The New Routing Algorithm for the ARPANET

JOHN M. McQUILLAN, MEMBER, IEEE, IRA RICHER, MEMBER, IEEE, AND ERIC C. ROSEN

Abstract—**The new ARPANET routing algorithm is an improvement over the old procedure in that it uses fewer network resources, operates on more realistic estimates of network conditions, reacts faster to important network changes, and does not suffer from long-term loops or oscillations. In the new procedure, each node in the network maintains a database describing the complete network topology and the delays on all lines, and uses the database describing the network to generate a tree representing the minimum delay paths from a given root node to every other network node. Because the traffic in the network can be quite variable, each node periodically measures the delays along its outgoing lines and forwards this information to all other nodes. The delay information propagates quickly through the network so that all nodes can update their databases and continue to route traffic in a consistent and efficient manner.**

An extensive series of tests were conducted on the ARPANET, showing that line overhead and CPU overhead are both less than two percent, most nodes learn of an update within 100 ms, and the algorithm detects congestion and routes packets around congested areas.

I. INTRODUCTION

THE last decade has seen the design, implementation, and operation of several routing algorithms for distributed networks of computers. The first such algorithm, the original routing algorithm for the ARPANET, has served remarkably well considering how long ago (in the history of packet switching) it was conceived. This paper describes the new routing algorithm we installed recently in the ARPANET. Readers not familiar with our earlier activities may consult [1] for a survey of the ARPANET design decisions, including the previous routing algorithm; readers interested in a survey of routing algorithms for other computer networks and current research in the area may consult [2].

A distributed, adaptive routing scheme typically has a number of separate components, including: 1) a *measurement process* for determining pertinent network characteristics, 2) a *protocol* for disseminating information about these characteristics, and 3) a *calculation* to determine how traffic should be routed. A routing "algorithm" or "procedure" is not specified until all these components are defined. In the present paper, we discuss these components of the new ARPANET algorithm. We begin with a brief outline of the shortcomings of the original algorithm; then, following an overview of the new procedure, we provide some greater detail on the individual components. The new algorithm has undergone extensive testing in the ARPANET under operational conditions, and the final section of the paper gives a summary of the test results. This paper is a summary of our conclusions only; for more complete descriptions of our research findings, see our internal reports on this project [3]-[5].

Paper approved by the Editor for Computer Communication of the IEEE Communications Society for publication without oral presentation. Manuscript received May 11, 1979; revised October 5, 1979. This work was supported by the Defense Advanced Research Projects Agency under ARPA Order 3941, and by the Defense Communications Agency (DoD) under Contract MDA903-78-C-0129, monitored by DSSW.

The authors are with Bolt Beranek and Newman Inc., Cambridge, MA 02138.

II. PROBLEMS WITH THE ORIGINAL ALGORITHM

The original ARPANET routing algorithm and the new version both attempt to route packets along paths of least delay. The total path is not determined in advance; rather, each node decides which line to use in forwarding the packet to the next node. In the original approach, each node maintained a table of estimated delay to each other node, and sent its table to all adjacent nodes every 128 ms. When node I received the table from adjacent node J, it would first measure the delay from itself to J. (We will shortly discuss the procedure used for measuring the delay.) Then it would compute its delay via J to all other nodes by adding to each entry in J's table its own delay to J. Once a table was received from all adjacent nodes, node I could easily determine which adjacent node would result in the shortest delay to each destination node in the network.

In recent years, we began to observe a number of problems with the original ARPANET routing algorithm [7] and came to the conclusion that a complete redesign was the only way to solve some of them. In particular, we decided that a new algorithm was necessary to solve the following problems.

1) Although the exchange of routing tables consumed only a small fraction of line bandwidth, the packets containing the tables were long, and the periodic transmission and processing of such long, high-priority packets can adversely affect the flow of network traffic. Moreover, as the ARPANET grows to 100 or more nodes, the routing packets would become correspondingly larger (or more frequent), exacerbating the problem.

2) The route calculation is performed in a distributed manner, with each node basing its calculation on local information together with calculations made at every other node. With such a scheme, it is difficult to ensure that routes used by different nodes are consistent.

3) The rate of exchange of routing tables and the distributed nature of the calculations causes a dilemma: the network is too slow in adapting to congestion and to important topology changes, yet it can respond too quickly (and, perhaps, inaccurately) to minor changes.

The delay measurement procedure of the old ARPANET routing algorithm is quite simple. Periodically, an IMP counts the number of packets queued for transmission on its lines and adds a constant to these counts; the resulting number is the "length" of the line for purposes of routing. This delay measurement procedure has three serious defects.

1) If two lines have different speeds, or different propagation delays, then the fact that the same number of packets is

0090-6778/80/0500-0711$00.75

queued for each line does not imply that packets can expect equal delays over the two lines. Even if two lines have the same speed and propagation delay, a difference in the size of the packets which are queued for each line may cause different delays on the two lines.

2) In the ARPANET, where the queues are constrained to have a (short) maximum length, queue length is a poor indicator of delay. The constraints on queue length are imposed by the software in order to fairly resolve contention for a limited amount of resources. There are a number of such resources which must be obtained before a packet can even be queued for an output line. If a packet must wait a significant amount of time to get these resources, it may experience a long delay, even though the queue for its output line is quite short.

3) An instantaneous measurement of queue length does not accurately predict average delay because there is a significant real-time fluctuation in queue lengths at any traffic level. Our measurements show that under a high constant offered load, the average delay is high, but many individual packets show low delays, and the queue length often falls to zero! This variation may be due to variation in the utilization of the CPU, or to other bottlenecks, the presence of which is not accurately reflected by measuring queue lengths.

These three defects are all reflections of a single point, namely, that the length of an output queue is only one of many factors that affect a packet's delay. A measurement procedure that takes into account only one such factor cannot give accurate results.

The new routing algorithm is an improvement over the old one in that it uses fewer network resources, operates on more realistic estimates of network conditions, reacts faster to important network changes, and does not suffer from long-term loops or oscillations.

III. OVERVIEW OF THE NEW ROUTING PROCEDURE

The routing procedure we have developed contains several basic components. Each node in the network maintains a database describing the network topology and the line delays. Using this database, each node independently calculates the best paths to all other nodes, routing outgoing packets accordingly. Because the traffic in the network can be quite variable, each node periodically measures the delays along its outgoing lines and forwards this information (as a "routing update") to all other nodes. A routing update generated by a particular node contains information only about the delays on the lines emanating from that node. Hence, an update packet is quite small (176 bits on the average), and its size is independent of the number of nodes in the network. An update generated by a particular node travels *unchanged* to all nodes in the network (not just to the immediate neighbors of the originating node, as in many other routing algorithms). Since the updates need not be processed before being forwarded because they are small, and since they are handled with the highest priority, they propagate very quickly through the network, so that all nodes can update their databases rapidly and continue to route traffic in a consistent and efficient manner.

Many algorithms have been devised for finding the shortest path through a network. Several of these are based on the concept of computing the entire tree of shortest paths from a given node, the root of the tree. A recent article [9] discusses some of these algorithms and references several survey articles. The algorithm we have implemented is based on one attributed to Dijkstra [10]; because of its search rule, we call it the shortest-path-first (SPF) algorithm.

The basic SPF algorithm uses a database describing the network to generate a tree representing the minimum delay paths from a given root node to every other network node. Fig. 1 shows a simplified flowchart of the algorithm. The database specifies which nodes are directly connected to which other nodes, and what the average delay per packet is on each network line. (Both types of data are updated dynamically, based on real-time measurements.) The tree initially consists of just the root node. The tree is then augmented to contain the node that is closest (in delay) to the root and that is adjacent to a node already on the tree. The process continues by repetition of this last step. LIST denotes a data structure containing nodes that have not yet been placed on the tree but are neighbors of nodes that are on the tree. The tree is built up shortest-paths-first—hence, the name of the algorithm. Eventually, the furthest node from the root is added to the tree, and the algorithm terminates. We have made important additions to this basic algorithm so that changes in network topology or characteristics require only an incremental calculation rather than a complete recalculation of all shortest paths.

Fig. 2 shows a six-node network and the corresponding shortest path tree for node 1. The figure also shows the routing directory which is produced by the algorithm and which would be used by node 1 to dispatch traffic. For example, traffic for node 4 is routed via node 2. Only the routing directory is used in forwarding packets; the tree is used only in creating the directory.

The two other important components of the routing procedure are the mechanism for measuring delay and the scheme for propagating information. The routing algorithm must have some way of measuring the delay of a packet at each hop. This aspect of the routing algorithm is quite crucial; an algorithm with poor delay measurement facilities will perform poorly, no matter how sophisticated its other features are.

Each node measures the actual delay of each packet flowing over each of its outgoing lines, and calculates the average delay every 10 s. If this delay is significantly different from the previous delay, it is reported to all other nodes. The choice of 10 s as the measurement period represents a significant departure from the old routing algorithm. Since it takes 10 s to produce a measurement, the delay estimate for a given line cannot change more often than once every 10 s. The old routing algorithm, on the other hand, would allow the delay estimate to change as often as once every 128 ms. We now believe, however, that there is no point in changing the estimate so often, since it is not possible to obtain an accurate estimate of delay in the ARPANET in less than several seconds. (See Section IV-B.)

The updating procedure for propagating delay information is of critical importance because it must ensure that each update is actually received at all nodes so that identical databases of routing information are maintained at all nodes. Each update is transmitted to all nodes by the simple but reliable

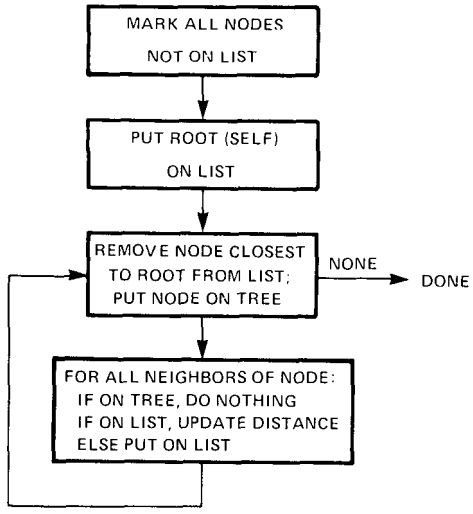
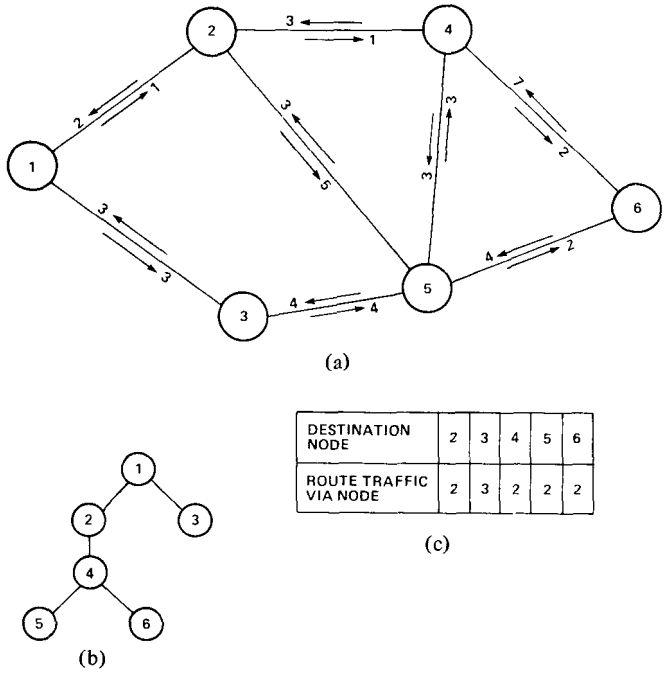


Fig. 1.



DESTINATION NODE	2	3	4	5	6
ROUTE TRAFFIC VIA NODE	2	3	2	2	2

Fig. 2. (a) Example network (line lengths indicated by the numbers beside the arrowheads). (b) Shortest path tree. (c) Routing directory.

method of transmitting it on all lines. When a node receives an update, it first checks to see if it has processed that update before. If so, the update is discarded. If not, it is immediately forwarded to all adjacent nodes. In this way, the update flows quickly (within 100 ms) to all other nodes. The fact that an update flows once in each direction over each network line is the basis for a reliable transmission procedure for the updates. Because the updates are short and are generated infrequently, this procedure uses little line or node bandwidth (less than two percent). We have augmented this basic procedure with a mechanism to ensure that databases at nodes are correctly updated when a new node or line is installed, or when a whole set of previously disconnected nodes joins the network. This is discussed in more detail in Section IV-C.

Since all nodes perform the same calculation on an identical database, there are no permanent routing loops. Of course,

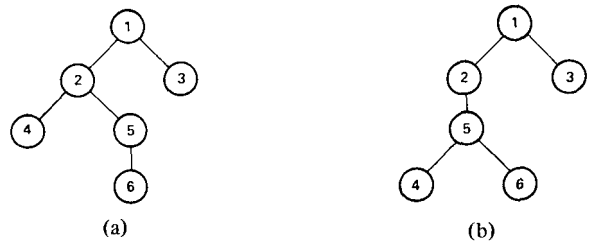


Fig. 3. (a) Shortest path tree for network of Fig. 2(a) after the length of the line 2 → 4 increase to 6. (b) Modified tree after the length of line → 5 decrease to 2.

transient loops may form for a few packets when a change is being processed, but that is quite acceptable, since it has no significant impact on the average delay in the network.

IV. DETAILED DESCRIPTION OF THE NEW ROUTING PROCEDURES

A. Routing Calculation—The SPF Algorithm

We now describe the additions to the basic algorithm of Fig. 1 which we have developed to handle various possible changes in network status without having to recalculate the whole tree. For each change described below, we assume that the shortest path tree rooted at node I prior to the change is known.

First, consider the case where the delay of the line AB from node A to node B increases. Clearly, if the line is not in the tree (i.e., not in the shortest path from that node to any other node), nothing need be done because if the line were not part of any shortest path prior to the change, then it will certainly not be used when its delay increases. If the line is in the tree, then the delay to B increases, as does the delay to each node whose route from I passes through B. Thus, the nodes in the subtree whose root is B are candidates for changed positions in the tree. Conversely, nodes not in this subtree will not be repositioned.

The first two steps for handling an increase of X in the delay from A to B are as follows.

1) Identify nodes in B's subtree and increase their delays from I by X.

2) For each subtree node S, examine S's neighbors which are not in the subtree to see if there is a shorter path from I to S via those neighbors. If such a path is found, put node S on LIST.

At the conclusion of these steps, LIST either will be empty or will contain some subtree nodes for which better paths have been found. In order to find the best paths to the nodes on LIST, a slightly modified version of SPF can be invoked. This will also find better paths, if any exist, for other subtree nodes. Fig. 3(a) shows the modification to the tree of Fig. 2 that results when the delay of the line from node 2 to node 4 increases to 6.

Now consider the case where the delay on AB decreases by X. If this line is in the tree, then paths to the nodes of the subtree which have B as its root will be unchanged because the subtree nodes were already at minimum delay, and hence the decreased delay will only shorten their distances from I. Moreover, any node whose delay from I is less than or

equal to B's new distance from I will not be repositioned, since the node's path must reach B first in order to take advantage of the improved line. However, nodes which are not in the subtree and which are farther from I than B may have a shorter distance via one of the subtree nodes.

The algorithm must thus first perform the following steps.

1) Identify the nodes in the subtree and decrease their distances from I by X.

2) Try to find a shorter distance for each node K that is not in the subtree but is adjacent to a subtree node by identifying a path to K via an adjacent node which is in the subtree. If such a path is found, put node K on LIST.

At the conclusion of these steps, LIST will contain some (possibly zero) subtree adjacent nodes that have been repositioned. Nodes adjacent to these that are not in the subtree are also candidates for improved paths, and starting with the LIST generated in step 2) above, the basic SPF algorithm (with minor modifications) can be used to restructure the rest of the tree. Fig. 3(b) shows how the tree of Fig. 3(a) changes when the length of the link from node 2 to node 5 decreases to 2, while the length of the link from node 2 to 4 remains at 6.

If the delay on line AB improved, but AB was *not* originally in the shortest path tree, the algorithm first determines whether B can take advantage of this improvement. Since the delay from I to A cannot be improved, the delay to B using the line AB will be equal to the original distance to A plus the new delay of AB. If the new delay is greater than or equal to the former delay from I to node B, then the improved line does not help and no changes are made to the tree or to the routing table. If, on the other hand, the updated delay is less than the original delay, then the best route to B now includes AB. The first change to the shortest path tree is, therefore, to relocate B (and its subtree), attaching it to node A via line AB. Now the situation is identical to that of the previous paragraph in which the line from A to B was in the tree in the first place and its delay decreased.

Finally, a change in the status of a node—namely, the addition of a new node, the removal of a node, a node failure, or its recovery from a failure—is implicitly recognized by the change in the status of its lines. For example, if a node fails, its neighbors determine that the lines to that node have failed, and when other nodes receive this information, they calculate that the failed node is unreachable. (Of course, nodes can become unreachable even if their lines do not fail.) Thus, the algorithm need explicitly consider only line changes.

The basic SPF calculation and all of the above incremental cases are consolidated into the semiformal version of the algorithm given in the Appendix.

B. Delay Measurement

Measuring the delay of an individual packet is a simple matter. When the packet arrives at the IMP, it is time-stamped with its arrival time. When the first bit of the packet is transmitted to the next IMP, the packet is stamped with its "sent time." If the packet is retransmitted, the original sent time is overwritten with the new sent time. When the acknowledgment for the packet is received, the arrival time is subtracted from the sent time. To this difference are added the propagation delay of the line (a constant for each line) and the packet's transmission delay (found by looking it up in a table indexed by packet length and line speed). The result is the packet's total delay at that hop—the time it took the packet to get from one IMP to the next.

Every 10 s the average delay of all packets which have traversed a line in the previous 10 s is computed. Our measurements show that when we take an average over a period of less than 10 s, the average shows too much variation from measurement period to measurement period, even when the offered load is constant. There is a tradeoff here: a longer measurement period means less adaptive routing if conditions actually change; a shorter period means less optimal routing because of inaccurate measurements.

Another important aspect of the measurement technique is that the measurement periods are *not* synchronized across the network. Rather, the measurement periods in the different IMP's are randomly phased. This is an important property because synchronized measurement periods could, in theory, lead to instabilities [4], [11].

The new routing algorithm does not necessarily generate and transmit an update at the end of each measurement period; it does so only if the average delay just measured is "significantly" different from the average delay reported in the last update that was sent (which may or may not be the same as the delay measured in the previous measurement period). The delay is considered to have changed "by a significant amount" whenever the absolute value of the change exceeds a certain threshold. The threshold is not a constant but is a decreasing function of time because whenever there is a large change in delay, it is desirable to report the new delay as soon as possible, so that routing can adapt quickly; but when the delay changes by only a small amount, it is not important to report it quickly, since it is not likely to result in important routing changes. However, whenever a change in delay is long lasting, it is important that it be reported eventually, even if it is small; otherwise, additive effects can introduce large inaccuracies into routing. What is needed, then, is a scheme which reacts to large changes quickly and small changes slowly. A threshold value which is initially high but which decreases to zero over a period of time has this effect. In the scheme we have implemented, the threshold is initially set to 64 ms. After each measurement period, the newly measured average delay is compared with the previously reported delay. If the difference does not exceed the threshold, the threshold is decreased by 12.8 ms. Whenever a change in average delay equals or exceeds the threshold, an update is generated, and the threshold is reset to 64 ms. Since the threshold will eventually decay to zero, an update will always be sent after a minute, even if there is no change in delay. (This feature is needed to ensure reliability of the updating protocol under certain conditions. See Section IV-C.) It should be pointed out that when a line goes down or comes up, an update reporting that fact is generated immediately.

C. Updating Policy

We next discuss the policy for propagating the delay information needed in SPF calculations, which require identical data bases at all the nodes [12]. The updating technique must

meet two basic criteria, high efficiency (i.e., low utilization of line and CPU bandwidth) and high reliability. Efficiency is important both under normal conditions and when a change is detected that requires immediate updating. Reliability means that updates must be processed in sequence, handled without loss during equipment failures, and treated correctly after failure recovery.

Rather than having separate updates for each line, each update contains information about all the lines at a particular IMP. That is, each update from a given node specifies all the neighbors of that node, as well as the delay on the direct line to each of the neighbors. This results in more efficiency (i.e., less overhead), and the simplicity of only one single serial number per node. The latter makes sequencing and other bookkeeping easier.

We considered different approaches for distributing the updates [8] and decided on "flooding," in which each node sends each new update it receives on all its lines except the line on which the update was received. An important advantage of flooding is that the node sends the same message on all its lines, as opposed to creating separate messages on the different lines. These messages are short (no addressing information is required), so that the total overhead due to routing updates is much less than one percent. A final consideration which favors flooding is that it is independent of the routing algorithm. This makes it a safe, reliable scheme.

We considered several different ways of augmenting the basic flooding scheme to ensure reliable transmission [4]. An important feature of all the schemes is that updates which need to be retransmitted can be reconstructed from the topology tables in each IMP. The protocol we have adopted uses an explicit acknowledgment which is a natural extension of the basic flooding scheme. Using flooding, there is no need to transmit an update back over the line on which it was received since the neighbor on that line already has the update. In our protocol, however, the updates are transmitted over *all* lines, including the input line. The "echo" over the input line serves as an acknowledgment to the sender; if the echo is not received in a given amount of time (measured by a retransmission timer for each line), the update is retransmitted. In order to cover the case of a missed echo, the retransmitted update is specially marked (with a "Retry" bit) to force an echo even if the update has been seen before. Note that acknowledging an update at each hop ensures that the update will be received by all nodes which have a path to the source.

One difficult problem in maintaining duplicate databases at all nodes is that some nodes may become disconnected from each other due to a network partition. For some period of time, certain nodes are unable to receive routing updates from certain other nodes. When the partition ends, the nodes in one segment of the network may remember the serial numbers of the last updates they received from nodes in the other segment. However, if the partition lasted a long enough time, the serial numbers used by the disconnected nodes may have *wrapped around* one or more times. If there has been wrap-around, it is meaningless to compare the serial numbers of new updates with the serial numbers of old updates. Some method must be developed to force all nodes to discard the prepartition updates in favor of the postpartition ones. The obvious approach of ignoring updates from unreachable nodes is not workable, since the SPF databases may temporarily be inconsistent, and different nodes may ignore different updates.

This problem is resolved by having the update packets carry around some indication of their age. There is a k-bit field in each packet, and each node has a clock which ticks once every t seconds. When an update is first generated, the "age field" is $2k - 1$. When an update is received, its age field is decremented once each tick of the clock. An update is considered "too old" when its age field has been decremented to zero. This scheme ensures that the age of an update as seen by a given node is determined by the time it has been held in the given node, plus the time it was held in any nodes from which it was retransmitted. The use of a time-out scheme like the one just described places several constraints on the parameters used by the routing scheme.

1) It should be impossible for the serial numbers of updates generated by any one node to wrap around (i.e., to get halfway through the sequence number space) before the time-out period expires.

2) The time-out period should be somewhat longer than the maximum period between updates from a single node. This means that good, recent updates from reachable nodes will not time out.

3) It should be impossible for a node to stop and be restarted within the time-out interval. This ensures that all of the node's old updates will time out before any new updates are sent.

There is one other important facet to the updating protocol. When a network line which has been down is determined to be in good operating condition, it is placed in a special "waiting" state for a period of one minute. The line is not "officially" considered to be up until the waiting period is over. While a line is in the waiting state, therefore, no data can be routed over it. However, routing updates *are* transmitted over lines in the waiting state. As we indicated in Section IV-B, each node is required to generate at least one update per minute, even if there is no change in delay. This means that while a line is in the waiting state, an update from every node in the network will traverse it; the line cannot come up until enough time has elapsed so that recent updates from all nodes have been transmitted over it. This feature is needed for three reasons.

1) In order to properly perform the routing computation, a node must have a copy of the network database which is identical to the copies in all the other nodes. Recall that the database specifies the topology of the network (i.e., which nodes are direct neighbors of which other nodes), as well as the delay on each network line. When a new node is ready to join the network, it has none of this information. It must somehow obtain the information before it can be permitted to join. Note, however, that the procedure described above ensures that a node cannot come up (because its lines cannot come up) until it has received an update from each other node. Since an update from a given node specifies the neighbors of that node, as well as the delay on the line to each neighbor, it follows that a node cannot come up until it has received enough information to construct a complete and up-to-date copy of the network database.

2) When the network is partitioned, the partition must not

be permitted to end until updates from each segment have flowed into the other. Otherwise, nodes in one segment may have copies of the database which are inconsistent with those in the other segment. Again, the procedure of having each node generate at least one update per minute, while holding a line in the waiting state before allowing it up, is sufficient to avoid this problem. Since a partition can only end when a line comes up, and a line cannot come up until updates from all nodes have traversed it, a partition cannot end until all nodes have complete, consistent, and up-to-date copies of the database.

3) There are certain peculiar cases in which flooding is not totally reliable, even when augmented by a retransmission strategy. For example, suppose a node has two lines, one of which comes up at the precise moment the other goes down. An update which is being flooded around the network might arrive at each line at a time when it is down. This means that the update may never reach that node, even though there is no instant when both of the node's lines are down. However, by ensuring that a line cannot come up until enough time has passed for updates from all nodes to have traversed it, this situation is prevented.

V. PERFORMANCE

We next describe some analytical and empirical results on the performance of the new routing algorithm. One important measure of the efficiency of the SPF algorithm is the average time required to process changes in the delays along network lines, since such changes comprise the bulk of the processing requirements. When a given node receives an update message indicating that the delay along some line has increased, the running time of the SPF algorithm is roughly proportional to the number of nodes in that line's subtree; that is, it is roughly proportional to the number of nodes to which the delay has become worse. When a given node receives an update message indicating that the delay along some line has decreased, the amount of time it takes to run the incremental SPF algorithm is roughly proportional to the number of nodes in that line's subtree *after* the algorithm is run; that is, it is roughly proportional to the number of nodes to which the delay got better. Thus, in either case, the SPF running time is directly related to the subtree size.

Since the average subtree size provides a measure of SPF performance, it is useful to understand how this quantity varies with the size of the network. Let N denote the number of network nodes, and let hi represent the number of hops on the path from the source node, $i = 1$, to node i; in other words, if the length of each line is 1, then hi is the length of the path to node i. Clearly, node i appears in i's subtree and in the subtrees of all the nodes along the path to i. Thus, hi is equal to the number of subtrees in which node i is present, so that

$$\text{total number of all subtree nodes} = \sum_{i=2}^{N} h_i$$

and since there are $N - 1$ subtrees (the complete tree from the source node is not considered to be a "subtree"), the average subtree size is given by

$$\text{average subtree size} = \frac{1}{N-1} \sum_{i=2}^{N} h_i.$$

But this expression is identical to the average hop length of all paths, and thus we have the remarkable result that in any tree, the average subtree size is equal to the average hop length from the root to all nodes. This result is significant because the average hop length generally increases quite slowly as the number of nodes increases. (For a network with uniform connectivity $c > 2$, the average hop length increases roughly as $\log N/\log (c - 1)$ [3].)

To establish some estimate of the running time of the algorithm, we programmed a stand-alone version for the ARPANET nodes. We randomly assigned each line in the ARPANET a length between 1 and 20. We ran the SPF algorithm to initialize the data structure in each node. Then we picked 50 lines at random and successively gave each a new random length. Every time we changed the length of a line, we changed it by at least 15 percent. Also, some lines were brought down by being assigned a length which represented infinity. Each time we did this, we ran the SPF algorithm with each node as the source node. We obtained the following results.

1) The average time per node to run the incremental SPF algorithm was about 2.2 ms.

2) The average time per subtree node to run the incremental SPF algorithm was about 1.1 ms.

Since we calculated that the average subtree size multiplied by the probability that a line is in the tree is about 2, these two results are in agreement. Note that these are average times; actual times varied from under 1 to 40 ms.

The figures given above are for the shortest path calculation only. Processing an update invokes a routine to maintain the topology database (including the ability to dynamically add or delete lines and nodes), and a routine to determine which nodes can be reached from the root node. These modules increase the running time by about a factor of two; and the total storage requirement, including these modules, the topology database, and the measurement and updating packages, is about 2000 16-bit words.

We designed and programmed the new routing procedure over a period of about six months. We than began an extensive series of tests on the ARPANET, at off-peak hours but under actual network conditions [5]. Our tests involved a great deal more than simply turning the new routing algorithm on to see whether it would run. The tests were specifically designed to stress the algorithm, by inducing those situations which would be most difficult for it to handle well. To stress its ability to react properly to topological changes, we induced line and node failures in as many different ways as we could think of, including multiple simultaneous failures. We also generated large amounts of test traffic in order to see how the algorithm performs under heavy load. (In this respect, it should be noted that the periods during which we were testing were "off-peak"

only with respect to the amount of ordinary user traffic in the network. The amount of test traffic we generated far exceeds the amount of traffic generated by users, even during peak hours.) We experimented with many different traffic patterns, in order to test the algorithm under a wide variety of heavily loaded conditions. In particular, we tried to induce those situations which would be most likely to result in loops or in wild oscillations. We also designed and implemented a sophisticated set of measurement and instrumentation tools, so that we could evaluate the routing algorithm's performance. Some of these tools enabled us to monitor the utilization of resources used by the algorithm. Others enabled us to monitor changes in delay (as measured by the routing algorithm), as well as changes in the routing trees themselves at particular network nodes. One of our most important tools was the "tagged packet." A tagged packet is a packet which, as it travels through the network, receives an imprint from each node through which it travels. When such a packet reaches its destination, it contains a list of all the nodes it has traversed, as well as the delay it experienced at each node. These packets provided us with a very straightforward indication of the routing algorithm's performance. Of course, since the network was also in use by ordinary users during our tests, we cannot claim to have performed "controlled" experiments, in the strict scientific sense. However, all our experiments were repeated many times before being used to draw conclusions. Some of our main results are as follows.

1) Utilization of resources (line and processor bandwidth) by the new routing algorithm is as expected, and compares quite favorably with the old algorithm. Line overhead is less than one percent; CPU overhead is less than two percent. We have measured these quantities repeatedly since the new routing algorithm became operational in May 1979, and we have found this result to hold even during peak hours on the network.

2) The new algorithm responds quickly and correctly to topological changes; most nodes adapt to the change within 100 ms.

3) The new algorithm is capable of detecting congestion, and will route packets around a congested area.

4) The new algorithm tends to route traffic on minimum hop paths, unless there are special circumstances which make other paths more attractive.

5) The new algorithm does *not* show evidence of serious instability or oscillations due to feedback effects.

6) Routing loops occur only as transients, affecting only packets which are already in transit at the time when there is a routing change. The few packets that we have observed looping have not traversed any node more than twice. However, the loop can be many hops long. Although packets which loop may experience a longer delay than packets which do not, there is no significant impact on the average delay in the network.

7) Under heavy load, the new algorithm will seek out paths where there is excess bandwidth, in order to try to deliver as much traffic as possible to the destination.

Of course, the new routing algorithm does not generate optimal routing—no single-path algorithm with statistical input data could do that. It has performed well, and is successful in eliminating many of the problems associated with the old routing scheme. After several months of careful testing during which both old and new routing algorithms were resident in the network and used for experiments [5], we began to operate the ARPANET with the new routing scheme in May 1979, and removed the old routing program. Since that time, we have continued to monitor the performance of the algorithm. The results obtained during our test periods have continued to hold, even during peak hours, and no new or unforeseen problems have yet arisen.

Is the new routing algorithm really better than the old? We are convinced that it is for reasons that we will summarize shortly. We would like first to point out, though, that there is no general answer to the question, "What makes routing algorithm A better than routing algorithm B?" If one could claim that algorithm A performs better in every possible situation than algorithm B does, according to some well-defined metric of performance, then one would have a good reason for preferring A. However, such a claim could never be supported for it is untestable. One might try to claim that algorithm A performs better than algorithm B in "most" situations, but that would not necessarily show that A is a better algorithm than B. A's performance in the minority of situations might be so much worse than B's that B is to be preferred. Furthermore, it is difficult to define a performance metric which is equally applicable to every possible situation. For example, in some situations it may be desirable to minimize delay; in others to maximize throughput. Yet these two intuitively desirable performance metrics are in conflict. In attempting to decide which of two routing algorithms is the better one, one cannot appeal to any procedure simple enough to be followed by rote. Rather, one must first look at particular situations which are known to give rise to performance difficulties. Then one must decide what sort of performance one would like to see in those situations (a decision often akin to a value judgment). Only then can one compare the two algorithms to see which gives the more desirable performance.

Our purpose in designing and implementing a new routing algorithm for the ARPANET was to eliminate certain problems in the performance of the old algorithm, while at the same time maintaining the strengths of the old algorithm. We believe that one of the strengths of the old algorithm was that it was distributed, in the sense that the routing computation was performed by every node. In the ARPANET environment, this makes good sense from the point of view of reliability and efficiency. The new routing algorithm retains this feature by replicating the SPF computation at every node. There is a sense, however, in which the old routing computation was a distributed computation but the SPF computation is not. In the old algorithm, the inputs to the computation at one node were the outputs of the computation at the neighboring nodes. In this sense, then, the old routing computation was a global computation, with each node performing just a piece of it. Since the nodes performed the computation in an unsynchronized manner, the output of the global computation at any instant depended more on the history of events around the network than on the traffic in the network at that instant. The SPF computation, on the other hand, is a local computation. It does depend on measurements which have been made all

around the network, but the updating protocol provides these measurements to all nodes unchanged and unprocessed; the SPF computation at one node never learns of the results of the SPF computation at any other node. In this way, we have kept the advantages of distributed routing while dispensing with the disadvantages of having a distributed computation.

Another good quality of the old algorithm was its attempt to adapt to changing delay conditions in the network. We realize that there may be certain applications, where network traffic can be accurately predicted and the network can be sized to handle exactly that traffic load and pattern, in which it may not be important for the routing to be adaptive. In the ARPANET, however, nodes and trunks are frequently added or removed. These changes are primarily made for administrative or economic reasons, rather than for the purpose of optimizing traffic flows. The traffic in the ARPANET is quite unpredictable, being largely determined by the behavior of a community of researchers. Furthermore, although there are sites on the ARPANET separated by as many as 11 hops, about one-third of the messages in the network travel no more than one hop; about half travel no more than three hops. This leads to situations in which the load in the network is very nonuniform, and these are the situations in which adaptive routing is likely to be of great utility. For reasons such as these, adaptive routing seems no less important to us now than it did to the original designers of the ARPANET many years ago. There were, however, several problems in the way the old algorithm responded to changes in network delay. Most of the problems stemmed from deficiencies in the delay measurement procedure of the old algorithm (see Section II). Because of these deficiencies, the old algorithm was often incapable of detecting congestion, and would sometimes send traffic directly into a congested area, thereby causing the congestion to spread. Our tests [5] show that we have eliminated this problem, and have done so without introducing any countervailing problems, such as instability or wild oscillation of the routing patterns. In its ability to adjust to changes in delay, the new algorithm appears to dominate the old completely.

An important deficiency of the old algorithm was its slow response to topological changes. The old algorithm would take many seconds to respond properly to a node or line failure. During this period, many nodes could be directing traffic towards a failed node. Having to buffer such traffic for seconds at a time was a significant cause of congestion in the network. With the new algorithm, however, the time for all nodes to respond to topological changes is on the order of 100 ms. Since the new algorithm was installed, we have not observed any congestion arising due to slow response to line or node failures.

The updates of the old routing algorithms were over 1200 bits long. There were often as many as seven such updates per second on each line. The new routing updates average 176 bits. Even during peak periods, it is rare to see more than two updates per second per line. One of the problems with the old algorithm was the increase in the delay of ordinary data packets due to the presence of the long, frequently sent routing updates. Clearly, the new routing updates interfere much less with ordinary network traffic than did the old.

The old routing algorithm took a fixed amount of time (15-20 ms) to process an update. The new algorithm, as we have pointed out, takes a variable amount of time, with the amount of time proportional to the size of the routing change necessitated by the update. This results in a much more efficient use of the CPU.

One of the major problems of the old algorithm was that it was prone to form loops which might persist for several seconds at a time. A given packet might be trapped in such a loop for a significant amount of time. Often a large number of packets would get "sucked in" to such loops, causing congestion which began at the locus of the loop and then spread throughout the network. While the new algorithm cannot be said to be loop-free, the loops that it forms occur only as transients while the network is adapting to a routing change. The loops which do form do not persist; a packet will sometimes loop once, but we have never seen packets traveling around and around in a loop, as would sometimes happen with the old algorithm. The small amount of looping which has been observed has *never* led to congestion, or even to a significant increase in average network delay. We conclude, therefore, that looping is not a problem with the new algorithm, as it was with the old.

Someone might object that any algorithm that permits loops is seriously deficient; this point is worth commenting on briefly. It is certainly true that, other things being equal, it is better not to have looping than to have it. But other things are never equal—we know of no pair of routing algorithms that perform exactly alike, except that one permits looping and the other does not. An algorithm which does not permit looping does not necessarily result in lower delay, less variable delay, higher throughput or less congestion than an algorithm which does. We simply do not believe that a small amount of transient looping should be regarded as a problem.

Are there any ways in which the old algorithm is better than the new one? The new algorithm does take about three times the memory as the old one, but conservation of memory is not generally considered to be an important desideratum for routing algorithms. From the point of view of performance, the new algorithm seems to dominate the old one in every respect. This is not to say that our approach is appropriate for every possible application, or even that it is the only possible approach for *our* application. We do believe, though, that we have met our goal of designing a new routing algorithm which kept the known strengths of the old one while eliminating many of its known weaknesses.

APPENDIX

This Appendix gives a semiformal description of the algorithm to calculate and update the shortest path tree. SOURCE denotes the node in which the algorithm is running. The algorithm's basic data structure, LIST, is a variable-length list whose elements are ordered triples. An ordered triple T is of the form ⟨SON, FATHER, DISTANCE⟩ where SON and FATHER are nodes and DISTANCE is a member. (We use the notation SON (T) in the obvious way to mean the first element of the triple T.) Each triple represents a particular path from SOURCE to SON. The penultimate hop on this path is FATHER, and the total length of this path is DISTANCE. The algorithm we describe here has been modified so that changes to the tree can be computed incrementally, without having to recalculate those

parts of the tree that do not change. Hence, it does not correspond exactly to the flow chart in Fig. 1.

SPF Algorithm

0) If no tree exists, place ⟨SOURCE, SOURCE, 0⟩ on LIST, and go to Step 4).

1) If the change was to line AB, then perform one of the following steps.

a) If AB is in the tree, set DELTA equal to the change in distance along AB.

b) If AB is not in the tree, set DELTA equal to the distance to node A plus the distance along AB minus the distance to B; if DELTA is greater than or equal to 0, done.

2) Identify B and all of B's descendants (both first generation and succeeding generations) as members of the subtree; increase the distances of all subtree members by DELTA.

3) For each subtree node S, perform one of the following steps.

a) If DELTA is positive, try to find a shorter path to S via each of S's neighbors that is not in the subtree; if such an improved path is found, put the triple representing S on LIST.

b) If DELTA is negative, try to find a shorter path to each of S's nonsubtree neighbors by attempting to route each neighbor via S; if such an improved path is found, put the triple for the neighbor node on LIST.

4) Search LIST for the triple T with the smallest DISTANCE. Remove T from LIST; place SON (T) on the shortest path tree so that its father on the tree is FATHER (T). (Exception: if SON (T) = SOURCE, place it in the tree as its root.)

5) For each neighbor N of SON (T), do *one* of the following steps.

a) If N is already in the shortest path tree, then

i) if its distance from SOURCE along the tree is less than or equal to DISTANCE (T) + LINE-LENGTH (SON $(T), N)$, do nothing;

ii) if its distance from SOURCE along the tree is greater than DISTANCE (T) + LINE-LENGTH (SON $(T), N)$, remove N from the tree and place ⟨N, SON (T), DISTANCE (T) + LINE-LENGTH (SON $(T), N)$⟩ on LIST.

b) If there is no triple T' on LIST such that SON $(T') = N$, then place the triple ⟨N, SON (T), DISTANCE (T), + LINE-LENGTH (SON $(T), N)$⟩ on LIST.

c) If there is already a triple T' on LIST such that SON (T') = N, and if DISTANCE $(T') \leqslant$ DISTANCE (T) + LINE-LENGTH (SON (N), do nothing.

d) If there is already a triple T' on LIST such that SON (T') = N, and if DISTANCE (T') > DISTANCE (T) + LINE-LENGTH (SON $(T') N)$, then

i) remove T' from LIST;

ii) place the triple ⟨N, SON (T), DISTANCE (T) + LINE-LENGTH (SON $(T), N)$⟩ ON LIST.

6) If LIST is nonempty, go to step 4). Otherwise, the algorithm is finished.

REFERENCES

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

[2] J. M. McQuillan, "Routing algorithms for computer networks—A survey," presented at the 1977 Nat. Telecommun. Conf., Dec. 1977.

[3] J. M. McQuillan, I. Richer, and E. C. Rosen, "ARPANET routing algorithm improvements—First semiannual technical report," BBN Rep. 3803, Apr. 1978.

[4] J. M. McQuillan, I. Richer, E. C. Rosen, and D. P. Bertsekas, "ARPANET routing algorithm improvements—Second semiannual technical report," BBN Rep. 3940, Oct. 1978.

[5] E. C. Rosen, J. Herman, I. Richer, and J. M. McQuillan, "ARPANET routing algorithm improvements—Third semiannual technical report," BBN Rep. 4088, Apr. 1979.

[6] J. M. McQuillan, G. Falk, and I. Richer, "A review of the development and performance of the ARPANET routing algorithm," *IEEE Trans. Commun.*, Dec. 1978.

[7] J. M. McQuillan, I. Richer, and E. Rosen, "ARPANET routing study—Final report," BBN Rep. 3641, Sept. 1977.

[8] J. M. McQuillan, "Enhanced message addressing modes for computer networks," *Proc. IEEE (Special Issue on Packet Communication Networks)*, Nov. 1978.

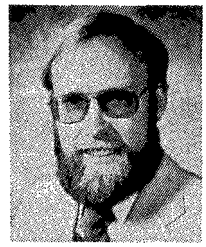
[9] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. Ass. Comput. Mach.*, vol. 24, pp. 1–13, Jan. 1977.

[10] E. Dijkstra, "A note on two problems in connection with graphs," *Numer. Math.*, vol. 1, pp. 269–271, 1959.

[11] D. P. Bertsekas, "Dynamic models of shortest path routing algorithms for communications networks with a ring topology," in preparation.

[12] E. C. Rosen, "The updating protocol of the ARPANET's new routing algorithm: A case study in maintaining identical copies of a changing distributed data base," in *Proc. 4th Berkeley Conf. Distributed Data Management and Comput. Networks*, Aug. 28–30, 1979, pp. 260–274.

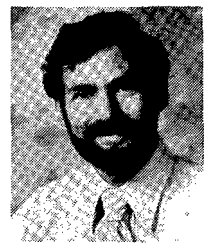
★



John M. McQuillan (M'77) was born in New York, NY, on February 23, 1949. He received the A.B., S.M., and Ph.D. degrees in 1970, 1971, and 1974, respectively, from Harvard University, Cambridge, MA, all in applied mathematics.

Since 1971 he has been with Bolt Beranek and Newman (BBN) Inc., where he was a major contributor to the design and implementation of the ARPANET. He has investigated several types of advanced computer communications systems, directed BBN's consulting in this field, and published over 30 articles in this area.

★



Ira Richer (S'58–M'63) received the B.E.E. degree from Rensselaer Polytechnic Institute, Troy, NY, in 1959, and the M.S. and Ph.D. degrees in electrical engineering from the California Institute of Technology, Pasadena, in 1960 and 1964, respectively.

After holding postdoctoral positions at the Technical University of Denmark, Lyngby, and at Cal Tech, Pasadena, he joined M.I.T. Lincoln Laboratory, Lexington, MA, where he was involved in a number of advanced communications projects that spanned orders of magnitude in both the frequency and altitude domains (ELF to UHF and submarine to satellite). In 1977 he joined Bolt Beranek and Newman (BBN) Inc., Cambridge, MA. As a Senior Scientist at BBN, he consults to both commercial and government organizations on a variety of network and communications topics.

★

Eric C. Rosen received the S.B. degree in 1973 from the Massachusetts Institute of Technology, Cambridge, and the Ph.D. degree in philosophy in 1976 from Princeton University, Princeton, NJ.

He has worked for M.I.T.'s Laboratory for Computer Science, for Boeing Computer Services, and since 1977, for the Computer Systems Division of Bolt Beranek and Newman (BBN), Inc. While at BBN, he has been extensively involved in measuring the performance of the ARPANET, and in designing and testing the ARPANET's new routing algorithm. He is currently conducting research in routing and congestion control techniques for computer networks.