of the sender to each received message, the receiving node puts the messages in a queue and processes them one by one on a first-come-first-served (FIFO) basis. We say that a message is received at node $i$ at time $t$, if the processor at node $i$ starts processing the message at time $t$. As a rule, the first part of processing of a message $(m,d)$ received at $i$ from $\ell$ say, consists of adding to $d$ the current distance $d_{i\ell}$ from $i$ to $\ell$ and then storing $m$ and $(d + d_{i\ell})$ in $N_i(k)$, $D_i(k)$ respectively. The distance $d_{i\ell}$ can be the estimated delay over the link $(i,\ell)$ (as e.g. in ARPA), the estimated incremental delay over $(i,\ell)$ as required in [1],[2], or any other quantity reflecting the routing criterion of interest. This quantity must be positive, but can be estimated in an arbitrary manner at each node $i$ for each outgoing link. Procedures for estimation of the incremental delay are indicated in [7], [8].

An update cycle is started when the SINK sends a message $(m, d = 0)$ to all its neighbors. Let us look now at an arbitrary node $i$ in the network, and describe its procedure under "normal" conditions, namely when no topological changes occur. A node $i$ collects all messages it receives from neighbors and stores them as described above; it does nothing else until a message is received from the preferred neighbor $p_i$. At this point the node enters an "alert" state named S2, updates its $d_i$ as the minimum of all $D_i(k)$ received up to now during the present update cycle and sends (the updated) $d_i$ to all neighbors, except its preferred neighbor $p_i$. After completion of these

operations, the node continues to collect and store messages, until messages are received from ALL its neighbors. At this point, node $i$ sends $d_i$ to its preferred neighbor $p_i$, then updates $p_i$ to be the node with minimum $D_i(k)$ among all neighbors, erases all stored values $N_i(\cdot)$ and returns to the "normal" state named S1. The idea of this part of the algorithm is that, since, as well be seen in Section III, the relation $(i,p_i)$ defines a tree in the network in normal conditions, the update cycle will propagate from the SINK to the peripheries while updating $d_i$ and then back from the peripheries to the SINK while updating the tree. The transition of node $i$ from state S2 to state S1 signifies completion of the update step for node $i$. In particular, transition from S2 to S1 of the SINK (the SINK enters state S2 when starting the cycle) means completion of the update cycle by the entire network.

Until now, we have not described the role of the cycle counter number $m$ and of the node counter number $n_i$. Indeed, if no topological changes occur, there is no need for these numbers as long as the SINK starts no new update before the previous one has been completed. It is easy to see that completion of each cycle is guaranteed to occur in finite time if there are no failures and if transmission of messages over any link takes a finite time. The formal statement of this property is included in Theorem 4.

Next, we describe the additions and changes to the basic algorithm for proper treatment of topological changes. It is here that the role of the cycle and node counter numbers $m$ and $n_i$ becomes apparent. The SINK starts consecutive update cycles with nondecreasing counter numbers. If a cycle is completed, the SINK is allowed to start a new cycle with the same counter number provided that a cycle with higher number has not been previously started. On the other hand, when topological changes occur, nodes in the network may request (by a distributed protocol to be described presently), that the SINK will start a cycle with a counter number that is higher than a specified number. In this case, if the SINK has not started such an update cycle before, it will

start it immediately and will never reduce the update counter numbers afterwards. For instance, a sequence of starts and ends of update cycles with their appropriate counter numbers may be: start 1, end 1, start 1, start 2, end 2, start 2, ... . If the SINK starts a cycle with counter number $m$ and completes it before starting a new cycle, we say that there has been a <u>proper completion</u>. We denote the <u>time</u> of proper completion of a cycle with number $m$ by $PC(m)$.

If a node $i$ discovers a failure on link $(i,p_i)$, it enters a "listening" state S3, sets $d_i \leftarrow \infty$ sends $(m = n_i,\ d = \infty)$ to all neighbors except $p_i$, sets $p_i \leftarrow nil$ and deletes the neighbor corresponding to the failed link from the list of neighbors $LIST_i$. A node $i$ receiving $(m,\ d = \infty)$ from $p_i$, performs similar operations, except that it also sets $n_i \leftarrow m$ and stores $(m,d)$ into $(N_i(p_i),D_i(p_i))$, but does not delete $p_i$ from $LIST_i$. In this way, the knowledge of the failure propagates backwards to all nodes whose best paths to the SINK passed through the failed link or node and to their neighbors. A node $i$ that loses its preferred neighbor by this operation goes into state S3 and reattachment (i.e. establishing a new preferred neighbor $p_i$) will occur as soon as it receives a message $(m,d)$ such that $m > n_i$ and $d < \infty$. If at the time it enters S3, node $i$ has already received such a message, it reattaches immediately. Reattachment consists of choosing as the new $p_i$ to be the node from which such message was received, going to state S2 and updating $n_i$. On their part, the neighbors of the nodes in S3 will know not to choose such nodes as their preferred neighbors.

Up to now, we have described the algorithm of a node $i$ in case failure occurs on a link $(i,p_i)$. If failure occurs on a link other than the preferred one, this link is erased from $LIST_i$, but no special action is needed except if the node is in state S2. In such situations it may happen (at this time or later) that the node will receive messages from all the re-

maining neighbors and will complete its part in the step of the algorithm by the usual transition to S1. This is a situation we would like to avoid, because the transition from S2 to S1 is supposed to signal proper completion of a step of the algorithm by the corresponding node and in the case under consideration the node will complete the step because it lost one of its neighbors and not because it received the appropriate signal from it. Consequently the algorithm dictates that if a node $i$ is in state S2 and discovers a failure on links other than $(i,p_i)$, the node will go into a "stagnated" state $S\tilde{2}$, from which it will not move until it receives a message over its preferred link. In this case, it will return to S2 and will continue the algorithm as usual. In Section III and Appendices, we show that proper advance of the algorithm after this transition is guaranteed. Another possible transition is from state S2 back to state S2. This happens when a node is in state S2 and receives a message (m,d) with $m$ large enough and $d < \infty$ from its preferred neighbor. From this point on, the node will proceed as usual. The state diagram for a node is depicted in Figure 1.

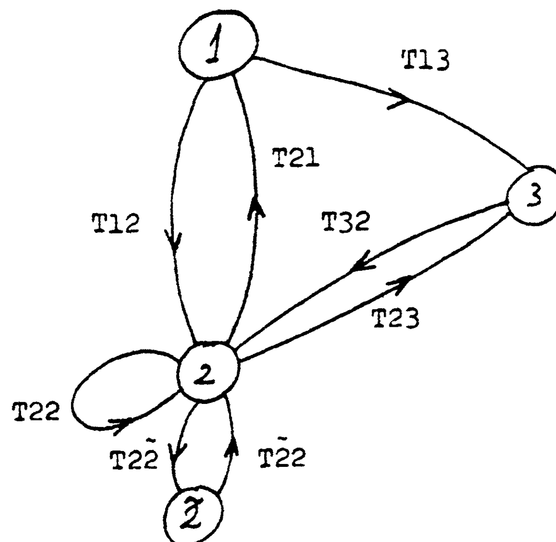


Fig. 1 - State diagram for a node.

In addition to the above operations, any node discovering a failure on any of the links connected to it generates a message called REQ($n_i$). The number $n_i$ is exactly the node counter number of the generating node. A node that generates or receives REQ(m), sends it over to its preferred neighbor $p_i$ if $p_i \neq$ nil; otherwise it destroys the message. When a message REQ arrives at a node, it is put in the regular queue and processed according to FIFO as all other messages. When the SINK receives REQ(m) it starts a new cycle with counter number (m+1), provided that such cycle has not been started previously. Proposition 2 guarantees that if a REQ(m) is generated, a cycle with counter number (m+1) will always be started within finite time.

We finally describe the protocol for adding components to the network. A node i comes up in state S3 with counter number $n_i = 0$. For bringing a link up (i,k) say, its two ends compare their node numbers $n_i$ and $n_k$ (via a local protocol) and decide that they will bring the link up with a number strictly higher than

$$z_i(k) = z_k(i) = \max(n_i, n_k) \ . \qquad (1)$$

Also, if $n_i \geq n_k$, then i generates REQ($n_i$). The link is finally brought up by node i when $n_i > z_i(k)$ <u>or</u> when it receives from k a message (m,d) with $m > z_i(k)$. The same algorithm holds for k. Bringing link (i,k) up at node i consists of appending k to $\text{LIST}_i$ and opening memory locations $N_i(k)$, $D_i(k)$. In the formal description of the algorithm, this is done in B.1 and SUBROUTINE NEW. Proposition 2 guarantees that a cycle with counter number higher than $z_i(k)$ will be started in finite time after the REQ message is generated.

## Notations

In this subsection, we present several notations that will be used in the rest of the paper. The notations $p_i$, $(m,d)$, $n_i$, $d_i$, $N_i(k)$, $D_i(k)$, $z_i(k)$, $LIST_i$, S1, S2, $\tilde{S2}$, S3, PC(m), have been introduced already. We add the time in parentheses when we want to refer to the above quantities at a given time $t$; for example $p_i(t)$, $N_i(k)(t)$, etc. We also use the notations:

SX[n] = state SX with node counter number $n$.

$s_i(t)$ = state and possibly node counter number $n_i$ of node $i$ at time $t$. Therefore we sometimes write $s_i(t) = S3$ for instance and sometimes $s_i(t) = S3[n]$.

$mx_i(t)$ = largest counter number $m$ received up to time $t$ by node $i$.

$ADD_i$ = list of nodes $k$ neighboring $i$ such that link $(i,k)$ is ready to be added to the network.

We use a compact notation to describe changes accompanying a transition, as follows:

Txy[t,i,RECV(m1,d1,ℓ1),SEND(m2,d2,ℓ2),(n1,n2),(d1,d2),(p1,p2),(mx1,mx2)]

will mean that transition from state Sx to state Sy takes place at time $t$ at node $i$ caused by receiving (m1,d1) from neighbor ℓ1; in this transition $i$ sends (m2,d2) to ℓ2, changes its node counter number $n_i$ from n1 to n2, its estimated distance to destination $d_i$ from d1 to d2, its preferred neighbor $p_i$ from p1 to p2 and the largest update

counter number received up to now $mx_i$ from mx1 to mx2. For simplicity, we delete all arguments that are of no interest in a given description, and if for example n1 is arbitrary we write $(\phi,n2)$ instead of (n1,n2). Similarly, if one of the states is arbitrary, $\phi$ will replace this state. In particular observe that

$$T\phi 2[t,SINK,(\phi,n2)] \quad (2)$$

means that an updating cycle with number n2 is started at time t and

$$T21[t,SINK,(n2,n2)] \quad (3)$$

means that proper completion of the cycle occurs at time t. If Txy[t], then we use the notations:

t- = time just before the transition ,

t+ = time just after the transition .

We also use

$$[t,i,RECV(m,d,\ell)] \quad (4)$$

to denote the fact that a message (m,d) is received at time t at i from $\ell$, whether or not the receipt of the message causes a transition.

Assumptions and Definitions

Throughout the paper, we assume that the following hold everywhere in the network.

1. All links are bidirectional (full duplex).

2. $d_{i\ell}(t) > 0$ for all links $(i,\ell)$ and all t.

3. If a message is sent by node i to a neighbor $\ell$, then in finite time, either the message will be received correctly at $\ell$ or

$\ell$ will declare link $(i,\ell)$ as failed. Notice that this assumption does not preclude transmission errors that are recovered by a local link protocol (e.g. "resend and acknowledgment").

4. Failure of a node is considered as failure of all links connected to it. A node $i$ comes up in state S3, with $n_i = 0$ and with empty tables.

5. A link is said to have become operational as soon as messages can be sent both ways through it.

6. A link $(i,k)$ is said to be up if $i \in LIST_k$ or $k \in LIST_i$, or both.

7. Two nodes are said to be physically connected at time $t$ if there is a sequence of links that are up at time $t$ connecting the two nodes.

8. A message is said to arrive at node $i$ when it physically arrives there. A message is said to be received at node $i$, when the message is taken from the queue and the message processor starts processing it.

9. When a message $(m,d)$ is received at a node, the possible values of $d$ may be numerical, $d = \infty$ or $d = \text{FAIL}$. In the following, $d < \infty$ means $d \neq \infty$ and $d \neq \text{FAIL}$. Also $D_i(k) < \infty$ means $D_i(k) \neq \infty$ and $D_i(k) \neq \text{FAIL}$.

10. The local protocols for discovering failures and declaring links operational are arbitrary, provided they possess the following properties:

    (a) If node $i$ declares link $(i,k)$ to be failed then node $k$ declares link $(i,k)$ to be failed in finite time or node $i$ will try to reopen the link in finite time.

    (b) If node $k$ notices that node $i$ tries to bring link $(i,k)$ up while node $k$ still considers the link operational, node $k$ first declares link $(i,k)$ as failed (and performs step A.4 in the Formal Description of the Algorithm) and then proceeds to reopen the link.

Formal Description of the Algorithm

We are now ready to display the formal algorithm performed by node $i$. We divide the description into three parts; the off-line operations, the message processor and the finite-state machine.

The off-line operations are performed independently of the message processor and are triggered by a message arrival to the node, by link failures detected at the node or by new links becoming operational. The main processor takes the message at the head of the queue and starts processing it. The main part of the processing consists of the finite-state machine being called and zero, one or more transitions taking place. As soon as no more transitions are possible, the action is returned to the message processor.

The "Facts" given in the algorithm are displayed for helping in its understanding and are proven in Theorem 2.

A. Off-Line Operations

A.1 Compiling the list of newly operational links

If $(i,\ell)$ becomes operational, set

$$z_i(\ell) \leftarrow \max\{n_i, n_\ell\} ;$$

Append $(\ell, z_i(\ell))$ to $ADD_i$. If $n_i \geq n_\ell$, generate $REQ(n_i)$ and put it in queue (if $n_i = n_\ell$, it is enough if only one of the nodes generates the REQ).

A.2 Message $(m,d)$ arrives at node $i$ from node $\ell$ such that $\ell \in LIST_i$ or $\ell \in ADD_i$

Put $(m,d,\ell)$ in queue;

A.3 Message $REQ(m)$ arrives at node $i$

Put message in queue

A.4 Failure of link $(i,\ell)$ detected at node $i$

Set

$$d \leftarrow FAIL, \quad m \leftarrow FAIL ;$$

Put $(m,d,\ell)$ in queue. Generate $REQ(n_i)$ and put it in queue.

B. Operations Done by the Message Processor when a Message is Received

(i.e., when processor at $i$ takes this message from the queue and starts processing it).

If the message is $REQ(m)$, send it to $p_i$ if $p_i \neq nil$ and destroy it if $p_i = nil$.

If the message is $(m,d,\ell)$, then:

B.1 Open new links:

If $\ell \in ADD_i$, then if $m > z_i(\ell)$, append $\ell$ to $LIST_i$ and delete it from $ADD_i$.

B.2 Execute

If $d \neq FAIL$, $d \neq \infty$, then $d \leftarrow d + d_{i\ell}$;

$N_i(\ell) \leftarrow m$;

$D_i(\ell) \leftarrow d$;

If $m \neq FAIL$, then $mx_i \leftarrow \max\{m, mx_i\}$;

EXECUTE FINITE-STATE MACHINE;

IF $m = FAIL$, delete $\ell$ from $LIST_i$.

C. Finite-State Machine

State S1

T12 Condition 12 $\ell = p_i$, $d < \infty$, $m = mx_i$.

Fact 12 $m \geq n_i$.

Action 12 $d_i \leftarrow \min_{\substack{k: N_i(k) = m \\ D_i(k) < \infty}} D_i(k)$;

$n_i \leftarrow m$;

$\forall k \in ADD_i$, if $n_i > z_i(k)$, CALL "NEW (k)";

transmit $(n_i, d_i)$ to neighbors, except $p_i$.

T13 Condition 13 $\ell = p_i$ and $(d = \infty$ or $d = FAIL)$.

Fact 13 If $m \neq FAIL$, then $m \geq n_i$.

Action 13 $d_i \leftarrow \infty$;

If $d \neq FAIL$, then $n_i \leftarrow m$;

$\forall k \in ADD_i$, if $n_i > z_i(k)$, CALL "NEW (k)";

transmit $(n_i, d_i)$ to neighbors, except $p_i$;

$p_i \leftarrow nil$.

State S2

T21 Condition 21 $\forall k$ $LIST_i$, $N_i(k) = n_i = mx_i$;

$\exists k$ $LIST_i$, s.t. $D_i(k) \leq d_i$;

$D_i(p_i) < \infty$;

$d \neq FAIL$.

Fact 21 $d_i < \infty$.

Action 21 Transmit $(n_i, d_i)$ to $p_i$;

$p_i \leftarrow k^*$ that achieves min $D_i(k)$;

$\forall k \in LIST_i$, set $N_i(k) \leftarrow nil$;

exit Finite-State Machine.

T22 Condition 22 $\ell = p_i$, $d < \infty$, $m = mx_i > n_i$.

Action 22 Same as Action 12.

$T\tilde{22}$ Condition $\tilde{22}$ $\ell \neq p_i$, $d = FAIL$.

Action $\tilde{22}$ NONE.

T23 Condition 23 Same as Condition 13.

Fact 23 Same as Fact 13

Action 23 Same as Action 13.

State S3

T32 Condition 32 $\exists k \in LIST_i$ such that $mx_i = N_i(k) > n_i$, $D_i(k) < \infty$.

Fact 32 $p_i = nil$, $d_i = \infty$.

Action 32 Let $k^*$ achieve $\min_{\substack{k:N_i(k) = mx_i \\ D_i(k) < \infty}} D_i(k)$

Then $p_i \leftarrow k^*$;

$n_i \leftarrow mx_i$;

$d_i \leftarrow D_i(k^*)$;

$\forall k \in ADD_i$, if $n_i > z_i(k)$, CALL "NEW(k)".

Transmit $(n_i, d_i)$ to neighbors, except $p_i$.

State $\tilde{S2}$

$\tilde{T22}$ Condition $\tilde{22}$ $\ell = p_i$

Fact $\tilde{22}$ If $m \neq$ FAIL, then ($m \geq n_i$ and if $m = n_i$, then $d = \infty$).

Action $\tilde{22}$ None

SUBROUTINE "NEW(k)"

Append $k$ to $LIST_i$;

delete $k$ from $ADD_i$;

set $N_i(k) \leftarrow nil$.

This completes the description of the algorithm for all nodes in the network, except the SINK. The SINK performs the following algorithm:

Off-Line Operations

Same as for all other nodes and in addition, if $s_{SINK} = S1$, the SINK can start a new update cycle at any time by going to S2 and transmitting $(n_{SINK}, d = 0)$ to all neighbors.

Operations Done by the Message Processor when a Message is Received

If the message is REQ(m) and $n_{SINK} > m$, destroy the message.

If the message is REQ(m) and $n_{SINK} \leq m$, go to state S2 and start a new cycle as follows:

$n_{SINK} \leftarrow (m+1)$;

$\forall k \in ADD_{SINK}$, CALL "NEW(k)";

transmit ($n_{SINK}$, $d_{SINK} = 0$) to all neighbors.

If the message is $(m,d,\ell)$, d = FAIL, then delete $\ell$ from $LIST_{SINK}$.

If the message is $(m,d,\ell)$, $d \neq$ FAIL, then

$N_i(\ell) \leftarrow m$;

EXECUTE FINITE-STATE MACHINE FOR SINK.

FINITE-STATE MACHINE FOR SINK

State S2

T21 Condition 21 $\forall k \in LIST_{SINK}$, $N_i(k) = n_{SINK}$.

Action 21 $\forall k \in LIST_{SINK}$, set $N_i(k) \leftarrow$ nil.

## III. Properties and Validation of the Algorithm

Some of the properties of the algorithm have already been indicated in previous sections. Here we state them explicitly along with some of the others. We start with properties that hold throughout the operation of the network, some of them referring to the entire network at a given instant of time and some to a given node or link as time progresses. Then we address the problem of recovery of the network after topological changes.

At a given instant t, we define the Routing Graph RG(t) as the directed graph whose nodes are the network nodes and whose arcs are given by

the pointers $p_i$; namely there is an arc from node $i$ to node $\ell$ if and only if $p_i(t) = \ell$. In order to describe properties of the RG(t), we also define an order for the states by $S3 > S2 = S\tilde{2} > S1$. Also $Sx \geq Sy$ means $sx > sy$ or $sx = sy$. For conceptual purposes, we regard all the actions associated with a transition of the finite-state machine to take place at the instant of the transition.

Theorem 1

At any instant of time, RG(t) consists of a set of disjoint trees with the following ordering properties:

i) the roots of the trees are the SINK and all nodes in S3;

ii) if $p_i(t) = \ell$, then $n_\ell(t) \geq n_i(t)$;

iii) if $p_i(t) = \ell$ and $n_\ell(t) = n_i(t)$, then $s_\ell(t) \geq s_i(t)$;

iv) if $p_i(t) = \ell$ and $n_\ell(t) = n_i(t)$ and $s_\ell(t) = s_i(t) = S1$, then $d_\ell(t) < d_i(t)$.

The proof of Theorem 1 is given in [11]. According to it, the RG consists at any time of a set of disjoint trees or equivalently, it contains no loops. Observe that a tree consisting of a single isolated node is possible. The algorithm maintains a certain ordering in the trees, namely that concatenation of $(n_i,s_i)$ is nondecreasing when moving from the leaves to the root of a tree and in addition, for nodes in S1 and with the same node counter number, the estimated distances to the SINK are strictly decreasing.

In addition to properties of the entire network at each instant of time, we can look at local properties as time progresses. Some of the most important are given in the following theorem whose proof appears in [11].

Theorem 2

i) For a given node $i$, the node counter number $n_i$ is nondecreasing and the messages $(m,d)$ received from a given neighbor have nondecreasing numbers $m$.

ii) Between two successive proper completions $PC(\bar{m})$ and $PC(\bar{\bar{m}})$, for each given $m$ with $\bar{m} \le m \le \bar{\bar{m}}$, each node sends to each of its neighbors at most one message $(m,d)$ with $d < \infty$.

iii) Between two successive proper completions $PC(\bar{m})$ and $PC(\bar{\bar{m}})$, for each given $m$ with $\bar{m} \le m \le \bar{\bar{m}}$, a node enters each of the sets of states $\{S1[m]\}$, $\{S2[m], \tilde{S2}[m]\}$, $\{S3[m]\}$ at most once.

iv) All "Facts" in the formal description of the algorithm in Section II are correct.

A third theorem describes the situation in the network at the time proper completion occurs:

Theorem 3

At $PC(\bar{m})$, the following hold for each node $i$:

i) If $n_i = \bar{m}$, then $s_i = S1$ or $s_i = S3$.

ii) If a message $(\bar{m},d)$ with $d < \infty$ is on its way to $i$, then $s_i = S3$ and $n_i = \bar{m}$.

iii) If either $(n_i = \bar{m}$ and $s_i = S1)$ or $n_i < \bar{m}$, then for all $k \in LIST_i$, it cannot happen that $\{N_i(k) = \bar{m}, D_i(k) < \infty\}$.

A combined proof is necessary to show that the properties appearing in Theorems 1, 2, 3 hold. The proof uses a two-level induction, first assuming properties at PC to hold, then showing that the other properties hold between this and the next PC and finally proving that the necessary proper-

ties hold at the next PC. The second induction level proves the properties between successive proper completions by assuming that the property holds until just before the current time $t$ and then showing that any possible change at time $t$ preserves the property. The entire rigorous procedure appears in [11].

In order to introduce properties of the algorithm regarding normal activity and recovery of the network, we need the following:

<u>Definition</u>

Consider a given time $t$, and let $m1$ be the highest counter number of cycles started before $t$. We say that a <u>pertinent topological change</u> happens at time $t$ if a node $i$ with $n_i(t-) = m1$ detects at time $t$ a failure of one of its neighboring links or observes at time $t$ that an adjoining link became operational. In other words, a pertinent topological change happens at time $t$ if and only if a message REQ($m1$) is generated at time $t$, where $m1$ is the largest cycle counter number available at time $t$ in the network.

<u>Theorem 4</u> (Normal activity)

Let

$$L(t) = \{\text{nodes physically connected to SINK at time } t\}.$$

Suppose

$$T\phi2[t1,\ \text{SINK},\ (m1,m1)] \qquad (5)$$

namely a cycle is started at $t1$ with a number that was previously used. Suppose also that no pertinent topological changes have happened while $n_{SINK} = m1$ before $t1$ and no such changes happen after $t1$ for long enough time. Then there exist $t0$, $t2$, $t3$ with $t0 < t1 < t2 < t3 < \infty$ such that a), b), c), d) hold:

a) $T21[t0, SINK, (m1,m1)]$; (6)

b) $\forall t \in [t0,t3]$, we have $L(t) = L(t0)$;

c) for all $i \in L(t0)$, we have

$$T\phi2[t2_i, i, (m1,m1)] \quad (7)$$

for some time $t2_i \in [t1,t2]$;

d) i) $T21[t3, SINK, (m1,m1)]$; (8)

ii) $RG(t3)$ for all nodes in $L(t0)$ is a single tree rooted at SINK.

In words, Theorem 4 says that under the given conditions, if a new cycle starts with a number that was previously used, then PC with the same number has previously occurred and the new cycle will be properly completed in finite time. The proof of Theorem 4 is given in [11].

The recovery properties of the algorithm are described in Propositions 1, 2 and in Theorem 5. The proofs of the propositions appear in [11].

<u>Proposition 1</u>

Let $L(t)$ be as in Theorem 4. Suppose

$$T\phi2[t1, SINK, (m1,m2)] \; ; \quad m2 > m1 \; , \quad (9)$$

namely a cycle starts at time $t1$ with a number that was not previously used. Suppose also that no pertinent topological changes happen for a long enough period after $t1$. Then

a) there exists a time t2, with $t1 \leq t2 < \infty$, such that for all $i \in L(t2)$

$$T\phi2[t2_i,i,(\phi,m2)] \quad (10)$$

happen at some time $t2_i$ with $t1 \leq t2_i \leq t2$.

b) There exists a time $t3 < \infty$ such that

i) $$T21[t3,SINK,(m2,m2)) \; ; \quad (11)$$

ii) RG(t3) for all nodes in L(t3) is a single tree rooted at SINK.

Part of a) of Proposition 1 says that under the stated conditions, all nodes in L(t) will eventually enter state S2[m2]. Part b) says that the cycle will be properly completed and all nodes physically connected to the SINK at time PC(m2) will also be connected to the SINK by the Routing Graph.

Finally, we observe that reattachment of a node loosing its path to the SINK or bringing a link up requires a cycle with a counter number higher than the one the node currently has. Proposition 2 ensures that such a cycle has been or will be started in finite time by the SINK.

Proposition 2

Suppose that a message REQ(m1) is generated at some time t at some node in the network. Then the SINK has received before t a message REQ(m1) or will receive such a message in finite time after t.

Propositions 1 and 2 are combined in:

Theorem 5 (Recovery theorem)

Let L(t) be as in Theorem 4. Suppose there is a time t1 after which no pertinent topological changes happen in the network for long enough time. Then there exists a time t3 with $t1 \leq t3 < \infty$ such that proper completion happens at t3 and such that all nodes in L(t3) are on a single tree rooted at SINK.

Proof

Let $t0 \leq t1$ be the time of the last pertinent topological change before $t1$. Let $i$ be the node detecting it and let $m = n_i(t0-)$. Then by definition, a message REQ(m) is generated at time $t0$. By Proposition 2, a message REQ(m) arrives at some finite time at SINK. Let $t2 < \infty$ be the time the first REQ(m) message arrives at SINK. The algorithm dictates that SINK will start at time $t2$ a new cycle, with number $m1 = m+1$. Since by the definition of pertinent change, $m$ is the largest number at time $t0$, we have that $t0 < t2$. By assumption, no pertinent topological changes happen after time $t0$ for a long enough period, so that no such changes happen after time $t2$. Consequently Proposition 1 holds after this time and the assertion of the Theorem follows.

## IV. DISCUSSION AND CONCLUSIONS

The paper presents an algorithm for constructing and maintaining loop-free routing tables in a data-network, when arbitrary failures and additions happen in the network. Clearly, these properties hold also for several other versions of the algorithm, some of them simpler and some of them more involved than the present one. We have decided on the present form of the algorithm as a compromise between simplicity and still keeping some properties that are intuitively appealing. For example, one possibility is to increase the update cycle number every time a new cycle is started. This will not simplify the algorithm, but will greatly simplify the proofs. On the other hand, it will require many more bits for the update cycle and node numbers m and $n_i$ than the algorithm given in the paper. Another version of the algorithm previously considered by us was to require that every time a node

receives a number higher than $n_i$ from some neighbor, it will "forget" all its previous information and will "reattach" to that node immediately, by a similar operation to transition T32. This change in the algorithm would considerably simplify both the algorithm and the proofs, but every topological change will affect the entire network, since after any topological change, all nodes will act as if they had no previous information. On the other hand, the version given in the paper "localizes" failures in the sense that only those nodes whose best path to SINK was destroyed will have to forget all their previous information. This is performed in the algorithm by requiring that nodes not in S3 will wait for a signal from the preferred neighbor $p_i$ before they proceed, even if they receive a number higher than $n_i$ from other neighbors. The signal may be either $\infty$, in which case the node enters S3 (and eventually reattaches) or less than $\infty$, in which case the node proceeds as usual.

## References

[1] R.G. Gallager, A minimum delay routing algorithm using distributed computation, IEEE Trans. on Comm., Vol. COM-25, pp. 73-85, Jan. 1977.

[2] A. Segall, Optimal distributed routing for line-switched data networks, submitted to IEEE Trans. on Comm.

[3] G.V. Bochmann and J. Gecsei, "A unified method for the specification and verification of protocols", Publication #247, Departement d'Informatique, University of Montreal, Nov. 1976. To be presented at the IFIP-Congress 1977, Toronto.

[4] P.M. Merlin, A methodology for the design and implementation of communication protocols, IEEE Trans. on Communications, Vol. COM-24, No. 6, pp. 614-621, June 1976.

[5] C.A. Sunshine, Survey of communication protocol verification techniques, Trends and Applications 1976: Computer Networks, (Symposium sponsored by IEEE Computer Society; National Bureau of Standards), Gaithersburg, Maryland, Nov. 1976.

[6] M.G. Gouda and E.G. Manning, protocol machines: A concise formal model and its automatic implementation. Proceedings of the Third International Conference on Computer Communication, pp. 346-345, Toronto, Aug. 1976.

[7] A. Segall, The modeling of adaptive routing in data-communication networks, IEEE Trans. on Comm., Vol. COM-25, pp. 85-95, Jan. 1977.

[8] M. Bello, Estimation of the delay derivative for purposes of routing in data networks, S.M. Thesis, Dept. EECS, MIT, Feb. 1977.

[9] W.D. Tajibnapis, A correctness proof of a topology information maintenance protocol for a distributed computer network, Communications ACM, Vol. 20, No. 7, pp. 477-485, July 1977.

[10] W.E. Naylor, A loop-free adaptive routing algorithm for packet switched networks, Proc. 4th Data Communication Symposium, Quebec City, pp. 7.9 - 7.14, Oct. 1975.

[11] P.M. Merlin and A. Segall, A failsafe algorithm for loop-free distributed routing in data-communication networks, submitted to IEEE Trans. on Comm.

[12] R.G. Gallager, Loops in multicommodity flows, Paper ESL-P-772, Electronics Systems Laboratory, MIT, Cambridge, Mass., Sept. 1977.