June, 1981

LIDS-P-1099

A Failsafe Distributed Protocol for Minimum Delay Routing

ADRIAN SEGALL, SENIOR MEMBER, IEEE, AND MOSHE SIDI, STUDENT MEMBER, IEEE

Abstract—**Previous distributed routing protocols in data-communication networks that achieve minimum average delay are extended to take into consideration topological changes in the network.**

I. INTRODUCTION

In two recent papers [1], [2], routing protocols have been proposed for data networks using message and virtual line switching, respectively. The main features of these protocols are: distributed computation, loop-free routing for each destination, adaptivity to load changes, and minimum average delay in steady state. The propagation of the protocols presented in [1], [2] is based on control messages received by nodes from certain neighbors and, as such, relies critically on the assumption that the topology of the network is fixed. On the other hand, practical networks contain components (nodes, links, etc.) that are not perfectly reliable and also, in a distributed network, various nodes may join the network at different times. It is therefore essential to provide protocol extensions that will take into consideration topological changes (failures and recoveries of links and nodes) occuring at arbitrary locations and in an arbitrary sequence in the network. This is exactly the purpose of this paper, but because of space limitations, we shall present here explicitly only the extension to the protocol of [2] and refer the reader to [4] for a similar extension to the protocol of [1].

The current work is a natural outgrowth of the protocol introduced in [3], where topological changes were treated for the situation when a single path is maintained at any given time from each node to each destination. This dynamically changing single path can be used in a variety of ways for data routing and several possibilities have been suggested in [3], the main idea being that alternate routing (i.e., traffic split) offers better performance than fixed routing. The exact fractions for splitting the traffic to optimize some general criterion can be obtained, however, only if one maintains at all times all possible "good" routes from each node to each destination. This is possible in a distributed way with the protocols of [1], [2], whose extension to incorporate topological changes is treated in the present paper. The resulting protocol possesses the following features:

1) distributed computation
2) loop-free routing for each destination is maintained in the network at all times
3) adaptivity to slow load changes
4) for stationary input traffic and fixed topology, the protocol reduces network delay during each cycle and minimum average delay is obtained in steady state
5) after arbitrary number, location and sequence of topological changes, the network recovers in finite time in the sense of providing routing paths between all connected nodes (in addition, nodes that are not affected by the topological change continue the algorithm and adapt to the new load pattern in a smooth way).

This last property is very important because it allows the network to continue to function normally, except for the part that is directly affected by the failure. Another important feature of the protocol is that if all routes currently maintained between a given pair of nodes are destroyed, new routes—not necessarily the best—will be found within a short time, allowing data transmission. The routes will then be improved as the protocol evolves.

II. THE PROTOCOL

The protocol of [2] (referred to as the "basic protocol" for the rest of this paper) evolves independently from destination to destination while updating the routes from all nodes to a destination. The extended protocol has the same property and, for the rest of this paper, we shall therefore consider only the protocol that *corresponds to a given destination.* In this section we summarize briefly the basic protocol and then indicate its extensions that take into account topological changes.

A. Summary of the Basic Protocol [2]

The basic protocol is implemented by a sequence of cycles, each started by and terminated at the destination node and each improving the network average delay (provided that the step size of the algorithm, denoted by η, is appropriately chosen [2, Theorem 3]). The terminology upstream, downstream, sons, and loops regarding the flow of data from all nodes to the destination is the same as in [2]: node k is said to be the son of node i if either node i sends traffic to k or node i has no traffic to the destination, but holds link (i, k) as the direction in which it would send possible future incoming traffic. If there is a sequence of nodes $i_1, i_2, \cdots, i_m$ such that i_{r+1} is the son of i_r for $r = 1, 2, \cdots, (m-1)$, then we say that

Paper approved by the Editor for Computer Communication of the IEEE Communications Society for publication without oral presentation. Manuscript received March 27, 1980; revised October 27, 1980. The work performed by A. Segall was performed on a consulting agreement with the Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA, and was supported in part by the Advanced Research Project Agency, U.S. Department of Defense (monitored by ONR) under Contract N00015-74-C-1183, and by the Office of Naval Research under Contract ONR/N00014-77-C-0532.

The authors are with the Department of Electrical Engineering, Technion—Israel Institute of Technology, Haifa, Israel.

i_1 is upstream from i_m, and i_m is downstream from i_1. The basic protocol summarized here assures that the flow of data from all nodes to the destination is loop-free, meaning that there are no two nodes in the network that are each upstream from each other.

Basically, each cycle of the basic protocol consists of two phases: 1) control messages propagate upstream from each destination, while updating the incremental delay coefficients $\{\lambda_i\}$ and the blocking status of each node in the network; 2) control messages propagate downstream towards the destination while performing routing changes. Control messages exchanged between neighbors (for each destination) contain the incremental delay coefficient λ and the blocking status of the sender.

The concept of blocking was first introduced in [1] and was designed to prevent formation of loops: although nodes with high incremental delays λ should normally send traffic to nodes with low incremental delays, it may happen that because of the step size η of the algorithm, in certain instances the opposite is true, namely, node k is a son of node i, but $\lambda_i(j) \leqslant \lambda_k(j)$ [2, Eq. (8a)] [here $\lambda_i(j)$ is the incremental delay at node i for destination j] and there is danger of generating a loop in the next cycle. Consequently, if, because of the constraints on the step size η, node i is not sure that it can reroute all the flow on link (i, k) in one step, i.e., if $\eta[\lambda_k(j) + D_{ik}' - \lambda_i(j)] < f_{ik}(j)$ [2, Eq. (8b)], then i declares itself blocked and so do all nodes upstream from it [in the above equation, $f_{ik}(j)$ is the flow from i to k destined for j and $D_{ik}' = (dD_{ik}/df_{ik})$, where D_{ik} is the average delay per unit time of traffic from i to k]. The protocol requires that if node k is blocked and was not a son of node i during the previous cycle, then it is not allowed to become its son during the current cycle. The proof that blocking prevents loops appears in [2].

An arbitrary node i participates in phase 1) of the protocol when it receives control messages from all its sons. For the purpose of this paper we shall say that the node goes then from an idle state $S1$ to a waiting state $S2$. At that time, it updates its incremental delay coefficient λ and its blocking status [2, Eq. (8)], and sends the updated quantities to all neighbors except for its sons. The node will then wait in state $S2$ until it receives control messages from all neighbors. At this time it performs its part of phase 2) of the protocol by sending control messages to all sons, performing routing changes [2, Eqs. (11), (12)] and going back to the idle state $S1$ in order to wait for the next cycle. The change in routing is performed at node i by choosing a specific neighbor as a "preferred son" $k_0{}^n(i, j)$ for this cycle (for destination j), increasing flow on the link to this neighbor and decreasing flow on links to all other sons. The "preferred son" $k_0{}^n(i, j)$ is chosen as the node that minimizes the incremental delay $[\lambda_m + D_{im}']$ among all neighbors m except those that are both blocked and nonsons of i [2, Eqs. (9), (13)].

The destination node triggers each cycle by sending control messages (containing $\lambda = 0$) to all neighbors while going from state $S1$ to $S2$. A new cycle might be triggered immediately after the completion of the previous cycle or at any time afterwards. A cycle is completed when the destination node receives control messages from all its neighbors (at this time, the destination goes back to state $S1$).

B. Extended Protocol to Handle Topological Changes

The extended protocol assumes the existence of a local protocol that allows the nodes at both ends of a link to sense its failure or recovery in finite time after the occurence of the change, but not necessarily at the same time at both ends. The assumption is that a link that fails cannot come up before both ends sense the failure. Also, failure or recovery of a node can be considered as failure or recovery of all adjacent links, and therefore needs no special attention.

The first extension to the basic protocol is to number consecutive cycles (corresponding to a given destination) with nondecreasing numbers. The cycle number is determined by the destination and is carried by all control messages belonging to the cycle (this in addition to the incremental delay coefficient λ_i and the blocking status).

In the extended protocol, a node i participates only in the cycle with the highest number currently known to it. This number is denoted by mx_i. Except for messages indicating path disconnection caused by failures, all messages with cycle numbers strictly lower than mx_i are disregarded. A node i participates in phase 1) of a cycle after receiving control messages with cycle number mx_i from all its current sons (12).[1] Similarly, after receiving control messages with mx_i from all its current neighbors (40), the node performs its part of phase 2). Whenever a cycle is properly completed, as in the basic protocol, the destination can start a new cycle with the same number as the previous cycle. However, a topological change may interfere with the normal evolution and proper completion of a cycle and therefore a cycle with a higher number will have to be started with the purpose of propagating the new situation throughout the network. As such, when a failure or recovery happen, the destination will have to be informed (in a distributed way) not to wait for the completion of the current cycle and to immediately start a cycle with a higher number (if such a cycle has not been started already). In addition, when a failure on a link carrying flow occurs, several other actions will have to be taken in the network: the node immediately upstream from the failure has to redistribute traffic and to realize that it should not wait for control messages on this link; if the node immediately upstream has no other sons, this node, and possibly other nodes upstream, have to take into consideration that they lose all their current paths to the destination. Although the described situations look similar to the ones appearing in [3], the fact that each node maintains several paths gives a new dimension to the problem and, in the following paragraphs, we describe the actions taken by nodes in each of these situations, mainly emphasizing our approach to the solution of the new problems.

As in [3], the protocol for notifying the destination about the occurence of a topological change is implemented by *re-*

[1] This refers to the corresponding line in the Appendix.

quest messages REQ generated by nodes adjacent to the change, carrying the number of the last cycle handled by this node (3, 6) and forwarded towards the destination.[2] The new problem arising here is to which son should the REQ message be sent by a node generating or receiving such a message. It turns out that it cannot be sent arbitrarily because then it may loop around and in fact, never arrive. The algorithm proposed by us, and that can be proven to have the desired properties (see Property 3 in Section III), is the following.

1) A node i that has a "preferred son" $k_0{}^n(i, j)$ as defined in Section II-A, forwards REQ to $k_0{}^n(i,j)$ (11a).

2) A node i for which the link to $k_0{}^n(i, j)$ has previously failed, but has links to other sons, sends REQ to any one of these sons (20, (11a).

3) A node i without sons (because of previous failures) discards the REQ message (11a).

Next we discuss the handling of link failures in the network. Here we have to distinguish between three typical cases: 1) failure of a link to a neighbor that is not a son, 2) node l is the only son of i and either the connecting link (i, l) fails, or i is informed that l has lost all its known paths to the destination, 3) same as case 2) except that l is not the only son of i. In all cases, when a failure is detected on an adjacent link, the corresponding node is deleted from the list of neighbors and, if appropriate, from the list of sons. In case 1) the needed actions are similar to the ones in [3] and of secondary importance anyway so that no detailed description will be given here. The main actions taken in case 2) are also similar to the ones in [3]: entering a "waiting for recovery state" $S3$, setting the estimated incremental delay coefficient λ_i to ∞ and sending control messages containing $\lambda_i = \infty$ to all neighbors except l, thereby propagating the appropriate information to all nodes that have lost their only path to the destination. However, there is a new problem arising here in connection with the procedure for recovery for these nodes, namely, reestablishing new routes to the destination, provided that such routes exist. In principle, this should be done, as in [3], whenever a node i in state $S3$ receives, from some node l, say, a control message with $\lambda \neq \infty$ and counter number strictly higher than currently known to i. The recovery consists of choosing this node as the new son and executing at the same time phase 1) of the new cycle. However, the question arises now what to do if it happens that node l is blocked. As previously described in Section II-A, in order to prevent formation of loops, the basic algorithm does not allow to choose a new son from among blocked neighbors. On the other hand, one cannot simply disregard the message received from l and wait for another message because the latter may never come. The solution we have found is to choose node l as the new son in this case, in spite of the fact that the node is blocked; we show in Section III (see Property 1), that this choice is indeed possible because we can still insure that a loop is not formed.

We finally discuss situation 3), when node i has more than one son and either the link to one of the sons l, say, fails or

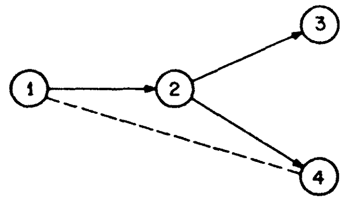

Fig. 1. Example of situation 3).

node i receives a message from son l with $\lambda_l = \infty$, thus indicating that node l has lost all its paths to the destination. In this case, node i redistributes the traffic previously routed through l in some arbitrary way as discussed later. A careful screening of the algorithm shows us, however, that in some situations further actions are required in order to prevent occurence of loops and deadlocks. To illustrate the problem, let us consider Fig. 1 where arrows indicate flow of data (to some destination j) and the dashed line indicates a link that carries no flow to this destination. With the notations used in Section II-A, suppose that $\lambda_1(j) = 8, \lambda_2(j) = 5, \lambda_3(j) = 3, \lambda_4(j) = 12, \eta = 0.01, D_{24}' = 5$, and $f_{24}(j) = 0.1$. Observe that $\lambda_2(j) < \lambda_4(j)$, but $\eta[\lambda_4(j) + D_{24}' - \lambda_2(j)] > f_{24}(j)$, and, as described in Section II-A, this implies that node 2 and consequently node 1 do not declare themselves blocked. The reason is that node 2 knows that while performing phase 2) of the cycle, it will reroute all traffic from (2, 4) to (2, 3). Now, suppose link (2, 3) fails before the rerouting occurs, i.e., while node 2 is still in state $S2$. Then the previously mentioned rerouting cannot take place, but when node 4 will perform the transition from $S2$ to $S1$ it may open a new line on link (4, 1) because $\lambda_4(j) > \lambda_1(j)$ (see also [2, Eqs. (9)-(12)]), thereby forming a loop. This situation must be prevented and our solution is to define a new state $\tilde{S2}$, entered by a node in $S2$ whenever it senses the failure of a link to a son that is not its only son or whenever it receives a message with $\lambda = \infty$ on such a link (63). In state $\tilde{S2}$, the node freezes all its operations until it hears about a new cycle.

Clearly, in addition to the above operations, each node that has lost one of its sons should stop the flow to that son, redistribute it among its remaining sons, if it still has any, and modify its routing variables correspondingly. Observe that the redistribution is arbitrary, for the purpose of this paper, since later cycles will improve the routing until the new optimum is reached. The question of how to redistribute the traffic in order to ensure fast convergence is still an open question. Another open question is the disposal of the flow if the node is left with no route to the destination after the failure. In this case, we may assume that it stores the flow until it establishes a new route or, alternatively, it rejects the flow, in which case there must exist a procedure that allows the originating node to reroute the flow.

The procedures for adding links that become operational and for initialization of the protocol are similar to [3]. In short, the nodes at the ends of a link that is ready to be added to the network have to coordinate their operations for bringing the link up. The coordination is achieved by having both nodes bring the link up as soon as they start to perform their part of the same new cycle. The initialization is performed by

[2] Recall that the protocol is described for a given destination, and has to be repeated independently for all destinations.

requiring that a new node i starts its operation in state $S3$ with node counter number equals to zero and empty lists of sons and neighbors. From this initial condition, a local protocol tries to bring the links up and the node proceeds as indicated in the extended protocol. The destination node comes into operation in state $S1$ with counter number zero and an empty list of neighbors, and then it proceeds as previously described.

III. PROPERTIES OF THE PROTOCOL

In this section the important properties of the protocol are stated explicitly. To save space, formal proofs are not given here, and the interested reader can find them in [4]. To proceed, we need the following definition. At a given instant t, the routing graph $RG(t)$ is defined as the directed graph whose nodes are the network nodes and there is an arc from node i to node l in $RG(t)$ if and only if l is a son of i at time t.

In the following, we use $n_i(t)$ and $s_i(t)$ to denote the node counter number (the number of the cycle handled by i) and the state of node i at time t, respectively. We also use the notation $p_i(t)$ for the "preferred son" $k_0{}^n(i, j)$ of node i at time t.

Property 1 (Loop Freedom)

At any time, $RG(t)$ is an acyclic graph (contains no loops) with the following properties: 1) only the destination and nodes in $S3$ have no sons; 2) if l is a son of i at time t, then $n_l(t) \geq n_i(t)$; 3) if l is a son of i at time t, and $n_l(t) = n_i(t)$, then $s_l(t) \geq s_i(t)$ (by definition $S3 > S2 = \tilde{S2} > S1$ and it is agreed that $Sx \geq Sy$ means that $Sx > Sy$ or $Sx = Sy$); 4) if $p_i(t) \neq$ NIL and $n_{p_i}(t) = n_i(t)$ and $s_{p_i}(t) = s_i(t) = S1$, then $\lambda_{p_i}(t) < \lambda_i(t)$.

To see why the loop-freedom property is guaranteed, notice that a loop might be generated only when a new son is chosen, i.e., when transition from $S2$ to $S1$ or from $S3$ to $S2$ happens. By induction, assume that the properties 2)–4) indicated above hold up to the time of the transition. The proof that a transition from $S2$ to $S1$ does not close a loop is similar to the proof of [2, Theorem 2] and will not be repeated here. Consider now the case of a transition from $S3$ to $S2$, namely, when a node i without sons chooses a neighbor l as its son. Observe that by 2) above, at the time just before the transition, the counter number of all nodes k that are upstream from i are not greater than n_i. However, node i can perform this transition only if it received from l a message with cycle number strictly higher than n_i, which implies $n_l > n_i$. Therefore, l cannot be upstream from i and hence choosing l as the new son cannot close a loop. Observe that this argument holds whether node l is blocked or not. In order to complete the proof, we only have to show that the indicated properties continue to hold after the transition. This will be omitted here and the interested reader is referred to [4].

Next, we indicate the recovery properties of the protocol. As in [3], we say that a link is *potentially working* if both ends see the link as capable of carrying traffic, and say that two nodes are *potentially connected* if there is a series of potentially working links connecting them.

Property 2 (Normal Operation)

If a cycle with counter number m is started by the destination, then, within finite time, this cycle will be properly completed or a topological change has occurred next to a node with counter number equal to m. Upon completion of this cycle and until such a change occurs, the set of all nodes i potentially connected to the destination remains constant and their directed graph $RG(t)$ will be rooted at the destination. In addition, for small enough step size η, the cycle will strictly decrease the average delay in the network.

Property 3 (Recovery)

If a topological change of order m occurs, a new cycle with counter number $m + 1$ will be, or has been, started. As pointed out, this property is guaranteed by the REQ messages.

Under the reasonable assumption that the average frequency of topological changes is not too high in comparison with the propagation time of the cycles, Property 2 and Property 3 guarantee recovery. The properties insure that cycles with ever increasing numbers will be triggered until a cycle is properly completed and all previous topological changes will have been taken care of. Finally, we state the optimality property.

Property 4 (Optimality) [2, Theorem 5]

If there is a time t after which no topological changes occur, and if the inputs into the network are stationary, then for small enough step size η, the average delay in the network will be brought to its minimum value over all routing assignments.

IV. DISCUSSION

The protocol indicated in this paper extends the ones of [1], [2] to the case when arbitrary topological changes happen in the network. The important features of this protocol are that it adapts to both slow load changes and topological changes, and the adaptivity to the new load pattern is smooth for nodes that are not affected by topological changes. We have presented here, explicitly, only the case of virtual line-switching networks. In a message switched network, the quantities to be controlled are the fractions of traffic routed over each outgoing link rather than the flows themselves [1] and the extension of this protocol to handle topological changes is quite similar to the one presented here. The details appear in [4].

APPENDIX

FORMAL DESCRIPTION OF THE PROTOCOL

Here we give, formally, the algorithm performed by each node i in the network to implement the protocol.

Definitions of Variables

N	number of nodes in the network
i	node under consideration
l	the lth neighbor of node i (values: $1, 2, \cdots, N$)
η	a parameter (see Property 4)
n_i	current counter number of node i (values: $0, 1, 2, \cdots$)

λ_i	estimated marginal delay from node i to the destination (values: $1, 2, \cdots, \infty$)
b_i	blocking status of node i (values: 0, 1); 0 means not blocked; 1 means blocked
p_i	preferred son of node i (values: NIL, $1, 2, \cdots, N$).

The processor at node i may receive the following types of messages related to each link (i, l):

MSG (m, λ, b, l)	updating message received by i from l: $(m = n_l, \lambda = \lambda_l, b = b_l)$
FAIL (l)	failure detected on link (i, l)
WAKE (l)	link (i, l) becomes operational, i.e., messages can be sent through it
REQ (m)	request for a new cycle: $(m = 0, 1, \cdots)$

We now continue the list of variables:

$F_i(l)$	status of link (i, l) as seen from node i (values: UP, DOWN, READY); UP means the link is operational; DOWN means the link is unoperational; READY means the link is ready to be brought up
$N_i(l)$	the number m received from neighbor l during the current cycle (values: NIL, $0, 1, \cdots$)
D_{il}'	estimated (or calculated) marginal delay on link (i, l) (values: $0, 1, 2, \cdots$)
$\lambda_i(l)$	last λ received at i from neighbor l (values: $0, 1, 2, \cdots, \infty$)
$D_i(l)$	$\lambda_i(l) + D_{il}'$ (values: $1, 2, \cdots, \infty$)
$B_i(l)$	blocking status of neighbor l as known at i (values: 0, 1); 0 means not blocked; 1 means blocked
$R_i(l)$	status of neighbor l (values: NIL, SON); SON means node l is a son of i
$Z_i(l)$	a synchronization number indicating the cycle number upon which the link (i, l) can be brought up, i.e., changed from READY status to UP status (values: $0, 1, 2, \cdots$)
mx_i	the largest number m received by node i up to the current time from all neighbors (values: $0, 1, 2, \cdots$)
f_{ik}	flow from node i to node k addressed to the destination
CT	a flag indicating the number of transitions the finite-state machine has already performed, triggered by the current message. (values: (0, 1); 0 means zero transitions; 1 means one or more transitions)
Txy	transition from state Sx to state Sy; it is performed if the appropriate condition holds, and then the steps under the corresponding action are executed.
Cx	changing the node tables while being in state Sx; condition and action have similar meanings as above.

In the formal description that follows, we will need to refer from time to time to certain sets of neighbors. To save space, we define those sets here:

$$C_i = \{k \mid R_i(k) = \text{SON or } [F_i(k) = \text{UP} \text{ and } N_i(k) = mx_i \text{ and } B_i(k) = 0]\}$$

$$A_i = \{k \mid R_i(k) = \text{SON or } [F_i(k) = \text{UP} \text{ and } B_i(k) = 0]\}.$$

Formal Algorithm (For Each Node i Except the Destination)

Note: The extent of an if-then clause is marked by semicolon (;).

Operations done by the message processor when a message is received (i.e., when the message processor at node i takes the message from the queue and starts processing it).

```
1.  For[3]  FAIL (l)
2.          $F_i(l) \leftarrow$ DOWN; $CT \leftarrow 0$; execute FSM;
3.          If $p_i \neq$ NIL, then send REQ($n_i$) to $p_i$;
4.  For     WAKE (l)
5.          If $i$ and $l$ agree to open link $(i, l)$,
              then $z_i(l) \leftarrow \max\{n_i, n_l\}$, $F_i(l) \leftarrow$ READY, $N_i(l) \leftarrow$ NIL;
6.          If $p_i \neq$ NIL, then send REQ($z_i(l)$) to $p_i$;
7.  For     MSG (m, λ, b, l)
8.          $N_i(l) \leftarrow m$; $\lambda_i(l) \leftarrow \lambda$; $D_i(l) \leftarrow \lambda + D_{il}'$; $B_i(l) \leftarrow b$; $mx_i \leftarrow \max\{m, mx_i\}$;
9.          If $F_i(l) =$ READY, then $F_i(l) \leftarrow$ UP;
10.         Execute FSM;
11. For     REQ (m)
11a.        If $p_i \neq$ NIL and $n_i \leqslant m$, then send REQ($m$) to $p_i$;
```

Finite-State Machine (FSM)

Note: The Finite-State-Machine is executed until no more transitions are possible.

STATE S1

```
12. T12 Cond:  $\forall k$ s.t. $R_i(k) =$ SON, holds $N_i(k) = mx_i$, $D_i(k) \neq \infty$ and $F_i(k) =$ UP;
13.            $CT = 0$;
```

[3] "For . . ." means the actions of the algorithm in response to receiving the message.

```
14.        Act:   $\lambda_i \leftarrow \min_{k:k\in C_i} \{D_i(k)\}$;
15.               If for any node $k$ s.t. $R_i(k) =$ SON, holds $\{B_i(k) = 1\}$ or
                      $\{\lambda_i(k) \geqslant \lambda_i$ and $\eta[D_i(k) - \lambda_i] < f_{ik}\}$,
                          then $b_i \leftarrow 1$,
                          else $b_i \leftarrow 0$;
16.               $n_i \leftarrow mx_i$;
17.               $\forall k$ s.t. $F_i(k) =$ READY and $n_i > Z_i(k)$, set $F_i(k) \leftarrow$ UP and $N_i(k) \leftarrow$ NIL;
18.               Send MSG$(n_i, \lambda_i, b_i, i)$ to all $k$ s.t. $F_i(k) =$ UP and $R_i(k) \neq$ SON;
19.               $CT \leftarrow 1$;
20.               If $p_i =$ NIL,
                      then choose any node $k$ s.t. $R_i(k) =$ SON and set $p_i \leftarrow k$;
21. T13    Cond:  $R_i(l) =$ SON;
22.               $\forall k \neq l$ s.t. $F_i(k) =$ UP, holds $R_i(k) =$ NIL;
23.               MSG$(m, \lambda = \infty, b, l)$ or FAIL$(l)$;
24.               $CT = 0$;
25.        Act:   $\lambda_i \leftarrow \infty$;
26.               If MSG, then $n_i \leftarrow m$;
27.               $\forall k$ s.t. $F_i(k) =$ READY and $n_i > Z_i(l)$, set $F_i(k) =$ UP and $N_i(k) =$ NIL;
28.               Send MSG$(n_i, \lambda_i, b_i, i)$ to all $k$ s.t. $F_i(k) =$ UP and $k \neq l$;
29.               $R_i(l) \leftarrow$ NIL;
30.               Cancel the flow to node $l$ and modify the routing variables by
                          setting $f_{il} = 0$;
31.               $CT \leftarrow 1$;
32.               If $p_i = l$, then $p_i \leftarrow$ NIL;
33. C1     Cond:  $R_i(l) =$ SON;
34.               $\exists k \neq l$ s.t. $R_i(k) =$ SON and $F_i(k) =$ UP;
35.               MSG$(m, \lambda = \infty, b, l)$ or FAIL$(l)$;
36.               $CT = 0$;
37.        Act:   $R_i(l) \leftarrow$ NIL;
38.               Reroute the flow to node $l$ while arbitrarily redistributing
                    it through the remaining sons and modify the routing
                    variables correspondingly;
39.               If $p_i = l$, then $p_i \leftarrow$ NIL;

                                        STATE S2

40. T21    Cond:  $\forall k$ s.t. $F_i(k) =$ UP, holds $N_i(k) = n_i = mx_i$;
41.               $\exists k \in A_i$ s.t. $D_i(k) \leqslant \lambda_i$;
42.               If $CT = 0$, then MSG;
43.               $\forall k$ s.t. $R_i(k) =$ SON, holds $D_i(k) \neq \infty$;
44.        Act:   Rerouting;
45.               Calculate $\alpha = \min_{k:k\in A_i} \{D_i(k)\}$;
46.               Let $k_0$ be any neighbor that achieves the minimum;
47.               If there is any node $q$ s.t. $F_i(q) =$ UP with $f_{iq} > 0$,
                    then for all neighbors $k \in A_i$ do:
48.                      $a_{ik} = D_i(k) - \alpha$,
49.                 Cancel all outgoing flows corresponding to incoming flows
                         that have been cancelled by fathers. Let $f_{ik}'$ be the
                         remaining outgoing flows,
50.                 $\Delta_{ik} = \min \{f_{ik}', \eta a_{ik}\}$,
51.                 Set the new flows ($\forall k$ s.t. $F_i(k) =$ UP)
```

$$f_{ik}^{\text{new}} = \begin{cases} 0 & k \notin A_i \\ f_{ik}' - \Delta_{ik} & k \in A_i, k \neq k_0 \\ f_{ik}' + \sum\limits_{\substack{k\in A_i \\ k\neq k_0}} \Delta_{ik} + \text{any new flow} & k = k_0; \end{cases}$$

```
52.                If $f_{ik} = 0\ \forall k$ s.t. $F_i(k) = $ UP,
                     then any new flow is routed through $k_0$;
53.                Send MSG$(n_i, \lambda_i, b_i, i)\ \forall k$ s.t. $R_i(k) = $ SON;
54.                $\forall k$ s.t. $F_i(k) = $ UP, set $R_i(k) \leftarrow$ NIL;
55.                Set $R_i(k_0) \leftarrow$ SON;
56.                $\forall k$ s.t. $f_{ik}^{\text{new}} > 0$ and $k \neq k_o$, set $R_i(k) \leftarrow$ SON;
57.                $\forall k$ s.t. $F_i(k) = $ UP, set $N_i(k) \leftarrow$ NIL;
58.                $CT \leftarrow 1$;
59.                $p_i \leftarrow k_o$;
60. T22 Cond:      $\forall k$ s.t. $R_i(k) = $ SON, holds $N_i(k) = mx_i > n_i, D_i(k) \neq \infty$ and $F_i(k) = $ UP;
61.                $CT = 0$;
62.      Act:      Same as Act in T12;
63. $T2\tilde{2}$ Cond:  Either same as Cond in $C1$ or FAIL$(l)$ s.t. $R_i(l) \neq$ SON;
64.                $CT = 0$;
65.      Act:      Same as Act in $C1$ and in addition set $CT \leftarrow 1$;
66. T23       :    Same as $T13$;
```

STATE S3

```
67. T32 Cond:      $\exists k$ s.t. $F_i(k) = $ UP, $mx_i N_i(k) > n_i, D_i(k) \neq \infty$;
68.      Act:      Let $k_o$ be a node that achieves
                       $\min_{\substack{k: F_i(k) = \text{UP} \\ N_i(k) = mx_i}} \{D_i(k)\}$;
69.                If $B_i(k_o) = 1$, then $b_i \leftarrow 1$;
70.                $R_i(k_o) \leftarrow$ SON;
71.                $n_i \leftarrow mx_i$;
72.                $\lambda_i \leftarrow D_i(k_o)$;
73.                $\forall k$ s.t. $F_i(k) = $ READY and $n_i > Z_i(k)$, set $F_i(k) \leftarrow$ UP and $N_i(k) \leftarrow$ NIL;
74.                Send MSG$(n_i, \lambda_i, b_i, i)\ \forall k$ s.t. $F_i(k) = $ UP and $R_i(k) \neq$ SON;
75.                Any new flow is routed through $k_o$;
76.                $CT \leftarrow 1$;
77.                $p_i \leftarrow k_o$;
```

STATE $S\tilde{2}$

```
78. $T\tilde{2}2$ Cond:  Same as Step 60.;
79.      Act:      Same as Act in $T12$;
80. $T\tilde{2}3$     :    Same as $T13$;
81. $C\tilde{2}$      :    Same as $C1$;
```

The operation of the destination node is the same as in [3, Table 4].

REFERENCES

[1] R. G. Gallager, "A minimum delay routing algorithm using distributed computation," *IEEE Trans. Commun.*, vol. COM-25, pp. 73–85, Jan. 1977.

[2] A. Segall, "Optimal distributed routing for line-switched data networks," *IEEE Trans. Commun.*, vol. COM-27, pp. 201–209, Jan. 1979.

[3] P. M. Merlin and A. Segall, "A failsafe distributed routing protocol," *IEEE Trans. Commun.*, Sept. 1979; Dep. Elec. Eng., Technion, Haifa, May 1978, EE Pub. 313.

[4] M. Sidi and A. Segall, "Failsafe distributed optimal routing in data communication networks," Dep. Elec. Eng., Technion, Haifa, Dec. 1978, EE Pub. 342.

Adrian Segall (S'71–M'74–SM'79), for a photograph and biography, see p. 497 of the April 1981 issue of this TRANSACTIONS.

Moshe Sidi was born in Israel on April 11, 1953. He received the B.Sc. and M.Sc. degrees from the Technion—Israel Institute of Technology, Haifa, Israel, in 1975 and 1978, respectively, both in electrical engineering. He is currently working towards the Ph.D. degree at the Technion in the area of radio packet networks.

From 1975 to 1976 he worked at the Israel Water Company as a Communication Engineer. Since 1976 he has been a Teaching Assistant at the Technion in communication and data networks courses. His research interests lie mainly in the areas related to computer communication.