Bolt Beranek and Newman Inc. 12 bbn

Report No. 4931

ARPANET Routing Algorithm Improvements

Volume 2

J.F. Haverty, B.L. Hitson, J. Mayersohn, P.J. Sevcik, and G.J. Williams

March 1982

Prepared for:
Defense Advanced Research Projects Agency
and
Defense Communications Agency

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 4931	2. GOVT ACCESSION NO. AD-A121350	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ARPANET Routing Algorithm Improvements Volume II		5. TYPE OF REPORT & PERIOD COVERED Technical Report 9/1/80 - 4/15/82
		6. PERFORMING ORG. REPORT NUMBER 4931
7. AUTHOR(s) J. Haverty, B. Hitson, J. Mayersohn, P. Sevcik and G. Williams		8. CONTRACT OR GRANT NUMBER(s) MDA90-78-C-0129
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street, Cambridge, MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order 3491
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects 1400 Wilson Blvd., Arlington, VA 22290		12. REPORT DATE March, 1982
		13. NUMBER OF PAGES 276
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Defense Supply Service - Washington Room 10 245, The Pentagon Washington, DC 20310		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

UNCLASSIFIED/UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer networks, ARPANET, routing algorithms, network simulation, SIMULA, adaptive routing, multi-path routing, gateway, internet, catenet, TCP

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report covers work performed during the second year of the extension to the ARPANET Routing Algorithm Improvements Contract. The ARPANET simulator developed during the first year of the extension is used to investigate the performance and behavior of a number of routing algorithms, including the current ARPANET SPF algorithm. Results from the simulator are compared to measurements of SPF running on a small test network, measurements of the line protocol on the operational ARPANET, and the predictions (cont'd)

20. (Continued)

of a stability model developed during the original contract. The simulation was run on a 14-node network using fixed single-path, fixed multi-path, and SPF (adaptive) routing. The performance of each routing method as a function of network load is compared to the predictions of a queueing model. As part of the deisgn of an Internet, this report discusses design issues in the implementation of gateways, including the host interface to the Internet, interoperability of autonomous gateway systems, congestion control, and logical addressing.

BBN Report No. 4931

ARPANET Routing Algorithm Improvements

Volume II

March 1982

SPONSORED BY
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY AND
DEFENSE COMMUNICATIONS AGENCY (DOD)
MONITORED BY DSSW UNDER CONTRACT NO. MDA903-78-C-0129

ARPA Order No. 3491

Submitted to:

Director
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Attention: Program Management

and to:

Defense Communications Engineering Center
1860 Wiehle Avenue
Reston, VA 22090

Attention: Capt. Wade Nielsen
Code R820

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

FIGURES

TABLES

1 INTRODUCTION

This report covers work performed during the second year of the extension of the ARPANET Routing Algorithm Improvements contract. The bulk of the work concerned development, validation, and use of a network simulator to investigate the behavior and performance of the new ARPANET SPF routing algorithm.

In Section 2 we present the results of a series of experiments to compare the simulator to the ARPANET, a laboratory test network running the ARPANET protocols, and a mathematical model of the stability of the SPF routing algorithm. The purpose of these experiments was twofold: first, to increase our confidence in the correctness and validity of the simulator; and second, to use the simulator to increase our understanding of how the SPF routing algorithm operates.

In Section 3 we describe our use of the network simulator to compare the performance of the SPF routing algorithm with the two other routing methods. We used a 14-node test network and ran the simulator in different modes using the three routing methods. The results from the simulator were compared with an analytic model based on queueing theory. The results show, for example, that the SPF algorithm is more sensitive to heavy traffic loads than either of the other two methods.

In Section 4 we give a detailed description of the protocols and processes modelled by the simulator. One observation we made during the development of the simulator was that its behavior and performance varied greatly with small changes in the protocols or processing model. It is therefore important that the simulator be described as exactly as possible if our results are to be useful.

In Section 5 we describe a modelling tool used for comparison with the simulator. This tool accepts a network description and traffic load, much as the simulator does, but uses a queueing model rather than simulation to generate a performance estimate for the network. The tool can also be used to generate an optimal fixed multi-path routing for the given network and traffic. Using the queueing model to estimate performance, the tool considers alternate routings until it finds one with the best performance.

Section 6 describes our experience in implementing the proposed ARPANET congestion control algorithm described in Volume I of this report. We found that although the design is adequately described, many details had to be resolved during the implementation.

As part of this contract, we have issued a number of Internet Experiment Notes (IENs) discussing issues in the design of internetwork gateways. Section 7 briefly describes this work.

The appendices contain documentation which is useful for anyone attempting to use the simulator or modelling tool. Appendix A is a manual for the use of the simulator, and Appendix B is a manual for the use of the modelling tool (including examples). Appendix C gives an example run of the simulator.

2 COMPARATIVE TESTS ON THE SIMULATOR

In this chapter we will compare the behavior of the simulation with measurements taken on a test network, with measurements on the ARPANET, and with the predictions of a mathematical model developed to predict the stability of the SPF algorithm. When convenient, simulation experiments are referred to by their internal designations which are of the form "X#" (e.g., X32, X106).

2.1 Four Node Laboratory Network

During the development period, the SPF algorithm was tested on a four-node laboratory network using a variety of traffic patterns. The results of these tests were reported in "ARPANET Routing Algorithm Improvements, Volume I" [3]. The purpose of these experiments was twofold: first, to validate the correct operation of the simulator; second, to repeat the experiments using our tools for producing statistically valid performance estimates. Because of slight differences in the traffic and protocols running on the network, quantitative comparison between the simulation and the test network is not possible. The first four tests were run on the topology shown in Figure 1. All lines are 50KB and all nodes are H316 IMPs.

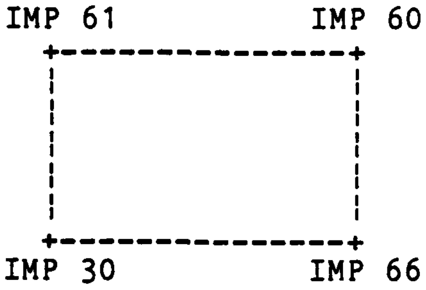


Figure 1 . Topology of Test Network

The four tests differed only in the traffic which was running in the network. In all tests the complete ARPANET protocols were running, except the end-to-end protocol, and traffic consisted of 1192 bit packets (including 200 bits of protocol and hardware framing) being sent from the message generator on one IMP to the message sink on another. The simulation attempted to duplicate these experiments exactly. That is, the link, line up/down, and routing protocols were running with the correct parameters. In the first experiment 10 packets per second were sent from IMPs 60, 61 and 66 to IMP 30. For each flow, two routes are possible. Table 1 gives, for both the simulation and the test network, the percentage of packets taking each route, the delay in milliseconds for each route, and the average delay for each flow. The results of the test network are taken from Rosen et al. ([3]) Appendix 3(a). This simulation experiment has an internal designation, X30.

		-SIMULATION-		-TEST NETWORK-	
	ROUTE	%	delay (msec)	%	delay (msec)
	60-66-30	59.8	73	60.3	75
	60-61-30	40.2	72	39.4	145
Total	60 -> 30		73		109
	61-30	100	39	99.5	57
	61-60-66-30	0	--	0.5	1136
Total	61 -> 30		39		63
	66-30	100	40	99.3	52
	66-60-61-30	0	--	0.7	599
Total	66 -> 30		40		56

Table 1. [3] Appendix 3(a) vs. X30

The simulation results were analyzed using our statistics package, to produce confidence limits for our performance estimates. We can be reasonably sure that the simulation was run long enough to collect enough samples so that the averages given accurately reflect the behavior of the simulator. Unfortunately this is not true of the results from the test network.

Looking at these results, it is obvious that the simulation does not duplicate the results of the test network exactly. First, the fraction of each route taken by each flow differs by less than 2% in the simulation and test network. Second, the delay along each route for the test network is always larger than in the simulation. Third, the discrepancy is greater for the longer (or slower) routes.

It is worthwhile repeating a point that was made in [3]. If the network and traffic load is symmetrical, why does 60% (rather than exactly half) of the traffic from IMP 60 to IMP 30 follow the route 60-66-30? The reason is that the IMP numbers its lines internally, and line 1 has a higher priority than line 2, and so on. The line from 60 to 66 has higher priority than the line from 60 to 61, so it is serviced faster. Since it is serviced faster, the routing algorithm tends to prefer it. As you can see from the above results, this asymmetry is faithfully preserved in the simulation.

The second experiment (X31) was the same as the first, but with double the traffic: 20 packets per second were sent from IMPs 60, 61 and 66 to IMP 30. In both the test network and the simulation, a small fraction of the packets took looping paths. These packets have been omitted from the results below. The fraction of packets omitted can be deduced from the sum of the percentages shown. The results of the second experiment, in the same format as before, are shown in Table 2.

		-SIMULATION-		-TEST NETWORK-	
	ROUTE	%	delay (msec)	%	delay (msec)
	60-66-30	52.2	430	65.3	1387
	60-61-30	47.4	492	33.3	1841
Total	60 -> 30		464		1555
	61-30	88.6	158	94.1	564
	61-60-66-30	11.4	887	5.9	1924
Total	61 -> 30		242		644
	66-30	86.4	177	99.7	437
	66-60-61-30	13.5	1104	0.3	882
Total	66 -> 30		303		438

Table 2. [3] Appendix 3(b) vs. X31

Comparing these results with the previous set, we see that the discrepancy between the simulation and test networks is even larger. The fractions of each flow along each route differ by up to 15%. Again, the discrepancy in the delay figures is larger than for the flow fractions. It is obvious that the network is completely saturated for both the simulation and the test network, since delays across the four-node network range from 158 milliseconds to nearly 2 seconds.

Note that the asymmetry in the flow from 60 to 30 which we mentioned earlier has diminished in the simulation results. This is perhaps due to the fact that the importance of scheduling effects is lessened as line loading (and hence queueing times) increases. Unfortunately, we do not observe this effect in the test network. It is possible that this is due to the fact that the results from the test network are not statistically valid.

The results for the flows from 61 and 66 to 30 show that, most of the time, the routing algorithm selects the better (one-hop) path, both in the simulation and the test network. The (combined) percentage of packets taking a three-hop route is much higher in this experiment than in the previous one where the traffic was lower: 3.1% vs. 0.6% for the test network, and 12.5% vs. 0% for the simulation. The difference between the test network and the simulation may be due to statistical variation. The increased use of the longer path reflects the fact that the routing algorithm becomes less stable as the network approaches saturation. A word is in order about the meaning of the term "stable" as it applies to the routing algorithm. The routing algorithm will no more choose exactly the same route each time it runs than an air conditioner will keep the temperature exactly constant. If the minimum-hop path from a node to a given destination is not unique (i.e., if there is more than one path with the minimum number of hops), then the routing algorithm may switch between them depending on small fluctuations in traffic. Switching between approximately equivalent routes is perfectly acceptable, and we could call the routing algorithm stable. If, however, the routing algorithm often chose routes longer than the minimum-hop path, it would not be considered stable.

The results show that under heavy load the SPF algorithm selects a route longer than the minimum-hop path a small percentage of the time. This behavior, known as "probing," is a necessary consequence of this form of adaptive routing. Under load, SPF will switch to a longer route in order to determine if a longer, less congested path would be faster than a shorter path. The algorithm will quickly revert to the original path if this is not the case.

The third experiment in this series (X32) applies an asymmetric load to the network: 10 packets per second are sent from IMPs 60 and 66 to IMP 30. The results are shown in Table 3.

Looking first at the flow from 66 to 30, we see that the simulator is much less stable than the test network: nearly 30%

		-SIMULATION-		-TEST NETWORK-	
	ROUTE	%	delay (msec)	%	delay (msec)
	60-66-30	44.1	72	49.0	52
	60-61-30	55.8	68	51.0	49
Total	60 -> 30		70		51
	66-30	72.1	40	100	25
	66-60-61-30	27.9	100	0	--
Total	66 -> 30		57		25

Table 3. [3] Appendix 3(c) vs. X32

of this flow took the worst-case 3-hop route in the simulator, while none did in the test network. In view of this discrepancy, the results for the flow from 60 to 30 are understandable: the percentages of the flow taking each route are similar for the simulator and the test network, and (because of the instability) the delay in the simulation is somewhat higher. Why did no packet of 66 -> 30 flow take the longer route in the test network? There are two possible explanations. First, if the test network was run for a longer time, it is likely that some packets would have taken the longer route; that is, we cannot take 0% as meaning "never." On the other hand, this cannot reasonably explain the difference between 0% and 27.9%. Second, the stability of the network is highly dependent on the traffic pattern, as opposed to its average level. The routing algorithm is quite sensitive to surges in traffic. Since the traffic load in both the simulator and the test network is being generated by a process within the IMP, rather than by an independent source, we cannot be sure that the simulator and test network see a comparable traffic pattern.

The fourth experiment in this series (X33) is the same as the third experiment, but with double the traffic load: 60 and 66 each send 20 packets per second to 30. The results are shown in Table 4.

		-SIMULATION-		-TEST NETWORK-	
	ROUTE	%	delay (msec)	%	delay (msec)
	60-66-30	48.9	282	54.4	730
	60-61-30	50.8	141	44.0	907
Total	60 -> 30		211		821
	66-30	70.0	128	95.1	336
	66-60-61-30	29.9	300	4.9	1050
Total	66 -> 30		181		371

Table 4. [3] Appendix 3(d) vs. X33

These results are similar to those of the previous experiment, except that the network is heavily congested so the average delays are much higher. We see that this time, there is a small flow in the test network along the 3-hop path from 66 to 30; in the simulation, this path is used a few percent more than before. The most interesting thing about these results is that, in the test network, the routing for 60 to 30 is somewhat different from that used in the third experiment. In that experiment, 51% of the flow used the route 60-61-30, which was 3 milliseconds faster than the route 60-66-30. In this experiment, the second route is used by 54.4% of the flow, and it is 177 milliseconds faster. This emphasizes the fact that the results are only approximate, particularly in the test network.

For the next experiment, the network topology was changed slightly: a link was added between 61 and 66. The new topology is shown in Figure 2. The traffic load was the same as that for the first experiment: nodes 60, 61 and 66 each sent 10 packets per second to node 30. The results of this experiment are shown in Table 5.

The first thing to note about these results is that the test network and the simulator are still comparable. Although there are significant _quantitative_ differences between the two sets of results, the behavior of the two networks is approximately the

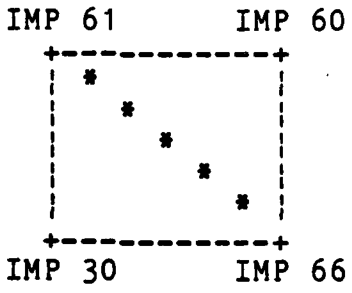


Figure 2 . Modified Test Network Topology

same. For the flow from 60 to 30, 87% is sent on either of the 2-hop paths, and the remainder on a 3-hop path. A small number of packets in the test network took looping paths, which we have *not shown.* *For the flow from* 61 to 30, the fraction taking the shortest, 1-hop path differs by about 4% between the simulation and the test network. This difference is about 5% for the flow from 66 to 30. For the routes which are longer than optimal, the discrepancies between the test network and the simulator are sometimes larger, but for each flow the order of route preference is the same.

If it is possible to draw a conclusion from the results of the five experiments it is this: the performance of the routing algorithm depends on the details of the network on which it is run, but the simulation nevertheless provides a particular environment which is similar to the particular test network used.

	ROUTE	-SIMULATION-		-TEST NETWORK-	
		%	delay (msec)	%	delay (msec)
	60-66-30	34.7	88	68.4	72
	60-61-30	52.2	78	18.6	138
	60-66-61-30	8.3	124	6.8	278
	60-61-66-30	4.8	129	5.7	149
Total	60 -> 30		87		105
	61-30	73.2	47	69.1	60
	61-66-30	26.8	97	28.3	112
	61-60-66-30	0	--	2.5	157
Total	61 -> 30		61		77
	66-30	78.7	47	83.6	44
	66-61-30	21.3	95	12.7	273
	66-60-61-30	0	--	3.3	219
Total	66 -> 30		57		81

Table 5. [3] Appendix 3(e) vs. X34

The qualitative agreement for light loads is quite good; for heavy loads the agreement is fair. This increases our confidence that the simulator is useful for studying qualitative network behavior.

2.2 Simulation of Transcontinental Trunk

During our experimentation with the simulator, an interesting incident occurred in the real ARPANET which prompted us to perform an experiment on the simulator to see if it behaved in a similar manner.

During a periodic reconfiguration of the ARPANET, a somewhat longer transcontinental trunk was added. Because the throughput achievable on a link depends on the speed at which packets can be acknowledged, and because this is naturally bounded by the round-trip propagation delay, an increase in propagation delay will cause a decrease in the maximum achievable throughput on a link. Observation of the ARPANET showed that the introduction of a slightly longer trunk was causing congestion, but in the lines feeding it rather than in the trunk itself. This is because an IMP will reject an incoming packet if resources are not available to transmit it out the next line in its path. A long trunk does not congest, but rather runs out of logical channels, so packets _entering_ the IMP which would be _leaving_ by the trunk are rejected, and must be retransmitted. It is these retransmissions which cause the lines feeding the trunk to congest. Note that the trunk and the feeders are all identical 50 kb lines; it is the topology which distinguishes them. We simulated this situation using the network shown in Figure 3.

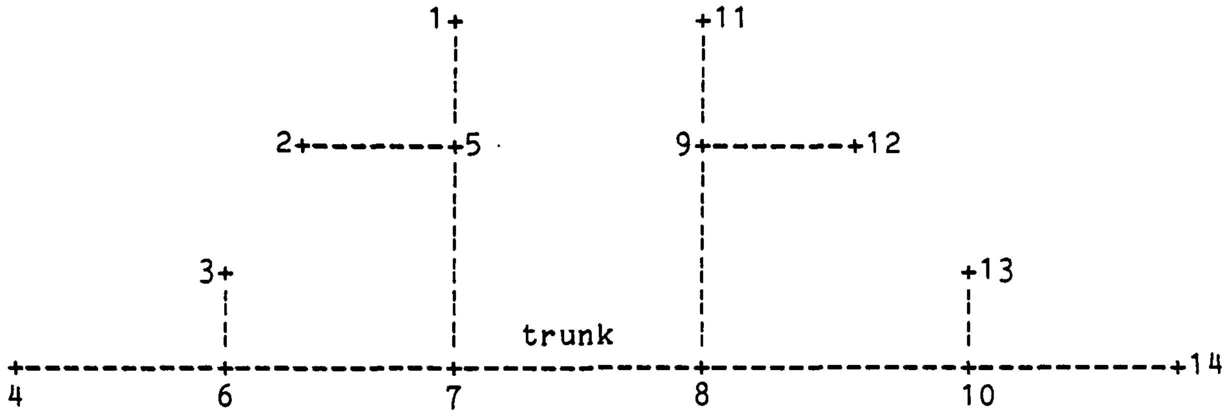


Figure 3 . Simulated Network

For the experiment, 25 packets per second were sent from 1 to 11, 2 to 12, 3 to 13 and 4 to 14, and similarly in the opposite direction. The packets consisted of 100 bits of data and 200 bits of framing and protocol overhead. Without retransmissions the utilization on the middle line (connecting 7 and 8) would be 60% in each direction. We found that the network could not support this level of traffic, and that, averaged over the measurement period, the network carried 24.7K in each direction. The remainder of the 30K offered load simply queued in the hosts. The percentage line utilizations (including retransmissions) for representative lines is given below, together with the load due only to the 24.7K traffic (and not retransmissions and null packets). The load is given as a percentage of the 50 Kbps line capacity; thus, 24.7 Kbps represents 49.4% of a 50 Kbps line, as shown for line 7-8.

Line	Utilization (%)	Load (%)
4-6	40.8	12.4
6-7	55.5	24.7
7-8	68.5	49.4
8-9	38.3	24.7
9-11	21.7	12.4

Before we discuss these results in detail, it is interesting to note that the network throughput is limited by a line whose utilization is only 68%. This compares well to tests done on the real network, where a limit of 76% was observed sending full 1208-bit packets from one IMP to a neighbor.

Now consider the table above. Line 4-6 is carrying a quarter of the traffic in this direction, or 7.5Kbps. Line 6-7 is carrying 15Kbps, and line 7-8 is carrying 30Kbps. However, the utilization on each line is not in these proportions. By dividing the observed utilization by the load we can measure the effect of retransmissions and null packets. This is shown below:

Line	Utilization/Load
4-6	3.30
6-7	2.25
7-8	1.39
8-9	1.55
9-11	1.76

This is identical to what was observed in the ARPANET: the line protocol has the effect of causing retransmission, not in a line which has reached capacity, but rather in the lines which feed it. The small amount of congestion in lines 8-9 and 9-11 is presumably due to the traffic going in the other direction on those lines.

A final observation, for which we will not present the supporting data, concerns the fraction of link-level acknowledgments which are sent in null packets. Link-level acknowledgments can be sent (from the data receiver to the data sender) either in a data packet, or in an empty ("null") packet. Measurements in the ARPANET revealed a surprising fact: that even under heavy load, most acknowledgments are sent in null packets, rather than being piggybacked on data packets travelling in the right direction. Measurements of the simulation are in agreement with this result.

2.3 Comparison with Stability Model

The third series of experiments using the simulator compares the behavior of the simulator to a mathematical model of stability developed by Bertsekas [2]. Bertsekas analyzed the behavior of the routing algorithm for ring topologies, of which the 8-node ring shown in Figure 4 is an example.

```
 7        8        1
 +--------+--------+
 |                 |
 |                 |
6+                 +2
 |                 |
 |                 |
 +--------+--------+
 5        4        3
```

Figure 4 . Bertsekas 8-Node Ring

The traffic pattern (specified in [2] Appendix 6) is for nodes 1 to 3 and 5 to 7 to send equal traffic to node 8. The optimal routing is for nodes 1 to 3 to send traffic counterclockwise and nodes 5 to 7 to send traffic clockwise. The worst routings are for all nodes to send traffic in the same direction, either clockwise or counterclockwise. Bertsekas investigates the use of 'bias' to ensure stability. Bias is simply a network-wide constant which is added to the reported delay on each link. A path of n hops will therefore have n times the value of the bias added to its delay. This tends to bias the algorithm towards choosing the minimum-hop routing. Bertsekas predicts that, for a given bias level, the routing algorithm will be stable (choosing the optimal route) up to a given traffic level, after which it will be unstable (diverging, and switching between the two worst routings). For zero bias, the algorithm will always be unstable, and the higher the bias, the higher the traffic level for which the algorithm is stable.

The ARPANET does not use an explicit bias in its routing algorithm, but it is nevertheless quite stable at moderate traffic levels. In 1978, it was believed that this discrepancy could be explained by a number of factors. First, the delay computation never reports a zero delay on any link: the minimum reported delay is therefore 1 (reporting unit). At the moment, the reporting unit is 6.4 milliseconds. This rounding-up was believed to have the same effect as adding a constant bias. Second, in the ARPANET the IMPs do not all run the routing algorithm at the same time; Bertsekas's model predicts that this asynchrony will increase the maximum traffic load before instability occurs. In our experiments, we first used the 8-node ring and traffic pattern described above to investigate the effect of the choice of the delay reporting unit, IMP synchronization, and a third factor, the choice of initial routing. The traffic from each source was either 1.0, 3.0 or 6.9 1200-bit packets per second; the initial routing was either optimal or worst case; the delay reporting unit was either 6.4 milliseconds (its current value in the ARPANET) or 0.8 milliseconds, and in all but one case the IMPs were running synchronously. Finally, this experiment represents a worst case topology and traffic pattern. In the real network some traffic is always flowing on a given link, so the delay estimate used by the routing algorithm will not usually drop to its zero-traffic value. The background traffic in the network thus provides another form of bias which tends to increase the stability of the SPF algorithm running in the real network. The result of each run was the hop count averaged over all packets. If the routing was always optimal, this figure would be 2.0; if the routing was always worst case, the average hop count would be 4.0. [This can be seen by inspection of the topology, given above.] The results of 8 runs with different combinations of parameters are shown in Table 6.

Consider the last four runs where the network is heavily loaded. The worst performance is for experiment 82, which uses a worst-case initial routing, a small delay reporting unit (0.8 msec), and synchronized routing updates. For each of these

Exp #	Traffic[1]	Init Routing	AVGUNIT[2]	Synch[3]	Mean Path Length
36	1.0	optimal	6.4	yes	2.0
37	1.0	worst	6.4	yes	2.3
94	3.0	optimal	0.8	yes	2.3
95	3.0	worst	0.8	yes	2.4
80	6.9	worst	6.4	yes	2.5
82	6.9	worst	0.8	yes	2.9
83	6.9	worst	0.8	no	2.5
93	6.9	optimal	0.8	yes	2.5

[1] pkt/sec per source, 1200-bit packets; 6 sources

[2] average delay reporting unit

[3] routing changes synchronized or not

Table 6. Results for 8-Node Ring

parameters, this choice of parameter value should give the worst (i.e., least stable) performance. We see that for this parameter choice the average hop count is 2.9, but a change in any single parameter causes the average hop count to improve to 2.5. This is a very striking result: the routing algorithm is fairly stable at high traffic load unless three impossible conditions are met!

The first four experiments show the behavior of the simulator at lower traffic levels. In particular, experiment 95 uses the same parameter choices as experiment 82, and demonstrates that the instability of the algorithm in that run is much smaller at a lower traffic level. Experiment 36 shows that, with a low traffic level and realistic choices for parameters, the routing algorithm is completely stable.

In order to verify that these results were not simply a function of the topology chosen, a second series of runs used the 4-node ring shown in Figure 5. As before, all lines are 50 Kbps.

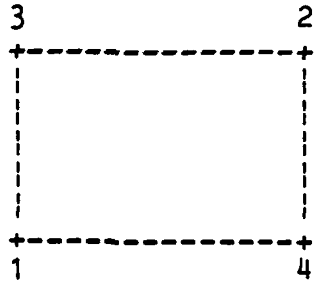


Figure 5 . Bertsekas 4-Node Ring

For this network, equal traffic (again consisting of 1200-bit packets) was sent from nodes 3 and 4 to node 1. There are only 3 possible routings: the optimal routing is for 3 and 4 to send traffic directly to 1, and the two worst-case routings are for traffic to be sent either clockwise (3-2-4-1) or counterclockwise (4-2-3-1). Because this system is simpler than the 8-node ring, results are reported as the fraction of time the routing algorithm used the optimal routing. The results, again for a variety of traffic levels and parameter choices, are shown in Table 7.

Exp #	Traffic[1]	Init Routing	AVGUNIT	Synch	% Optimal
96	10	worst	6.4	yes	49
97	20	worst	6.4	yes	45
98	10	optimal	6.4	yes	89
99	20	optimal	6.4	yes	58
106	10	worst	6.4	no	97
108	20	worst	6.4	no	48
105	10	optimal	6.4	no	100
107	20	optimal	6.4	no	93

[1] pkt/sec per source; 1200-bit packets; 2 sources

Table 7. Results for 4-Node Ring

Notice that in this series of experiments, the delay reporting unit was always 6.4 milliseconds. The 8 runs shown represent the 8 possible combinations of traffic load (10 or 20 packets per-second per-source), initial routing (optimal or worst-case), and synchronization (routing changes synchronized or not). An examination of these figures will show that an increase in traffic, a worst-case initial routing, or synchronized routing updates always causes a decrease in the fraction of time the network routing was optimal. Regression analysis shows that the effect of the three factors mentioned is approximately equal.

These results are roughly consistent with Bertsekas's analysis. First Bertsekas analyzes asynchronous updating using a continuous model ([2], Appendix A.4.2 algorithm 2, page A-60ff) and concludes that asynchronous updating does significantly increase stability. Second, Bertsekas recommends a high value of "bias" to ensure stability. In our experiments, bias is modelled by the grain of delay reporting, and the fact that a delay of zero cannot be reported. Our results show that this form of bias improves stability, and a large bias is not required. Bertsekas does not specifically address the question of the effect of the initial routing. It is not surprising, however, that a network started with the optimal routing will perform better than one started with a worst-case routing.

We therefore differ with Bertsekas not in his explicit results, but rather in his implication that a network using the ARPANET routing algorithm is in a delicate state, in fact, teetering on the brink of instability. While it is true that the routing algorithm will become less and less stable at high traffic levels, and in particular as the network becomes congested, *in practice*, a number of factors contribute to making the algorithm stable.

3 NETWORK PERFORMANCE AS A FUNCTION OF ROUTING

The second series of experiments using the simulator forms the core of the work reported here. The goal of this series of experiments was to compare the performance of the SPF routing algorithm with other plausible alternatives. Such a comparison naturally depends on what SPF is compared to, and what standard of comparison is used.

We will base our analysis on Gerla [5]. Gerla starts with a general model of static routing, that is, routing which does not change over time depending on network conditions. In his model, each source and destination has associated with it a set of alternate routes. For a given source and destination, each route in the set is assigned a fraction between 0 and 1, representing the fraction of the total traffic travelling between that source and destination which is sent over that route. The model assumes that the process of dividing the traffic between the routes is perfect. If, for example, 20 kbps is travelling from X to Y over one of two alternate routes, each with weight 0.5, then each route sees exactly 10 kbps.

When a packet enters a node in the network, the node must decide whether it should be delivered to a host attached to the node or be transmitted to another node. In the latter case, the node must decide which neighboring node to send it to. This process, known as forwarding, is routing from the point of view of a single node rather than the network as a whole. In Gerla's general model, the node must know the source and destination of the packet, and the assigned route of the packet. For each source and destination, and *for each route which passes through the node*, the node must know the next node in the route. This means that for each source and destination, a fixed proportion of the traffic *through that node* will go to each neighbor, although the decision as to which packet goes to which neighbor will have been made when the packet was assigned to a particular route.

Gerla's general model is not really feasible, since each node must store a considerable amount of routing information. However, he also proposes an alternate model which is specifically designed to reduce the information required by each node. In this model, a packet is not assigned to a permanent route at the source; instead, it is forwarded by each node depending only on its destination. Each node has a table that specifies the fraction of the traffic for each destination that should be forwarded to each of the node's neighbors.

We call Gerla's general model "source-dependent routing," and the alternate model "source-independent routing."

Finally, Gerla considers routing where only a single path is specified from source to destination. We call this "single-path routing," and the alternative "multi-path routing." Single-path routing may be either source-dependent or source-independent. In the case of source-independent single-path routing, packets arriving at a node for a particular destination are always forwarded to the same neighbor; in the case of source-dependent single-path routing, the neighbor depends on the source as well as the destination of the packet.

For each kind of routing, Gerla gives an algorithm for generating the routing which minimizes a given objective function. He considers measures of performance which are a function of the flow on each link in the network; for example, a typical measure of performance is the average delay caused by a message crossing the network. Of course, a routing which is optimal with respect to a particular objective function may not be optimal with respect to another. Various functions suggested by Gerla are discussed below.

A objective function is one kind of standard which can be used to compare different routing schemes, but is by no means the only kind. A comparison standard may not be a function at all. We should resist the temptation to reduce every problem to numbers! A certain routing may be preferable because it is more robust and therefore functions well in the presence of node or line failures. A good routing should perform well with traffic flows which are different from those used to design it.

Throughput, an important measure of network performance, is not a function of the flow on each link. Other things being equal, a routing which allows the network to support a higher throughput is preferable. Of course such a comparison is only possible if other measures of performance, such as average delay, are the same for all routings.

Delay is an important measure of performance since it is amenable to analysis. The average time needed by all packets to cross the network is a function of the total load on each link. The total load on each link is a simple function of traffic and routing. This means that it is simple to compute the effects that even infinitesimal routing changes have on delay.

Simple average delay is not entirely satisfactory as a standard for comparison. It has been shown by Meister et al. [6] that a routing which minimizes average delay will cause some packets to have relatively large delays, offset by other packets with relatively small delays. The variance of the delay is a good measure of the extent to which the delays of individual packets differ from the average. This can be used as a secondary standard of comparison between routings which have similar average delays.

Since it is not possible to simultaneously minimize both average delay and variance of delay, the objective function chosen for Gerla's algorithms must be a compromise. A suitable objective function is suggested in [6]. If the routing is chosen to minimize the k-th root of the average over all packets of the k-th power of delay, then for large k the optimal routing will be one which tends to avoid wide variations in packet delays at the expense of somewhat larger average delay.

The SPF routing algorithm used in the real network does not attempt to achieve a global minimum for some objective function, but instead tries to individually minimize the delay for each packet. In addition, the routing is not static, but varies with the delay on each line. Indeed, the route a packet takes cannot be predicted when it enters the network, since the route may change as it crosses the network. Gerla considers an idealized model of adaptive routing, and concludes that adaptive routing can be modelled by static routing which is chosen to minimize a particular objective function. He calls this static routing policy "idealized adaptive routing."

3.1 Topology of the Simulated Network

We have said that we will compare SPF routing to the routing generated by Gerla's algorithm for a given objective function and (related, possibly identical) standard for comparison. We must still decide on the network on which the comparison will be run. This involves defining a network topology and a traffic matrix. The topology includes the network nodes and the lines which connect them, and the traffic matrix specifies the flow from each source node to each destination node. For each source-destination flow we must specify the rate and distribution of message arrivals, and the distribution of message lengths.

Two approaches are possible in defining the topology of the target network: 1) construct a target network as much like the real ARPANET as possible; or 2) use an imaginary network constructed so that it has particular properties (such as a particular connectivity). Since the ARPANET is now quite large, another possibility is to model the target network on the ARPANET at an earlier point in its development. Our first validation tests of the simulation used a 19-node network, taken from Kleinrock [4] and based on the ARPANET.

Because of the modelling tools we are using, the network must be a much smaller size than the present ARPANET. However, we want to model the performance of the SPF algorithm on the present ARPANET, rather than the network at an earlier point in its development. We are therefore forced to construct an imaginary network which is as much like the present ARPANET as possible, but considerably smaller. There are many ways to specify network characteristics which are independent of size. Connectivity (average number of lines per node) is perhaps the most obvious.

The connectivity of the ARPANET, at the time of writing, was about 2.4. It is not clear what effect this figure has on the operation of the SPF algorithm, so it does not seem to be a very good way of ensuring that the model will correspond to the real network. On the other hand, connectivity is clearly not irrelevant to the operation of the SPF algorithm. It could be argued, for example, that a low connectivity figure implies that SPF has few alternate routes from which to choose, whereas a high connectivity figure implies that there are many alternate routes.

Other characteristics which are independent of network size are the size and number of small loops. A particular routing scheme may be very sensitive to the existence of small loops. For example, the original ARPANET routing scheme would generate loops in the routing if the network contained loops of three nodes. In the present ARPANET, the smallest loop is eight nodes.

Average hop count is the average number of hops in the minimum-hop path given by each source-to-destination pair. Each source-destination pair can be either weighted equally or in proportion to the traffic between them. Arguably, the average hop count and connectivity together determine the number of alternate routes between a given source and destination. That is not to suggest that there is an arithmetical relationship between hop count, connectivity, and number of alternate paths, but rather that the larger the hop count and connectivity, the larger the number of alternate paths will tend to be.

The SPF routing algorithm will switch from the minimum-hop path to another path when the total delay on an alternate path is less than the total delay on the minimum-hop path. Since there are more hops on the alternate path, the (average) delay on each hop of the alternate path must be smaller. In fact, the delays on each path must be inversely proportional to the ratio of the number of hops. This suggests that an indication of the stability of the SFP algorithm, running on a given topology, is the ratio (averaged over all source-destination pairs) of the minimum-hop path to the next best path.

Not all these network parameters are necessarily equally useful in predicting the behavior of the SPF algorithm. We have designed a 14-node test network which is based on the ARPANET, but it differs in a number of the parameters discussed above. The connectivity of the test network is quite close to that of the ARPANET, but because the test network is much smaller than the ARPANET the average hop count is much smaller. The topology of the test network is shown in Figure 6.

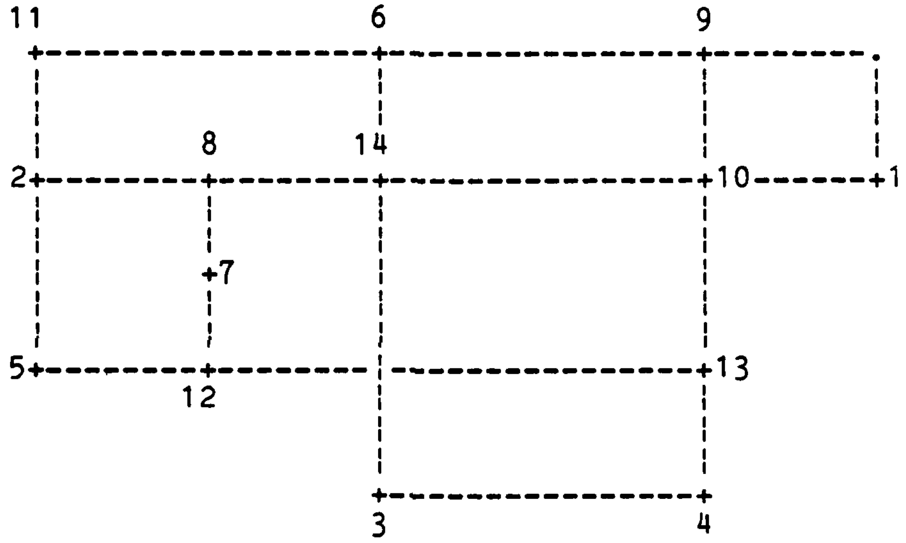


Figure 6 . Simulator Test Network

3.2 Defining Network Load

Just as the network topology may be imaginary or based on a real topology, the traffic matrix may be imaginary or based on real traffic figures. In either case, we want to scale the traffic up and down to test network performance for both high and low traffic levels. Gerla uses a test network where the topology is taken from a 26-node 30-line ARPANET, and the flow from one node to another is either 1 or 0. Note that because he scales the traffic, the initial traffic matrix is effectively dimensionless; it simply defines the ratio of the sizes of different source-destination flows. Gerla's traffic matrix is given by specifying the number of active node pairs, selecting node pairs at random, and specifying the traffic scaling factor. Suppose that the traffic scaling factor is chosen so that the total traffic is equal to some predetermined level. The number of active node pairs is then a measure of how balanced the traffic is: if every node pair is active the traffic is quite balanced; for small numbers of active node pairs the traffic is more concentrated. Gerla generates traffic in this way since he is interested in the effect of traffic balance on his algorithms. Similarly, we might use this procedure to examine the effect of traffic balance on SPF routing.

Gerla does not address the question of the confidence interval of the results with respect to the random selection of node pairs, but gives results for different random selections which differ by up to 15%.

Gerla's algorithms generate the routing which minimizes a given objective function. We will now discuss the assumptions necessary to make this true for a real network. We will tacitly assume that the objective function is average delay, but this is just for convenience; what we have to say below does not depend on the choice of objective function.

Three things are required in order to use Gerla's algorithm to generate the optimal routing for a real network. First, the topology and traffic used in the experimental network must accurately represent the real network. Second, the analytic model used to estimate the value of the objective function must be valid for the experimental network. Third, the algorithm must find the true global minimum of the objective function. We have already discussed various strategies we could use to generate the test network topology and traffic matrix. Now suppose that we are minimizing average delay. As we have said, the average packet delay can be computed from the total load on each link, but this is true only if you assume a certain model of the network.

The standard network model was developed by Kleinrock [4]. It requires that traffic is generated and enters the network with a negative-exponential inter-arrival time distribution, and negative-exponential packet lengths. In order to calculate the delay on each link, Kleinrock assumes that, on each link, packet arrivals are independent and have negative-exponential inter-arrival times. Given these assumptions, the average delay on each link can be calculated using an M/M/1 queueing model. The average packet delay is the average of the delay on each link, weighted to take into account traffic levels and hop count. We are willing to assume that Kleinrock's assumptions are valid since predictions based on them agree with simulation results for networks which satisfy his preconditions.

In the ARPANET, on the other hand, messages have a maximum length (around 8K bits), and a message may be divided into several packets, where a packet has a maximum length of around 1K bits. Even if message arrivals were negative-exponential, the fact that a message may be divided into several packets means that packet arrivals cannot be.

If we want to use average packet delay as our objective function for Gerla's algorithm, we must rely on Kleinrock's model -- there is no alternative. This means that we must justify the assumption that packet arrivals on each link are independent and have negative-exponential inter-arrival time distributions, even if packet lengths are not negative-exponential, and do not enter the network with negative-exponential inter-arrival times. We can verify this assumption in the same way that Kleinrock verified his original assumption: by comparing the predictions of the model with simulation results.

We will define two different models of the network. One will match the ARPANET as closely as possible, and the other will match the assumptions of the analytic model as closely as possible. We can run the analytic model on the latter network, and can run the simulation on both. Ideally, we would expect close agreement between the analytic model and the simulation for the network on which both are run, and good agreement in the other case. Although this procedure does not compare the results of either the analytic model or the simulation to the behavior of the real network, we can be more confident that the assumptions underlying the analytic model are valid if the results of the model agree with the results of the simulation (which does not depend on the same assumptions).

To generate a small test network, we first looked at traffic statistics on the real ARPANET. For the peak hour we used, the total data traffic, including the 200-bit framing and protocol overhead per packet, was 57.0 kbps. The traffic between the highest 14 traffic sources was 22.2 kbps, or 39%. We used those 14 IMPs and the traffic between them as the basis for our test network. We reduced each IMP-IMP flow to a flow consisting of messages of two lengths (using a process described below). The lines in the reduced network, and the propagation delay on each line, were taken from the real ARPANET: two IMPs in the reduced network were connected if there was a short path between them in the real ARPANET, and the propagation delay on the line was set to the sum of the propagation delays along the corresponding path in the real ARPANET.

This produced a network with 3 nodes in Southern California, 3 in Northern California, 3 Mid-Western nodes, 3 nodes in the Boston area, and 2 in Washington, D.C. Considering the procedure used to create this test network, we feel it represents as good a match to the ARPANET as is possible with just 14 nodes.

Because of the way traffic is generated in the simulator, it is necessary to keep the number of different IMP-IMP flows in the network as small as possible. The simulator accepts a description of the network traffic in the form of individual traffic flows, specified by:

source (IMP and host)
destination (IMP and host)
average message length
average message rate

The distribution of message lengths can be either constant or negative-exponential. We will discuss how to approximate the real message length distribution by several flows with constant length. Message arrivals may be either deterministic or Poisson; we used Poisson arrivals. Our goal is to construct a traffic matrix consisting of individual source-destination flows that can be used as input to the simulator.

Statistics gathered from the ARPANET over a particular (i.e., peak) hour provide a source-destination traffic matrix specifying the average traffic (in bits/second) between each pair of IMPs. Summary statistics for each node specify various parameters for messages entering and leaving the network at that node: the distribution of packets/message, the message rate, and the average packet length. There are two problems with the data: the traffic matrix includes packet overhead (200 bits) and control traffic (e.g., RFNM's), but the summary statistics do not; and the summary statistics do not provide separate statistics for each source-destination pair, but rather data for each source averaged over all destinations. A third problem is caused by the nature of the simulation: we need to approximate the message length distribution for each source-destination flow in the ARPANET by a number of flows with constant message length.

We will discuss the third problem first. Examination of the distribution of packets/message shows that almost all messages are a single packet (94%). A disproportionate fraction of the remainder (25%) are 8-packet messages. This corresponds to the hypothesis that almost all traffic is Telnet character-by-character, and that the remainder is mail or FTP. Therefore, we chose to model the real message length distribution by two constant message lengths: short packets and full (i.e., 8063-bit) 8-packet messages. The length of the short packets is given by the average length of the last (and hence probably not full) packet of a message. The ratio of short packet traffic (in bits/second) to long message traffic (in bits/second) in the simulation is given by the ratio of 1-4 packet message traffic to 5-8 packet message traffic in the real network. In effect, this means that network messages of 1-4 packets are simulated by single-packet messages, and network messages of 5-8 packets are simulated by 8-packet messages.

Now we will discuss how to correct the traffic matrix for RFNM's and overhead and how to generate particular source-destination flows. Since we have statistics (message length, etc.) only for each source, we assume that the traffic to each destination from a particular source has identical characteristics. For example, if we suppose that the average packet length of all packets transmitted from a particular source is 147 bits, we assume that the average packet length of packets sent from that source to each particular destination is 147 bits.

For each source we know the message rate into and out of the network, and the average number of packets per message (for messages in each direction). This statistic refers only to data messages, but we know that a RFNM is generated by the source for each message delivered to a host (i.e., for which it is the destination). Thus the total number of messages generated by the source is just the number of data messages entering the network plus the number of data messages leaving the network at the source. We assume that RFNM's are the only control messages (this assumption is also made in constructing the traffic matrix). Since we can calculate the number of packets/second, we can adjust each element of the traffic matrix to remove the 200-bit per-packet overhead.

We have now computed the traffic we wish to simulate between each source-destination pair, and we must divide this traffic between single-packet messages and full messages. The traffic consists of data messages, for which we know the message length distribution (by number of packets) and message rate, and RFNM's, for which we know the rate and length. As we discussed above, we have these statistics only for each source.

From the message length distribution we compute the average number of packets in a message, then correct the average packet length by removing full packets (i.e., all but the last packet) of multi-packet messages. This gives us the length for single-packet messages.

Single-packet messages simulate both RFNM's and 1-4 packet messages. The RFNM's have a fixed length and a known rate; the traffic rate for 1-4 packet messages is computed from the message rate, the message length distribution, and the average length of a message with a given number of packets. The traffic rate for RFNM's and 1-4 packet messages divided by the total traffic rate gives the fraction of each source-destination flow from this source which is to be simulated by single-packet messages.

We now have the traffic rate for each source-destination pair, the proportion of traffic for short and long messages, and the lengths of short and long messages. We simply compute the message rate for short and long messages to each destination, and we are done.

3.3 Comparison with Analytic Model

We ran the first series of experiments on our 14-node network using the generated traffic matrix running the simulator in its "analytical model" mode, where some of the protocols are disabled so that the simulator is simulating the same network model that the analytic model is analyzing. The analytic model was run using optimal single-path routing generated by using Chou's algorithm [7] and optimal multi-path routing generated by using Gerla's algorithm. The simulator was run using the ARPANET SPF routing algorithm, with buffer limitations and the line protocol disabled. Both the simulator and the model were run at a number of traffic loads (or traffic factors): 1.0, 4.0, 8.0 and 12.0 times the traffic matrix generated from the real ARPANET. Multiple packet message flows were converted to the equivalent single packet flow. Packet arrivals were Poisson. The average delay (in milliseconds) across the network was measured. The results are shown in Table 8 below.

	-Traffic Factor-			
	1.0	4.0	8.0	12.0
multi-path model	23	24	25	28
single-path model	23	24	26	34
SPF simulated X40 series	23 [22,23]	33 [30,36]	44 [39,50]	182 [86,278]

Table 8. Results of 14-Node Simulation I

For the results from the simulator, the 95% confidence interval is shown in square brackets under the average delay. Remember that, apart from the routing, the simulator and analysis are based on the same model of the network. For traffic factor 1.0 we see that the results are identical, and as the traffic level increases the results diverge. The average delay for multiple-path routing is somewhat lower than for single-path routing because at higher traffic levels multiple-path routing allows the load to be spread more evenly. The results for SPF routing are higher because at higher traffic levels the algorithm becomes unstable and causes congestion by switching traffic from one route to another.

3.4 Simulation of Different Routing Schemes

The second series of experiments on the 14-node network repeated the model analysis, but used a fixed delay through each node to approximate processing delay. The simulator was run with its protocols (in particular the line protocol) enabled. The analytic model was run using optimal single-path routing or optimal multiple-path routing. The simulator was run using ARPANET SPF routing or the same routings used by the model. The results for 1.0, 2.0 and 3.0 times the base traffic matrix are shown in Table 9 below:

		-Traffic Factor-	
	1.0	2.0	3.0
multi-path model	28.9	29.2	29.6
single-path model	30.5	30.8	31.1
multi-path simulated	36.7	39.5	46.6
X60 series	[35.8,37.6]	[38.8,40.2]	[45.7,47.4]
single-path simulated	38.5	--	50.3
X50 series	[37.6,39.4]		[49.4,51.2]
SPF simulated	38.7	43.3	57.3
X70 series	[37.9,39.5]	[42.1,44.4]	[52.9,61.6]

Table 9. Results of 14-Node Simulation II

As before, the results for the single-path model are slightly higher than the results for the multi-path model. The difference between this set of results for the model and the previous set is explained by the inclusion of a fixed processing delay in the latter set, and the fact that in the first set of experiments the packet lengths did not include protocol overhead. The average delay for the single-path model is higher than the average delay for the multi-path model, not only because of the load-spreading effect of multi-path routing, but also because the average hop count (and hence aggregate processing delay) is lower for multi-path routing than it is for single-path routing.

The variance between the simulation and model results (i.e., for either single-path or multi-path routing) is caused by two different effects: first, the model includes processing time but not the effect of processes queueing for the processor, while the simulator includes a detailed model of the process structure of a node, taking into account each delay as a packet moves from process to process through the node; second, the model assumes that the limiting resource is transmission capacity on the line, whereas in the simulator it was either buffer space or the number of logical channels on a line. The simulator is more accurate than the analytic model, since it is able to take into account effects which are missing from the model. The difference between multi-path and single-path routing is almost certainly due to the second effect, since the loading of the processor at the given traffic levels is still quite low. This does _not_ explain the difference between (fixed) single-path routing and (adaptive) SPF routing. At traffic factor 1.0 the results are almost identical, indicating that SPF is choosing the same (or equivalent) routing as the fixed single-path routing generated by Gerla's algorithm. As the traffic level increases, SPF increases faster than fixed routing. This is because SPF is intermittently choosing paths which are longer than optimal.

Perhaps the best way to understand the data presented in Tables 8 and 9 above is to look at each configuration's traffic sensitivity level. An exact comparison is not possible, but the reader may consider the difference between traffic factors 1.0 and 4.0 in Table 8 or 3.0 in Table 9.

The lowest change with an increase in traffic is given by the model results, which change by a millisecond or less. All the simulations have a higher sensitivity to traffic; as we mentioned above, this is due to the fact that the limiting resources for the simulator do not appear in the model. In both cases, the delay for SPF routing increases slightly faster than for either type of fixed routing. In other words, a fixed routing which is tailored to the particular network and traffic used is only slightly better than a routing algorithm capable of adapting rapidly to changes in topology or traffic pattern.

3.5 Computer Resources and Limitations

An important issue which we have not mentioned before is whether detailed simulation, such as we have performed for the ARPANET, is really a feasible way of investigating the behavior and performance of network protocols. During our investigation of the SPF routing algorithm we found that the computer resources required were formidable. In addition, our requirement for statistically valid results meant that we had to generate a very large amount of simulation output. In this section we will discuss both the computer problems and the statistical constraints.

There is no doubt that the simulation is not as efficient as it could be. It is written in Simula, runs on a DEC TOPS-20 operating system, and emulates a TOPS-10 operating system. The Simula compiler is very old and does not use modern techniques for generating efficient code. The fact that a Simula program interfaces with the TOPS-20 operating system via a TOPS-10 emulation means that it incurs an extra time penalty. The program itself was written with flexibility rather than efficiency as its chief goal. These effects can be expressed by considering the ratio of computer time to simulated time (for a given network running on a given computer); this ratio depends on the traffic level in the network, as well as the number of nodes. For the 14-node network we used it is approximately 100; that is, a minute of simulated time requires 100 minutes of computer time (on a DEC 2060 system).

The computer system we used also imposes a limit of 256K on the size of the program, including the Simula run-time system and the TOPS-10 emulation package. Again, because Simula does not have a very sophisticated compiler, it is not very efficient in its use of space. We found that the largest network we could simulate was about 20 nodes. This limit would grow almost in proportion to an increase in the size of the network. No single datum requires a lot of space, but the limit is so low because a simulation is made up of numerous small pieces: nodes, lines, hosts, traffic generators and so on.

It would be wrong, however, to conclude that the real limits on the simulator are due to computer limitations. We believe that these limits mask much more serious limits due to the nature of simulation itself. As we explained in [3], special statistical techniques are needed to analyze output from simulation because of the way successive samples are correlated. As network conditions fluctuate, successive packets may all see relatively high or relatively low delay. Our technique for removing this correlation consists of combining samples into batches large enough to eliminate the effect of network fluctuations. The size of batch required therefore depends on the period of fluctuations in the network. In a network with fixed routing, these fluctuations will be due to fluctuations in the traffic load, and queueing effects. For an uncongested network, they will have a relatively short period, and the batch size required for our statistical analysis will therefore be small.

The situation is quite different with adaptive routing. In this case, fluctuations in network conditions will be caused by changes in routing. The period of fluctuations in network conditions will therefore be at least as large as the period of changes in routing. In the ARPANET, the minimum period for routing changes is 10 seconds; 10 seconds of simulated time might require 1000 seconds (or 17 minutes) of computer time. For our experiments with SPF routing, we typically had to simulate hundreds of seconds in order to obtain acceptable confidence limits for our performance estimates.

This problem will only be exacerbated with additional protocols included in the simulator. For example, the congestion control algorithm may operate, in each IMP, in synchrony with the routing algorithm, but with a period several times longer (to avoid resonance effects). In addition, it is possible that the dynamic behavior of the combined routing and congestion control protocols will be very complex. A priori, it is not possible to bound the period of network fluctuations at all.

On the other hand, our experiments with the various 4-node topologies and traffic pattern we have discussed in this section shows that useful results can be obtained from very small networks. Indeed, the operation of the network protocols is often clearer in a small network. Our goal, therefore, should be to build simulations which yield useful data on the network protocols, in particular their sensitivity to parameter choices, rather than simulated networks which are comparable to real networks in size. Finally, the necessity to run the simulation as long as we did is a consequence of our goal of statistically defensible results. It is possible that sophisticated variance reduction techniques, which we have not had time to investigate properly, might provide a way to produce valid results with shorter simulation runs.

4 SIMULATION NETWORK DESCRIPTION

The simulation is divided into many modules. A module is just a collection of data structures and routines to access or modify them. For example, the line protocol, the protocol which ensures that packets are reliably transmitted from IMP to IMP, is implemented by a single module. This module contains a number of tables, and some routines which perform the protocol. Some modules contain code which can run independently; these modules are called processes. IMPs, hosts and lines are all modules which contain a number of processes for simulating the effects of the real IMPs, hosts, and lines.

The structure of the IMP module closely mimics the real IMP program. The module itself contains data structures and routines for implementing routing and buffer allocation; subsidiary modules implement the line protocol and line up/down protocol, and the processes which make up the real IMP program. These processes are: Task, HostIn, HostOut, ModemIn, ModemOut, TimeOut and Background. HostIn is responsible for accepting messages from a host; HostOut delivers messages to a host, ModemIn accepts packets from a line (i.e., transmitted from a neighboring IMP); ModemOut transmits packets onto a line (i.e., to a neighboring IMP); Task hands packets from an input process to the correct output process; TimeOut performs various periodic functions;

Background contains the IMP functions which are run when there is nothing else to do. There are also four processes which run under the control of Background: BackgroundSource, BackgroundSink, BackgroundDummy and BackgroundRerouting.

Each line is simulated by a module and two subsidiary processes: LineInput and LineOutput. LineInput accepts packets from a ModemOut process in the sending IMP; LineOutput delivers them to a ModemIn process in the receiving IMP. Two processes are necessary since any number of packets may be in flight on the line, and, in particular, one packet may still be being delivered at the output while the next is arriving at the input end of the line.

The host is responsible for generating and submitting messages to the network, and accepting delivery from the network. It is implemented by a host module and a number of subsidiary processes: MessageGen generates messages according to a random distribution; MessageOut submits messages one by one to the network (i.e., local IMP); PacketSink accepts packets from the network. Packets and messages are discussed in the next section.

The modules and processes mentioned above are described in the following pages. The line and host are each described in a single section, since they are quite straightforward; the IMP is not, so each process and protocol is described in a separate section.

4.1 Topology and Packet Flow

The topology of the simulated network is defined by the IMPs, hosts and lines in it, and by the connections between them. Each host is connected to a single IMP; each IMP may have many hosts. Each line is connected to two IMPs: one at the sending end and the other at the receiving end; each IMP may be connected to many lines. IMPs which are connected by a line are called neighbors.

The user defines the topology of the simulated network using IMP, HOST and LINE commands. Each time one of these commands is used, the simulator creates the corresponding module and connects it to the other appropriately. When an IMP is created, the user specifies the number of hosts and the number of lines it is connected to. When a line is created, the user specifies which IMPs it is connected to. The simulator connects each IMP to the line, and vice versa. When a host is created, the user specifies which IMP it is connected to. The simulator connects the IMP to the host, and vice versa.

In the ARPANET, messages flow from host to host. Each message is broken up into a number of packets as it enters the source IMP from the source host. This process is called packetizing. The packets are transmitted through the network and eventually all arrive at the destination IMP. There the message is reconstructed before being delivered to the destination host. This process is called reassembly. In the current implementation, the simulation does packetizing but not reassembly.

In the simulation, each message is generated by a host and passed to the IMP that the host is connected to, where it is broken into packets. The maximum packet size is set by the IMP parameter "PACKETLENGTH." Then the packets are passed from IMP to IMP, via the lines that connect them, until they arrive, packet by packet, at the destination IMP. There they are delivered to the destination host. No attempt is made to deliver them in the same order as they entered the network.

As a packet goes through each IMP, host, and line in its path, it is handled by several processes. As an example, suppose that the path from source host to destination host goes through an IMP, a line, and another IMP. The message will be generated by the host process MessageGen, which immediately passes it to the host process MessageOut, for transmission to the IMP.

MessageOut passes it to the IMP Process HostIn, which breaks it into packets and passes it to the IMP Process Task. Task decides where each packet should go next, and passes it to the correct ModemOut process. ModemOut passes each packet to the line process LineInput, for transmission over the line to the next IMP. LineInput accepts the packet from ModemOut, and passes it to the line process LineOutput. LineOutput passes it to the ModemIn process in the _second_ IMP. That ModemIn process passes it to Task, which passes it to HostOut. HostOut delivers the packet to the PacketSink process in the destination host.

If there were store-and-forward IMPs in the path, each packet would be passed from LineOutput, to ModemIn in the IMP, to Task, and to ModemOut. ModemOut would pass each packet to the LineInput process of the line to the next IMP.

4.2 IMP

Each IMP module is created by the command interpreter when it encounters an "IMP" command. When an IMP module is created, it initializes its data structures, then creates its subsidiary modules and processes. The data structures, modules, and processes are described below.

The following four parameters must be passed to the IMP when it is created: the number of IMPs in the simulated network; the number of simplex lines in the simulated network; the number of hosts connected to the IMP which is being created; and the number of lines connected to the IMP. The first two are, of course, the same for every IMP in the network; the second two are specified by the user in the IMP command when the IMP is created. The parameters are used to fix the size of various internal tables in the IMP module.

The IMP has many other parameters that are set by the various IMP subcommands. The IMP module and its subsidiary processes and modules have many local variables which control the way the modules behave. For example, each process has a variable which specifies how long the process executes on the simulated IMP for each packet processed. In general, each IMP subcommand simply sets or changes the value of a particular variable, either in the IMP module itself, or in one of the subsidiary processes or modules. The commands are described in Appendix A.

4.3 Line

Each line module is created by the command interpreter when it encounters a "LINE" command. When a line module is created, it creates its subsidiary processes, LineInput and LineOutput.

LineInput is responsible for accepting packets from a ModemOut process in the sending IMP; LineOutput is responsible for holding packets in flight, then delivering them to a ModemIn process in the receiving IMP.

The line module has three parameters: the number of framing bits per packet, and flags for controlling tracing and debugging output. They are set by the LINE subcommands "FRAMING," "PACKET," and "DEBUG."

The LineInput process has two parameters: the propagation delay from one end of the line to the other, and the speed of the line (expressed as its reciprocal, time per bit). They are set by the LINE subcommands "LAG" and "SPEED." LineInput has a single input queue; a packet is put onto the queue by the ModemOut process which is driving this line, and the ModemOut process then goes to sleep. LineInput immediately removes the packet from the queue, computes the time the first bit will arrive at the other end, then passes the packet to LineOutput. It then waits for the last bit to arrive, and wakes up the ModemOut process. Note that the ModemOut process is asleep while the line is busy; it is impossible for it to put more than one packet on LineInput's queue at a time.

The time it takes to read a packet onto the line is calculated by multiplying the time per bit by the total packet length. The total packet length is the sum of the packet length (from the packet) and the framing (from the line). The packet length is set when the packet is created (i.e., when the message of which it is a part enters the network), and is set to the sum of the number of data bits and the protocol overhead.

The LineOutput process is responsible for holding packets in flight on the line, and delivering them to the receiving IMP's ModemIn process at the proper time. It has two parameters: the line error rate, and the line speed. They are set by the LINE subcommands "ERROR" and "SPEED." It has one data structure: a queue to hold the packets in flight. The LineInput process passes each packet to the LineOutput process as soon as the _first_ bit arrives; the LineOutput process must pass the packet to the receiving IMP as soon as the propagation delay has elapsed. This time is stamped in the packet by LineInput before it puts the packet on LineOutput's transit queue; when LineOutput has finished with one packet, it removes the next packet from the transit queue and examines its arrival time; then, it waits until the arrival time and calls the routing ModemInterface in the ModemIn process for this end of the line, passing the packet and the length of the packet in seconds. ModemInterface reads the packet into the IMP, waits for the _last_ bit of the packet to arrive, then returns. LineOutput then waits for the next packet.

LineOutput is also responsible for simulating line errors. Before handing over the packet, LineOutput calls a random variable routine which returns true or false. The probability of a "true" or "error" result is set to the line error rate multiplied by the total packet length. This approximates the effect of bit errors occurring independently and at random throughout the packet. If the random variable routine returns true, a flag ın the packet is set. This flag will cause the receiving IMP's ModemIn process to discard the packet.

4.4 Host

There may be many hosts per IMP. The number of hosts on an IMP is specified when the IMP is created. Each host is a module which contains some data structures and some procedures. Each host contains one packetSink process which is responsible for accepting packets from the net, an array of messageGen processes which generate messages for each destination, and a messageOut process which transfers the messages to the IMP. A destination is specified by a host and IMP. To send to an IMP's fake host, specify host 0 on that IMP.

The host implements two procedures, Start and Close. Start takes a destination host and IMP number, the average message rate (in messages per unit of simulated time), the average message length and the priority, and starts up a messageGen process to that destination sending at that rate. Repeated calls to Start always produce separate processes and message flows. Close takes the destination IMP number and stops all messageGen processes which are sending to that destination IMP.

The messageGen process is a loop which waits for a randomly determined period of time and then generates another message. The distribution of messages can be either deterministic or negative-exponential and is set by the HOST subcommand "RATE." If the distribution is specified as deterministic, the time between messages is constant; if the distribution is negative-exponential, the time between messages is determined by calling the library routine for a negative-exponential distribution, whose only parameter is the average message rate. The average message rate is specified in the call to Start (which created this process) and is passed as a parameter. The message length is selected from a particular distribution, set by the HOST subcommand "LENGTH." If the distribution is deterministic, the messages have constant length; if the distribution is negative-exponential, the library routine is called. The message priority is passed as a parameter when messageGen is created. When each message is created, the source, destination, priority, and length fields are filled in. The creationTime field is filled in with the current time. Finally, the message is put on messageOut's queue.

MessageOut pulls messages off its input queue, and for each message, calls HostInterface in the IMP's HostIn process. HostInterface puts the message on HostIn's input queue and waits. HostIn copies the message into a packet and passes it to Task. When Task has accepted the packet, Task wakes HostIn, and HostIn wakes HostInterface, which returns to MessageOut.

Because a message is accepted by the IMP packet by packet, the whole message may not have been accepted. HostIn adjusts the message length by subtracting off the number of bits that have been accepted. MessageOut calls HostInterface repeatedly until the message length has been set to zero.

The packetSink process is started by the host when the host is started. The packetSink process has one parameter, the execution time per packet. It is set by the HOST subcommand "SINK." The effect of this parameter is to set the minimum time between successive packets, and hence the maximum rate at which packets can be accepted.

The packetSink process has an input queue. It waits until a packet arrives on the queue, then removes the packet. The total net delay is computed by subtracting the time the packet entered the net (which is given in the packet) from the current time. Note that the delay imposed by the parameter SINK will not affect the network delay, unless the arriving packet must wait behind the previous packet. Then the process waits for its execution time, returns the packet to the Simula garbage collector, and loops to wait for another packet.

4.5 Priority Structure

Every process in the IMP has a separate priority, including each separate modem and host process. The order of priorities, highest first, is: ModemIn, ModemOut, HostIn, HostOut, Timeout, Task, Background. Within ModemIn and ModemOut, modem 1 has the highest priority, followed by modem 2 and so on. Similarly for HostIn and HostOut. Note that the fake host, host 0, is extra. All IMPs have such a host. Thus, for an IMP with two modems and two hosts the priority structure would be:

process	priority
ModemIn(1)	1
ModemIn(2)	2
ModemOut(1)	3
ModemOut(2)	4

HostIn(0)	5
HostIn(1)	6
HostIn(2)	7
HostOut(0)	8
HostOut(1)	9
HostOut(2)	10
Timeout	11
Task	12
Background	13

The process scheduling is preemptive. That is, if a lower priority process is running when a higher priority process starts, then the lower priority process is suspended until the higher priority process has finished. Therefore, the "execution time" of a particular process does not mean that the process takes exactly that much simulated time, but rather that it must be scheduled for that much CPU time. For low priority processes such as Task, the elapsed time may be much greater than the amount of CPU time used by the process.

4.6 Task

There is one task process per IMP. When Task is started by the IMP, the IMP sets its priority; the priority is set lower than TimeOut and higher than Background (see Section 4.5 for details.) The execution time per packet is set by the parameter

"TASK."

The interface to Task is a single procedure, Add. Add takes as parameters the packet to be submitted to Task, and a flag which indicates whether the caller wants to wait for the result. In the simulation, Add is called from ModemIn, HostIn, and Background (rerouting from a down line). HostIn waits for Task to complete, ModemIn does not; Background does not wait for Task to complete, but returns and goes to the next Background process. Each time around the Background loop the rerouting process checks to see if Task has finished. Add puts the packet on one of Task's queues. If the call specifies wait, Add goes to sleep. This affects the caller of Add, not Task.

When Task has processed a packet, it can either accept or reject it. This does not imply anything about whether the packet will eventually be forwarded or delivered out of the IMP. In the present simulation, Task will accept a packet if it can be placed on a queue for HostOut or ModemOut, or if its destination IMP is inaccessible. (This is explained in more detail below.) In the latter case, Task immediately discards the packet. Thus, in just this case, Task accepts and discards the packet. When Task accepts a packet, it checks to see if it entered the IMP from a modem. If it did, and if it is a data packet, Task acknowledges the packet by calling a routine in the line protocol module for the line the packet arrived on. A packet will be acknowledged even it is accepted and discarded because if its destination is inaccessible, it should be cleared from the network. In order to clear a packet from the network, it must be accepted by Task, since otherwise it would not be acknowledged, and would therefore be retransmitted by the previous IMP.

If a packet is rejected, and the process that submitted it is waiting (in Add), the packet is flagged "rejected" and the process is woken up. It is then the responsibility of that process to handle the packet. HostIn holds the packet and goes to sleep -- it will be woken by FastTimeout and will immediately resubmit the packet; in the case of Background, the packet is put back on the reroute queue and is eventually resubmitted. If a packet is rejected and the process that submitted it is <u>not</u> waiting, the packet is discarded.

Task has two queues: a queue for data packets, and a queue for routing update and line up/down protocol packets. The line up/down protocol packets are processed first. If there is a line up/down protocol packet on the queue, Task removes it and calls the routine ProcessLineUpDown in the line protocol module for the line the packet arrived on. Line up/down protocol packets are always accepted. If there is a routing update on the queue, Task removes it and calls the IMP routine ProcessUpdate. Routing updates are always accepted. If there are no routing updates, Task checks its queue of data packets. If there is a packet on the queue, Task removes it and attempts to pass it either to one of the HostOut processes or to one of the ModemOut processes. If it entered the IMP via a modem (rather than a local host), Task calls a routine in the line protocol module to check if the packet is a duplicate; if it is a duplicate it is discarded. If the packet's destination is a host on this IMP, Task attempts to allocate a reassembly buffer. If it can allocate a buffer, it passes the packet to the hostOut process to be output to the host, otherwise the packet is rejected.

If the packet's destination is some other IMP, Task attempts to allocate a store-and-forward buffer. If it cannot allocate a buffer, the packet is rejected. If it can allocate a buffer, it looks up the destination in its IMP's routing table where the output lines are specified. If the output line is specified as zero, the destination IMP is inaccessible; the packet is accepted then discarded. If the output line is not zero, Task attempts to allocate a logical channel on that line. If a channel cannot be allocated, the packet is rejected. If a channel is allocated, the packet is put on the queue of the correct output process. If a data packet is passed to HostOut or ModemOut, or if it is discarded because the destination IMP is inaccessible, these actions must be acknowledged. Task calls the routine AckPacket in the line protocol module for the line on which the packet arrived; this routine flips the receive flag for the channel and calls the ModemOut routine Ack which sets a flag indicating that an acknowledgment should be sent, and wakes up ModemOut if it is asleep. If the packet was a duplicate, Task calls the Ack routine in ModemOut to send a duplicate acknowledgment. Task then loops back to check its queues.

4.7 HostOut

There is one HostOut process for each host on the IMP. When the IMP creates a HostOut process, the IMP sets its priority. Each HostOut process is given a different priority, lower than HostIn and higher than Timeout (see Section 4.5 for details). The execution time per packet is set by the parameter "HOSTOUT."

Each HostOut Process has an input queue and a pointer to a local host. Task puts packets on the input queue. HostOut waits for a packet to be put on the queue, removes it, executes HOSTOUT units of simulated time on the simulated CPU, adds the packet to the local host's input queue, and waits. When the local host's packetSink process has removed the packet, the HostOut process wakes up, returns the packet to the Simula runtime system, and loops back to wait for another packet.

4.8 HostIn

There is one HostIn process for each host on the IMP. HostIn has two responsibilities: it inputs messages from the host, and it breaks messages into packets. When the IMP creates a HostIn process, the IMP sets its priority. Each HostIn process is given a different priority, lower than ModemOut and higher than HostOut (see Section 4.5 for details). The execution time per packet is set by the parameter "HOSTIN."

To transfer a message to the IMP, the host calls the routine HostInterface which is implemented as part of HostIn. HostInterface waits until there is an available buffer into which a packet can copy the message, and until Task has accepted the packet. It does this by putting the message on HostIn's input queue, then going to sleep. It will be woken by HostIn.

HostIn does not necessarily process the whole message at once. Data messages may be longer than a packet, and if so, are accepted packet by packet. The maximum length of a packet is a parameter to HostIn, which is set by the IMP subcommand "PACKETLENGTH." HostIn reads the message length from the message, and when it has finished processing a packet, sets the message length to the number of bits remaining. Callers of HostInterface, therefore, call HostInterface repeatedly until the message length is set to zero, indicating that the entire message has been accepted.

HostIn takes the message off its input queue and attempts to allocate a buffer. If no buffer is available, it goes to sleep. The next time FastTimeout runs, it will notice this and wake up HostIn. HostIn will then check again. This process is repeated until a buffer is available. When a buffer is allocated, the message is copied into the packet and the input line field is cleared so that Task knows this packet came from a local host and does not have to be acknowledged. Then HostIn gives the packet to Task and waits. When Task has processed the packet, it wakes HostIn. If Task rejected the packet, HostIn goes to sleep. It will be woken by FastTimeout and will immediately resubmit the packet to Task. When Task accepts the packet, HostIn wakes HostInterface.

If it takes HostIn 15 seconds to process a packet, it aborts, and reads in and discards the entire message.

4.9 ModemOut

There is a ModemOut process for each output line in the IMP. When the IMP creates a ModemOut process, the IMP sets its priority. Each ModemOut process is given a different priority, lower than ModemIn and higher than HostIn (see Section 4.5 for details). The execution time per packet is set by the parameter "MODEMOUT."

The ModemOut code is in two parts: the first checks to see if there is a packet to be transmitted; the second actually transmits the packet. When ModemOut first starts, it goes to sleep; it returns to this state whenever it has nothing to do. Whenever ModemOut is woken, it checks for a packet to be transmitted. It checks for a line up/down packet, then a routing update, then a retransmission, then a data packet, then an acknowledgment (or null packet). As soon as it finds a packet to be transmitted, it goes to the second section and transmits the packet; if there is nothing to do, it goes back to sleep. When it has finished transmitting a packet, ModemOut again checks to see if there is another packet to be sent, starting over again with line up/down packets. Thus all line up/down packets will be sent before any routing updates, all routing updates before a retransmission, and so on.

When ModemOut goes to sleep, it can be woken by calling the transmit routine to send a packet, by a wakeup to send an acknowledgment, or by a periodic wakeup from the fast timeout routine (to check for retransmissions).

When ModemOut wakes up, it checks for a line up/down packet first. Thus, line up/down protocol packets have the highest priority. There is a single pointer which either points to a line up/down protocol packet or to nothing. If the pointer does point to a packet, the packet is sent and the pointer cleared (i.e., set to nothing). Routing update packets have the next highest priority and they have a separate queue. If there is no line up/down protocol packet waiting to be sent, ModemOut checks the routing update queue. If there is a packet in the queue, it is sent. Line up/down protocol packets and routing update packets are sent even if the line is down (but not in reset state, see Section 4.17, The Line Up/Down Protocol); other packets are not.

If there are no line up/down protocol packets or routing update packets to be sent, and the line is up, ModemOut calls the routine GetRetransmit in the line protocol module for this line and passes as a parameter the retransmission threshold, retransmitWait. This parameter is set by the IMP subcommand "RETRANSMITWAIT." This routine checks to see if there is a packet which was sent more than retransmitWait seconds ago that has not yet been acknowledged. It checks a different logical channel each time it is called. It will only be called if there are no line up/down packets or routing updates to be sent, and then only for a single logical channel. If there is a packet to be retransmitted, it is sent. If not, ModemOut checks its queue of data packets; if there is a packet in the queue it is sent. If not, it checks a flag to see if an acknowledgment should be sent; if so, a null packet is sent. If there is nothing to send, it goes to sleep, otherwise, it goes on to the second section.

When there is a packet to be sent, ModemOut executes MODEMOUT units of simulated CPU time. In the case of a data packet, it copies the receive flags from the corresponding input line into the packet (see "line protocol"), and clears the flag which indicates that there are acknowledgments to be sent. Then it adds the packet to the line's input queue, and waits. When the line has transmitted the packet, it wakes up ModemOut, which then checks to see if the (data) packet was acknowledged while it was being transmitted. If it was, the channel is freed and the flag is cleared. Then the output process loops back to the first section to check for more packets.

4.10 ModemIn

There is one ModemIn process for each input line in the IMP. ModemIn contains an input queue and implements a routine, ModemInterface, which is called by the line to pass a packet to the process.

When the IMP creates a ModemIn process, the IMP sets its priority. The ModemIn processes have the highest priority, above the ModemOut processes (see Section 4.5 for details). Each ModemIn process is given a different priority. The execution time per packet is set by the parameter "MODEMIN."

Whe.ı the line calls ModemInterface, it passes a pointer to the packet from the sending IMP. First, ModemIn checks the line protocol state to see if the line is in "reset" state (see Section 4.17, The Line Up/Down Protocol). No packets are accepted from the line if it is in reset state; ModemInterface simply returns, without copying the packet into the IMP, and the packet is lost. Next, ModemInterface checks a flag to see if the modem is ready to receive another packet. If not, ModemInterface again returns without copying the packet into the IMP. Last, ModemInterface checks to see if there is a buffer ready for the packet. If there is not, the packet is discarded and an attempt is made to allocate a buffer for the next packet; otherwise, ModemInterface copies the packet into the buffer and puts it onto ModemIn's input queue.

Immediately after (the last bit of) a packet has arrived, ModemIn cannot accept another packet until a short interval has passed. This is because the hardware must be reset, not because of any processing on the first packet. The length of this interval is a parameter to ModemIn, set by the IMP subcommand "LATENCY."

When ModemIn removes a packet from its input queue, it first executes some time on the simulated CPU (equal to the hardware latency), then it clears a flag to indicate that it is ready to receive another packet. Next, ModemIn checks to see if the packet suffered a line error; if so, the packet is discarded. Otherwise, ModemIn attempts to allocate a buffer ready for the next packet. There is one exception: if the packet is a "Hello" or "IHY" (explained in Section 4.17, The Line Up/Down Protocol), the packet does not require a buffer, and the current buffer is immediately recycled for the next packet. If the packet is a routing update it can take the buffer even if a buffer cannot be allocated for the next packet (which will therefore be lost). If the packet is a null data packet it can use the last buffer since it will immediately be processed by ModemIn and discarded; otherwise, if no buffer is available, the packet is discarded and the current buffer is recycled for the next input. If another buffer is available, or not required, the packet will be processed and handed to Task. If the packet is a data packet, it will have a line protocol header; ModemIn calls a routine in the LineModule for this line to process the acknowledgments in the header (for more details see Section 4.16, The Line Protocol). If a data packet has its discard bit set, it is discarded and acknowledged (this is explained in Section 4.9, ModemOut). If the packet is a null packet (i.e., a zero length data packet), it is discarded; if not, it is submitted to Task and the process loops.

4.11 Timeout Process

Each IMP has one Timeout process. Timeout runs at regular intervals, and is responsible for calling various routines which need to run at regular intervals.

When the IMP creates the Timeout process, the IMP sets its priority between HostIn and Task (see Section 4.5 for details). The interval between runs of Timeout is set by the parameter "TIMEOUT." In the actual network, this is always 25.6 msec. The exact time of each wake-up is controlled by the parameters "RATE" and "FASTOFFSET," which are described in Section 4.18, IMP Time.

Timeout has a counter which counts up to 25 and is then reset to 0. Normally when the process wakes up it calls the routine FastTimeout. On every 25th execution, it calls SlowTimeout and then FastTimeout. The execution time of FastTimeout and SlowTimeout are set by the parameters "FASTCPUTIME" and "SLOWCPUTIME," respectively.

FastTimeout calls the line up/down protocol routine "Tick" every "LINEUDPERIOD" ticks. The parameter LINEUDPERIOD may be set differently for each line. Tick is described in Section 4.17, The Line Up/Down Protocol.

FastTimeout calls the IMP routine "AverageDelay" every "DELAYAVGPERIOD" ticks, after an initial quiescent period of "DELAYOFFSET" seconds. AverageDelay is described in Section 4.14, Routing Update Protocol.

Each time FastTimeout runs, it checks each line to see if it is idle (i.e., if the corresponding ModemOut process is asleep) and, if so, wakes the line to check whether a retransmission or a null packet should be sent. It also checks to see if any HostIn process is asleep (waiting for a buffer) and, if so, wakes it up.

SlowTimeout calls the IMP routine "AgeTick" every "UPDATEAGINGPERIOD" ticks. AgeTick is described in Section 4.14, Routing Update Protocol.

The IMP routine TimerTick decrements the update timers on each line. TimerTick is described in Section 4.14, Routing Update Protocol. It is called by FastTimeout for lines that are up, and SlowTimeout for lines that are down, each time either timeout routine runs.

4.12 Background

There is one background process per IMP. When the IMP creates the background process, the IMP sets its priority. Background has the lowest priority, below Task.

The Background process is a simple loop, with a number of subsidiary processes, also running at Background priority. Only one of these processes (or Background itself) is running at a time. Background transfers control to one of its subsidiaries, then goes to sleep. The subsidiary does some processing (or finds it has nothing to do) then wakes Background, which wakes the next subsidiary and goes back to sleep.

The subsidiary processes are: Source, for generating fake host traffic; Sink, for accepting fake host traffic; Dummy, for using up time at Background level; and Reroute, for rerouting packets from down lines.

The parameters of Source are the destination IMP and destination host for the message traffic, the message length and priority, the number of fast ticks between messages, and the execution time per packet. They are set by the IMP subcommands "DESTIMP," "DESTHOST," "LENGTH," "PRIORITY," "MSGRATE," and "SOURCE." If the destination IMP is specified as zero, the generator is turned off; otherwise, Source counts fast ticks until another message should be sent. Then the message is constructed and HostInterface is called. HostInterface immediately returns to Source, which returns to Background. Each time Source is woken it checks to see if the message has been processed. When it has been accepted by HostIn, Source then goes back to counting fast ticks until the next message is due to be sent.

When Sink is woken by Background, it checks to see if HostOut has put a packet on its input queue; if so, the packet is removed. The execution time per packet is given by the IMP subcommand "SINK." Then Sink returns to Background.

Dummy is merely used to represent parts of the "real"

Background loop, such as the end-to-end protocol, which have not yet been implemented. It executes some CPU time and returns to Background. The execution time per invocation is given by the IMP subcommand "DUMMY."

When a line goes down, data packets which are queued for output on the line, or which have been sent but not acknowledged, are placed on a special queue in the IMP, the reroute queue. Reroute waits for a packet to be placed on the queue, then removes it. It executes some time on the simulated CPU, then submits the packet to Task (which will reroute it on a different line, or if the destination IMP is now inaccessible, discard it). The execution time per packet is given by the IMP subcommand "REROUTE." If Task rejects the packet, it is put back on the end of the reroute queue; then Reroute returns to Background.

4.13 Routing and Forwarding

Forwarding in the present simulation is simple. The forwarding table found in the IMP tells Task which output line to use to send a packet to a given destination. If an IMP is inaccessible, its entry will be zero. Routing is the method used to build that table.

Each IMP has a table which gives the delay out every (accessible) line in the network. The table is kept up-to-date by the routing update protocol (see Section 4.14). Packets called routing updates are used by the routing update protocol to distribute information on line delays from each IMP to every other IMP. When an update arrives at an IMP, Task passes the update to the IMP routine ProcessUpdate. If the update has not been seen before, its information is copied into the IMP's delay tables and the routine UpdateRouting is called. This routine uses the shortest path first (SPF) algorithm to compute the shortest (minimum delay) path to every other IMP, and at the same time fills in the forwarding table.

4.14 Routing Update Protocol

The routing update protocol ensures that the delay table in every IMP is kept up-to-date, and that duplicate, old, or out-of-date information is ignored. Delay information is carried in routing updates. Every update contains a serial number, an age, and a retry bit. The serial number is a mod 64 number which is unique for each new update. The age is a 3-bit number which is set to 7 when the update is created, and counted down at regular intervals. When the age field reaches zero the update is neither discarded nor broadcast to other IMPs.

The protocol guarantees that the update will be successfully transmitted from IMP to IMP. When a new update is received by an IMP, it is echoed (or retransmitted) back to the IMP that sent it. That IMP then knows that the other IMP received it. Each IMP has a set of timers for each line. Each timer measures the time since an update for a particular IMP was transmitted out the line. When the update is echoed by the neighboring IMP, the timer is cleared. Each timer is decremented at regular intervals. If a timer expires, the update for that IMP is sent out on that line with the retry bit set. The retry bit is simply a request for an echo. It forces the IMP that receives the update to retransmit it to the IMP that sent it, whether or not the receiving IMP has seen the update before.

Each IMP must keep the age and serial number of the latest update from each other IMP. When a new update arrives it supersedes the current update if the current update has an age of zero, or if the new update has a later serial number (mod 64) than the current update. If it does, the age, serial number, and delay information are copied from the new update into the IMP's tables. If the current update has a serial number of C, and the new update has a serial number of N, then the new update is later if:

N-C < 32 (mod 64)

The IMP routine TimerTick decrements all non-zero timers; should the timer for any IMP and line be decremented to zero and the update for that IMP has non-zero age, it is transmitted over that line with the retry bit set and the timer reset to 3. TimerTick is called every Fast Timeout for lines that are up, and every Slow Timeout for lines that are down.

The IMP routine AgeTick decrements the age of every update. No action is taken when the age is decremented to zero, but updates whose age is zero are never retransmitted by TimerTick. This ensures that after a certain period of time, an update will stop circulating through the network. AgeTick is called by Slow Timeout after a certain number of slow ticks; the number of ticks between calls is set by the IMP parameter "UPDATEAGINGPERIOD."

4.15 Delay Measurement

There are two separate measurements of the time it takes a packet to go through a node: one for tracing, the other for the delay used in the routing computation. For trace purposes, the delay is the length of time expended since the last bit entered the IMP (from a modem or local host) until the last bit leaves the IMP (to a modem or local host). For routing, the delay is the length of time expended since the last bit entered a modem <u>or</u> the last (i.e., and successful) time HostIn passed the packet to Task, until the time the last bit arrives at a neighboring IMP. Packets which leave the IMP to go to a local host are not counted in the delay for routing purposes.

Here we will consider only the second kind of delay measurement. The entry time for a packet is put by ModemIn or HostIn into the delayStamp field in the packet. When the line protocol (q.v.) gets an acknowledgment for the packet, the delay is computed from the information in the packet, and the routine TallyDel is called. This routine increments the packet counter totalPackets, and adds the delay into the delay accumulator totalDel.

Periodically, the fast timeout routine calls the IMP routine, AverageDelay, which is responsible for maintaining the delay threshold, calling the ModemOut routine which averages delay, and deciding whether to send out an update. The number of fast ticks between calls is set by the IMP parameter "DELAYAVGPERIOD."

First, AverageDelay reduces the delay threshold by a fixed amount (set by the IMP subcommand "DECAY"); then, for each line, it calls the ModemOut routine AverageDelay which computes the average delay, stores the result in avgDel, and returns the difference between avgDel and the current delay del. If no packets have been transmitted out the line during the last period, the "average delay" is taken to be the propagation delay of the line. If the difference is less than the threshold for any line, an update must be sent out. First, the threshold is reset to its initial value; then, the just computed average delay is copied into the current delay variable in each output process, and into the IMP's delay table. Finally, the update serial number is incremented, the update age is reset, and the update is sent out on each line.

In order to model the fact that that IMP has a finite word size, and hence a finite precision, the simulation rounds the results of delay averaging in three ways. The delays themselves are rounded to units specified by the parameter "DELAYUNITS"; the average delay is rounded to units specified by the parameter "AVGUNITS"; and the computation of the average is limited to the number of bits specified by the parameter "WORDSIZE." In order to gauge the effect of these roundings, the simulation does the calculation in parallel, producing both rounded and unrounded results. Only the rounded result is distributed in routing updates, both are available in debugging output.

There is a slight discrepancy between the way the simulation works, described above, and the way the IMP actually computes average line delays. In the IMP, separate lines are averaged on successive fast ticks. Suppose the IMP has n lines, and is doing delay averaging every 10 seconds, or K fast ticks, the first line is averaged on tick K-n, and the n'th line is averaged on tick K-1. The decision of whether to send an update is made on the K'th tick, at the 10-second mark.

4.16 The Line Protocol

There is one line protocol module for each line in the IMP; it is created when the line is created. The line protocol module consists of a set of functions which share a common data structure, it is not a separate process. The purpose of the protocol is to ensure that one and only one copy of a data packet is passed from IMP to IMP. The routing update protocol and the line up/down protocol do not use this mechanism.

The line protocol is based on having a number of separate channels for transmitting and acknowledging packets. The number of channels is a parameter to the line protocol module. This parameter can be set by the LINE subcommand "NUMCH." Packets are identified with a channel number, and succeeding packets on a given channel are distinguished by having different values of a one-bit flag, called the channel bit. Only one packet at a time can be sent on each channel. In the sender, the line protocol module keeps track of the value of the channel bit last sent on each channel in a set of flags called send flags; in the receiver, the module keeps track of the channel bit last received on each channel in a set of flags called receive flags.

To acknowledge a packet, the receiving IMP reports to the sending IMP the values of its receive flags. When the receive flag in the receiver equals the send flag in the sender, the packet on that channel has been received as well as sent; it is thus acknowledged. For efficiency, the receive flags are carried from receiver to sender in the header of data packets travelling in that direction. If a packet has been sent but not acknowledged after 125 msec, it may be retransmitted. Duplicate packets will be detected since their channel bit will be the same as the receive flag for their channel in the receiving IMP. When a packet has been acknowledged, the channel is then available for another packet; when it is allocated to the channel it is given the opposite value of the channel bit (stored in the send flags) so that it will be clear that it is the next packet and not a retransmission.

For example, suppose the receiving IMP receives a packet on a particular channel with the channel bit equal to 0. It sets its receive flag to 0, and sends this off to the sender in the header of a packet going in that direction. There are two cases to consider: either the sender gets the acknowledgment, flips its send flag, and sends the next packet on that channel; or the acknowledgment does not arrive and the sender retransmits a duplicate of the first packet. The receiver can distinguish between these two cases since a duplicate will have a channel bit of 0 (equal to the channel bit of the original packet, and hence the receive flag), while the next packet will have a channel bit of 1 (different from the receive flag).

The line protocol module uses 3 data structures to implement the line protocol: the send flags, the receive flags, and an array containing packets which have been sent but not acknowledged. The packet contains a copy of the sender's receive flags, the channel number, and the channel bit, which is a copy of the send flag for the channel used by the packet.

The line protocol module implements 8 separate routines to perform various protocol functions: ProcessLPH, for processing acknowledgments in an incoming packet; FreeChannel used by ProcessLPH to free acknowledged packets; Duplicate, which tests whether a packet is a duplicate (i.e., retransmission) or not; AcceptPacket, used when Task accepts a packet; SelectChannel, which finds a free channel, if any, for an outgoing packet; GrabChannel, which allocates the channel found by SelectChannel; PiggyBackAcks, which copies the receive flags into an outgoing packet; and GetRetransmit, which checks whether a given channel has an unacknowledged packet which should be retransmitted. These functions are discussed in more detail below.

ProcessLPH is called by ModemIn when a data packet is accepted by ModemIn. Every data packet has a header, the line protocol header, or LPH, which carries a copy of the sender's receive flags. ProcessLPH takes these receive flags and compares them with its send flags. Any matches indicate that the packet sent on that channel has been received and accepted by the other IMP. For each channel where there is a match, ProcessLPH calls FreeChannel which clears the entry in the unacknowledged packet array, and flips the send flag so that the next packet sent on that channel gets the opposite value to the last one.

When a data packet is being processed by Task, it first checks to see if the packet is a duplicate, by calling Duplicate. This routine compares the packet's channel bit to the receive flag for the packet's channel, if they are the same, a packet with that value of the channel bit was accepted last (on that channel) and the packet is a duplicate. Task calls AcceptPacket when it accepts a data packet. This routine sets the receive flag on the packet's channel to equal the packet's channel bit (so, for example, a retransmission of the same packet will be detected as a duplicate), and signals ModemOut on this line that an acknowledgment should be sent. When a packet in the opposite direction carries the receive flags from this IMP to the other IMP, ProcessLPH will recognize that this packet has been received and accepted.

SelectChannel simply searches the array of unacknowledged packets. An empty entry indicates that the corresponding channel is free, and can be used for an outgoing packet. GrabChannel takes a packet, and the channel number returned by SelectChannel, sets the entry in the array of unacknowledged packets to point to this packet (even though it has not yet been sent) to indicate that the channel is now taken; sets the channel number in the packet; and copies the send flag for this channel into the packet's channel bit. Of course, it is possible that there will be no channel available. In this case, SelectChannel returns zero (not a legal channel number) and Task rejects the packet.

Just before a data packet is put on the line by ModemOut, it calls PiggyBackAcks. This routine copies the receive flags into the line protocol header of the outgoing packet, thus acknowledging all the packets which have been received and accepted up to that point. ModemOut will also call PiggyBackAcks to fill in the header of a null packet; this is sent if there is no other traffic to carry the acknowledgments.

Before ModemOut sends ordinary data traffic, it checks one channel to see if there is a packet on this channel which should be retransmitted. It does this by calling GetRetransmit, which must distinguish between four cases: the channel is free (the entry in the array of unacknowledged packets is empty); the channel is in use but the packet has not yet been sent (the packet has not been stamped with its exit time); the packet has been sent but not long enough ago; and the packet has been sent more than a certain length of time ago (a parameter in ModemOut which can be set by the IMP subcommand "RETRANSMITWAIT"). Only in the last case is a packet returned for retransmission. In order to avoid clogging the line with retransmissions, ModemOut calls GetRetransmit at most once each time it runs, and GetRetransmit only checks a single channel.

In order to limit the number of retransmissions which are sent, ModemOut keeps track of the number of times it has sent a packet. When it is about to retransmit a packet after 32 or more unsuccessful attempts, it sets a bit (called the discard bit) in the packet. If the ModemIn routine at the other end of the line successfully receives the packet, it must discard and acknowledge the packet. This is a way of verifying that the line and line protocol are working even though the receiving IMP may be too congested to accept (and acknowledge the packet). If ModemOut is about to retransmit a packet after 64 unsuccessful attempts, it concludes that the line is bad and brings the line down. This packet and any others queued for the line are rerouted.

4.17 The Line Up/Down Protocol

The line up/down protocol is a protocol which tests a line by sending special packets on the line from one IMP to the other, bringing the line down if too many of these packets are missed, and bringing the line up again if enough are received correctly. The motivation for the protocol, and the details of the protocol itself, are described in BBN Reports 3803 and 3940.

In the simulation, the protocol is implemented by a special module, LineUpDownProtocol. There is an instance of this module for each line in the IMP. It contains variables for the protocol state, and implements two routines: Tick, and ProcessLineUpDown. Tick is called by SlowTimeout; the number of slow ticks between calls is set by the IMP parameter "LINEUDPERIOD." Tick performs the various periodic protocol functions described below. ProcessLineUpDown is called by Task when it gets a line up/down packet. Protocol actions (changes of state, sending a packet, bringing the line up or down) can thus be triggered either by an external event (the arrival of a line up/down packet) or by an internal event (a call to Tick from SlowTimeout).

There is a requirement, imposed by the routing update protocol, that a line should take at least 60 seconds to come up. Therefore, the line is not brought up when the line protocol enters Master-Up state, but at every tick the routine MyTick checks to see if the protocol is in Master-Up state <u>and</u> 60 seconds have elapsed since the line went into Reset state; if so, the line is brought up. It is only then that the Master starts sending Hello-Up's to bring the slave from Slave-Down to Slave-Up.

The protocol has five states: Reset, Slave Down, Slave Up, Master Down, and Master Up. Up and down refer to the state of the line as well as the protocol state, although the state of the line is also stored in the variable lineState in ModemOut, and can be deduced from the delay tables in the IMP. Master and Slave refer to the fact that the protocol is asymmetric, the higher numbered IMP on the line being the Master and the lower numbered the Slave. The Reset state is used simply to reset the protocol and ensure that both IMPs declare the line down. Since the line is down during the Reset state, it might be more correctly called Reset Down.

The protocol modules on each end of the line exchange packets called Hello's and IHY's (from "I Heard You"). These packets have three fields: the (sending) IMP number, the state of the line, and whether the packet is a Hello or an IHY. They may be named from the last two fields: Hello-Down, Hello-Up, IHY-Down or IHY-Up. In either of the slave states, the IMP only sends IHY's; in either of the master states the IMP only sends Hello's; in Reset state the IMP only sends Hello-Down's.

The protocol actions and events which cause state changes are given below. "Tick" refers to a call to Tick from SlowTimeout; "Receive" refers to processing a packet by ProcessLineUpDown; "Miss" refers to Tick noticing that a certain sort of packet has not been received since the last call to Tick. The parameters K, N and NUP are set simultaneously by the LINE subcommand "LUD."

Protocol Actions

RESET:	Send a Hello-Down each tick Ignore input for K+1 ticks
SLAVE:	Send IHY for every hello
MASTER:	Send Hello every tick

State Changes

RESET --> MASTER DOWN:	Automatic transition after K+1 ticks

MASTER DOWN --> SLAVE DOWN:	Receive Hello with larger IMP number than self
SLAVE DOWN --> RESET:	Miss K consecutive Hello's
MASTER DOWN --> RESET:	Miss K consecutive IHY's
SLAVE DOWN --> SLAVE UP:	Receive Hello-up
MASTER DOWN --> MASTER UP:	Receive NUP consecutive IHY's
SLAVE UP --> RESET:	Receive Hello-Down _or_ Miss K consecutive Hello's
MASTER UP --> RESET:	Receive Hello-Down _or_ Miss K out of N IHY's

The line may also be brought down (from either Slave-Up or Master-Up to Reset) if ModemOut discovers that a packet has been unsuccessfully transmitted 64 times.

4.18 IMP Time

Processes in the IMP take up time in one of two ways: they run on the simulated CPU, or they wait for a specified interval to elapse. The latter is used to simulate the ticking of the clock which is used to wake up Timeout. Of course, a process may also go to sleep some other way, such as waiting for a packet to arrive on a queue, in which case it may be later when it is woken.

We must simulate the situation where the IMP clock runs either fast or slow, and may not be synchronized with either "real" simulated time, or the clocks of other IMPs.

First, the ratio of IMP time to simulated time is set by the parameter "RATE." If this parameter is R, and Timeout is set to run every T seconds, then Timeout will in fact run every R * T seconds, and a process whose execution time is S seconds will execute for R * S seconds on the simulated CPU.

Second, each IMP starts its clock at a different time, specified by the parameter "FASTOFFSET." Suppose this parameter is set to 0, and assume the simple case where R = 1 (the IMP clock is neither fast nor slow). Then Timeout will run at 0 + T, 0 + 2*T, 0 + 3*T, and so on. FASTOFFSET should be set to a value between 0.0 and T.

Third, each IMP does not start its first delay averaging interval until a particular time, specified by the parameter "DELAYOFFSET." Since delay averaging is controlled by Timeout, DELAYOFFSET is effectively rounded up to the next multiple of R * T. It should be set to a value between 0.0 and R * T * (the value of the parameter "DELAYAVGPERIOD").

This allows us to control the relationship between events in different IMPs. If we want all routing updates to happen at the same time, we can set RATE to 1.0 and DELAYOFFSET to 0.0. If we want updates to happen at different times but in a synchronized way, we can set RATE to 1.0 and DELAYOFFSET to particular values for each IMP. Finally, if we want to approximate the asynchronous behavior of real IMPs, we can set both RATE and DELAYOFFSET to particular values for each IMP.

4.19 Buffer Management

In the IMP, allocated buffers are divided into 3 classes: reassembly, store-and-forward, and uncounted. Packets entering from the host are allocated as reassembly. Packets entering from a modem are allocated as uncounted. Task reallocates (or discards) packets on the basis of their destination: packets for the local host are reallocated as reassembly; packets to be forwarded to another IMP are reallocated as store-and-forward. For each buffer type, the IMP maintains a counter of the number of buffers, and a maximum count. A buffer can be allocated from a given type if the counter is less than maximum, and if certain other restrictions are met. A store-and-forward buffer can be allocated as long as there are more than three free (i.e., unallocated) buffers left. When a reassembly buffer is allocated, the request specifies a count parameter. If there are count reassembly buffers available, and if, after allocating that many buffers, there will be more than three free buffers left, then a reassembly buffer can be allocated. An uncounted buffer can be allocated as long as there is a free buffer available. There are no other constraints. It is also legal to allocate a store-and-forward buffer for an output line which has no other buffer allocated, even if this would cause the count to exceed the nominal maximum.

The buffer limits are set as a function of the total number of buffers available and the number of lines in the IMP. If the number of lines is just one, then the calculation is done as though the IMP had two lines. Let the (adjusted) number of lines be M, and the total number of buffers be NUMBUFFERS. Then the maximum number of store-and-forward buffers (SFMAX) is set to 6 + 2 * M, and the minimum number of store-and-forward buffers (SFMIN) is set to 3 * M. If SFMIN is divisible by 8, it is decreased by 1. Next, the maximum number of reassembly buffers (MAXR) is set to NUMBUFFERS - SFMIN. Finally, the maximum number of uncounted buffers is set to NUMBUFFERS.

There is an interesting feature of the scheme as described. As the number of buffers in the IMP gets larger and larger, the number of buffers allowed for store-and-forward does not change. It is a function only of the number of lines in the IMP. In order to allow simulation of an IMP with an unlimited number of buffers, the command "NOBUFFERLIMIT" is provided. This command sets all buffer limits to a very large number.

5 MULTI-PATH ROUTING: MODEL AND OPTIMIZER

The multipath routing analytic model and optimizer (hereafter referred to as MRMAIN) is a computer program that can be used to study the behavior of packet switched networks that employ multi-path routing in their internal operation. It is capable of modelling multiple priorities of traffic, and can estimate network performance for different line parameters, node parameters, and packet length distributions. It has a straightforward user interface which allows the experimenter to modify the network and then test it for improved performance. MRMAIN can predict end-to-end delay if given traffic and routing information. It can also create feasible routings if none is specified, or find globally optimal routes starting from any feasible initial routing. MRMAIN is implemented as a collection of Simula (see Simula Language Handbook, [8]) subroutines and procedures.

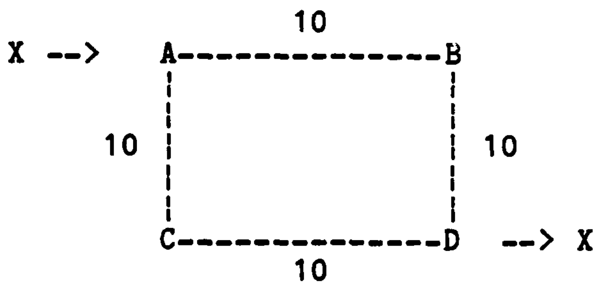
5.1 Analytic Model

MRMAIN uses an analytic model based on an M/G/1 n-priority non-preemptive queueing system (Kleinrock [4]) to predict network behavior. The average end-to-end network delay is computed from the delay of packets in each flow. This is the average delay seen by packets from the time they are accepted at the first IMP in the path to the time they are ready to be forwarded from the last IMP in the path to the destination device (host, TAC, etc.). Network delay is reported separately for each priority level. Average hop count and combined network delay (weighted average of all priority levels) are also reported.

As part of its input, MRMAIN accepts traffic data for the network being analyzed. Packet length distributions for flows in the network can be either exponential or deterministic. If an exponential distribution is chosen, the length of each packet in all network flows is drawn from an exponential distribution with a mean that is specified by the user. If a deterministic distribution is chosen, each flow must specify both the rate and the length of packets in that flow. These two types of packet length distributions allow us to study a wide range of interesting traffic patterns.

Multi-path routing is used in distributing flows throughout the network. MRMAIN routes traffic only on the basis of its destination (commodity based routing). Unlike single-path routing where individual flows are constrained to follow only one path at a time from the source to the destination, multi-path routing allows multiple paths. This makes it possible to route traffic in ways that make better use of the available system resources.

In some cases, it may not be possible to route traffic according to any single-path routing; for example, if a flow to be routed was larger than the largest capacity path connecting two nodes. Consider the following example:



The nodes are labelled A, B, C, D, and the lines connecting them are AB, BD, CD and AC. The capacity of each line is 10, while the flow to be sent from node A to node D is X. If X >= 10, then single-path routing could not handle this flow. However, if X = 9.5, then single-path routing could handle this flow by sending traffic along a route such as ABD. In this case, we would expect lines AB and BD to have large queueing delays because they would be heavily loaded. It would be difficult or impossible to send additional traffic from B to D. If multi-path routing had been used, we could send half along ABD and the other half along ACD. The load on all lines in the network would then be uniform at 4.75, and queueing on all lines would be substantially reduced.

Note that additional flow from B to D could now be routed without difficulty. MRMAIN can handle all these cases: if any line is overloaded (utilization >= 1.0) it will report an "infinite" network delay. When the delay is not infinite, it can report the delay for each priority of packet (currently only 2 priorities are supported) in addition to line utilizations, bit rates, and packet rates. Routing can be analyzed by printing the routing tables for any node, or by printing a route summary for each source-destination flow showing all routes taken by packets in each flow, the percent of packets taking each path, and the delay along each path. This is particularly useful when used in conjunction with the routing analysis produced by the simulator.

Nodes in MRMAIN are modelled as Simula objects. Each node in the network keeps track of the percentage of the flow it should send on each of its lines for each destination. This means that multi-path routing tables require (at least) N times as much memory as single-path routing tables, where N is the average connectivity of the network. In addition to routing tables, statistics are kept separately for high priority and low priority packets on each line. Packet rates through the nodes are also stored separately so that the rates through each part of the IMP (ModemIn, ModemOut, HostIn, HostOut, FakeIn, FakeOut, Task) can be reported. The delay model for the node consists of summing the delays through each process of the node as appropriate; queueing for resources in the node is not explicitly modelled (but can be taken into account by increasing the average time for each subtask in the IMP as appropriate). As an example, store and forward delay in a node is just the sum of a ModemIn, Task, and ModemOut delay. Similarly, host input delay is the sum of a HostIn, Task, and either a ModemOut or HostOut delay (depending on whether the packet is destined for a local host or a host on a remote IMP).

MRMAIN can simultaneously model traffic for many different "times of day." Facilities exist to select the peak time of day, and to run analysis only on the peak data. In addition, it is possible to analyze the data for any time of day separately and conveniently without having to constantly stop, restart, and reload MRMAIN with the new data each time. This "multiple time of day" feature can be disabled (i.e., we set the parameters of MRMAIN so that only a single time of day can be modelled) to leave more memory available for modelling large networks.

There are a number of other MRMAIN features that make it useful as a tool for modelling network behavior. Routing tables can be modified interactively and checked for validity either automatically or manually. It is easy to dump the state of the MRMAIN program and resume it again at a later time. This is useful for large or complex network models that have taken large amounts of CPU or elapsed time to produce. Extensive facilities exist to trace and debug the operation of the internal algorithms. In addition, the user can enable or disable switches which cause warning messages to be printed or suppressed. These messages inform the user of conditions that may be important in the correct operation of the internal algorithms. Appendix B contains a detailed description of MRMAIN commands and sample output.

5.2 Routing Optimizer

In additiun to modelling a network to estimate its performance, MRMAIN can create a feasible multi-path routing when none exists, or can modify an existing routing to find optimal multi-path routes for a given topology and traffic load. The performance metric chosen is the k-th power average end-to-end network delay. The delay for each link flow in the network is raised to the k-th power, weighted according to the magnitude of the total network flow, and summed over all links in the network. The k-th power average network delay is reported as the k-th root of this weighted sum. The formula describing the average network delay (T) is:

$$T = \left[\sum_{i=1}^{M} \frac{\lambda_i}{\gamma} (T_i)^k \right]^{1/k}$$

Lambda is the average traffic rate on each link, and gamma is the total source-destination traffic rate. We use the k-th power function rather than the simpler case where k=1 so that we can better investigate performance of the delay metric. Values of k > 1 cause the performance metric to be more sensitive to variance in the delay between individual network flows. This may prove useful as we will show when we discuss convergence criteria. The Simula implementation of the k-th power performance metric has been made more efficient for the common case k=1 so that the runtime cost of this extra complexity is minimized.

Flow Deviation

The Flow Deviation (FD) algorithm described by Kleinrock [4] and Gerla [5] is used to generate optimal multi-path routes. The FD algorithm can be described briefly as follows. Starting from an initially feasible flow, the FD algorithm first calculates a "shortest-path flow" based on the k-th power performance function described earlier. This represents the cheapest or best way to add new traffic to the network. Using the initial flow and the shortest-path flow, FD computes a new flow which is guaranteed to improve overall network performance as defined by the k-th power function. This new flow then becomes the initial flow for the next iteration, and the process is repeated until the flow converges to an optimum for the given topology and traffic. The problem of finding an initial feasible flow is also described by Kleinrock [4] and Gerla [5], and implemented in our code. For a detailed discussion of how this algorithm works, why FD is guaranteed to converge to a solution, and why the solution will be globally optimal, refer to [4] and [5].

Note that we have used the k-th power function to evaluate network performance. For the case k=1, this reduces to the weighted average network delay described by Kleinrock and Gerla. Values of k > 1 cause flows with a large difference from the mean delay (as computed for k = 1) to have a larger impact on the performance function T. If our objective is reducing the variance of delay between the flows in a network, values of k > 1 will cause the FD algorithm to take this into account automatically. In some cases this may also help FD to converge more quickly to the optimal flow. Although we have not yet had the chance to make extensive use of k > 1 values, this capability is included in MRMAIN.

The FD algorithm produces a set of flows that will minimize the network performance function for a given network topology and traffic load. From this optimal _flow_, we can construct the corresponding optimal routing. In order to construct the optimal routing tables when the FD algorithm has completed, we must store a great deal of information about the flows in the network. The easiest way of computing the optimal routing involves storing individual network flows on each link. Thus, for each link, we separately store the amount of flow destined for each node in the network. For an N node network, this requires that we store N values for each link. This would not be terribly inefficient except that we are also storing information for each of two different priority levels and many different times of day! Limiting ourselves to modelling only one time of day helps this problem, but the size of the tables needed for each link is still quite large. This limits us somewhat in the size of networks we can model.

As an alternative to storing flows for each destination on each link, we attempted to implement an algorithm that recomputes a new routing at each iteration of the flow deviation algorithm. Although this has the disadvantage of requiring more CPU time for computing routes at each iteration, it allowed us to model larger networks than we could model using the simpler (and more memory intensive) method described above.

Unfortunately, recomputing routes at each iteration created significant problems with numerical roundoff errors. For large numbers of iterations and with highly loaded networks, it was not possible to create a routing table that resulted in *exactly* the same flows as produced by the flow deviation algorithm. A very slight difference in flow on a heavily loaded link results in dramatic differences in queueing delay on the line in question. This can significantly affect the network performance function. Thus, after a number of iterations, the flow produced by redistributing traffic into the network based on the calculated routing table would be significantly different from the flow calculated by the FD algorithm!

To reduce the severity of this problem, we modified the algorithm in the following way. Each iteration of FD produces a new flow which can then be used to produce a new routing (as in the published algorithm). Instead of proceeding immediately to the next iteration of FD, we first recompute the flow based on the *new routing*. Note that the new flow that results will be only slightly different from that created by FD originally; it is exactly this small difference that we are interested in because it brings the new flow and the new routing into agreement once more. Thus, the numerical error is corrected on each iteration rather than accumulating over many iterations. This approach solves the problem of flow and routing mismatch, but introduces another problem that is even more serious.

The FD algorithm is guaranteed to converge (possibly after many iterations) to within an arbitrary tolerance of the true global optimum. The rate of convergence, however, can in many cases be quite slow. This problem is described in some detail by Gerla [5]. The problem arises when we combine this slow rate of convergence with recomputing the flow based on the derived routing at each iteration. We have observed cases where the flow produced by the new routing was significantly different from the flow produced by the FD algorithm directly. The perturbation we have introduced, though very small, has caused one of the requirements of the algorithm to become invalid; the performance has not improved with each iteration of FD. This invalidates the convergence guarantees for the FD algorithm. Much to our dismay, we discovered that this situation was not infrequent. As a result, we abandoned this method of recomputing routing after each iteration in favor of the memory intensive method of storing flows internally on each link for each destination.

Routing Loops

The routing tables produced by MRMAIN occasionally result in looping paths for some packet flows. A sample iteration from MRMAIN as it attempts to optimize routing for a simple 6-node network demonstrates how this may occur (see Figure 7). The nodes are connected in a "figure 8" topology; the flow to be routed is input to the network at node 2 and output from the network at node 5.

```
            input                 input                input
              ¦                 1.0 ¦                 .43 ¦
              V                <--- V                <--- V
       1------2              1------2              1------2
       ¦      ¦ ¦          ¦ ¦      ¦            ¦ ¦      ¦ ¦
       ¦  1.0 ¦ ¦ 1.0  1.0 ¦ ¦ 1.0  ¦        1.0 ¦ ¦  .57 ¦ ¦ .57
       ¦ <--- ¦ V          V ¦ ---> ¦            V ¦ <--- ¦ V
       4------3              4------3              4------3
     ¦ ¦      ¦              ¦      ¦ ¦          ¦ ¦ ---> ¦ ¦
 1.0 ¦ ¦      ¦              ¦      ¦ ¦ 1.0  .57 ¦ ¦ .43  ¦ ¦ .43
     V ¦      ¦              ¦      ¦ V          V ¦      ¦ V
       5------6              5------6              5------6
       ¦                     ¦ <---                ¦ <---
       V                     V  1.0                V  1.0
    output                 output                output

        Initial          Shortest Path        MRMAIN-produced
        Routing             Routing              Routing
         (A)                  (B)                  (C)
```

Figure 7 . A Single FD Iteration

The initial routing (before any iterations of FD) is shown in (A). The arrows (and numbers) next to some links indicate the path taken by the flow (and the fraction of the total flow) destined to node 5 at each node in the network. Stated in another way, we have shown only the node 5 entry of the routing table for each node in the network. Thus, we see that at node 2, all (1.0 or 100%) of the flow destined for node 5 is routed via node 3. Similarly, all the traffic that reaches node 3 and is destined towards node 5 is routed via node 4.

The FD algorithm calculates a shortest path routing based on the k-th power performance metric. This represents the best single-path routing for an incremental increase in traffic. This routing is shown in (B). By combining the initial routing and the shortest-path routing, the FD algorithm produces a new flow and routing which improves the network performance metric. The FD algorithm terminates if it is not possible to improve the initial routing (i.e., the initial routing is optimal or very close to optimal).

The new routing that is produced by MRMAIN after the first iteration of FD is shown in (C). The average network delay for (C) is less than the delay for either (A) or (B). For 50KB links and a 47.5KB/sec flow from node 2 to node 5, the average network delays for cases (A), (B), and (C) are 600ms, 1000ms, and 74ms respectively. Note, however, that even though the network delay has improved, packets are looping on the link between nodes 3 and 4! We see that node 3 sends 57% of the traffic destined to node 5 via node 4. At the same time, node 4 sends 43% of the traffic destined to node 5 via node 3. Since these numbers are less than 100%, a decreasing fraction of the packets loop between nodes 3 and 4 until they are eventually routed via alternative paths and arrive at node 5.

The sequence (A), (B), (C) above shows only the first iteration of the FD algorithm. The routing is not yet optimal, so the algorithm continues to run, improving the routing a little with each iteration. The network delay, however, does not improve dramatically with each subsequent iteration as it did for the first iteration. After 100 iterations, the FD algorithm had not yet converged to the optimal routing. After 360 iterations, the FD algorithm aborted because the network performance metric improved less than 0.001ms between iterations. The routings after 100 and 360 iterations are shown in (D) and (E) of Figure 8.

We note that the fraction of packets routed on the link from 4 to 3 is slowly decreasing with increasing numbers of FD iterations. If we look a little closer at a sequence of 6 or 8 iterations of MRMAIN, we notice a pattern. First, the split of traffic at node 2 will approach 50-50 for a few iterations. Then, the fraction sent in each direction on the link between nodes 4 and 3 will decrease slightly for a few cycles while the split at node 2 drifts slightly away from 50-50. This process is repeated as FD iterates, with the splitting fraction at node 2 remaining near 50-50 while the amount of traffic sent on the line between nodes 4 and 3 gradually decreases. The flow and the routing are improving, but at a painfully slow rate.

```
                 input                              input
             .471 |                             .468 |
             <--- V                             <--- V
           1------2                           1------2
        |  |      |  |                     |  |      |  |
    1.0 |  | .254 |  | .529            1.0 |  | .182 |  | .532
        V  | <--- |  V                     V  | <--- |  V
           4------3                           4------3
        |  | ---> |  |                     |  | ---> |  |
   .845 |  | .155 |  | .746           .933 |  | .067 |  | .818
        V  |      |  V                     V  |      |  V
           5------6                           5------6
           | <---                             | <---
           V  1.0                             V  1.0
         output                             output

      MRMAIN-produced                   MRMAIN-produced
         Routing                           Routing
     (100 iterations)                  (360 iterations)
           (D)                               (E)
```

Figure 8 . Routing After Many FD Iterations

Even after 360 iterations, node 3 still sends .182 of its packets destined for node 5 via node 4. For a 47.5KB/sec flow, and including the small amount of looping that still remains, there are still 47.5 * .532 * .182 (1 + .067 * .182 + smaller terms) = 4.6KB/sec on the line from node 3 to node 4! The occurrence of looping paths in routings produced by MRMAIN was (unfortunately) quite common. In many cases, this contributed to the slow convergence of the multi-path routing optimizer.

Many of the difficulties we encountered with the FD algorithm can be explained in terms of the way in which FD "navigates" through the solution space as it searches for the optimal multi-path flow and routing. The solution space is defined by the performance metric, and is a concave surface in an n-dimensional space defined by the network variables. For ease of visualization, we can think of the analogous case in only 3 dimensions (see Figure 9), where the z-axis represents the network performance metric, and the x and y-axis represent all other variables in the n-dimensional space.

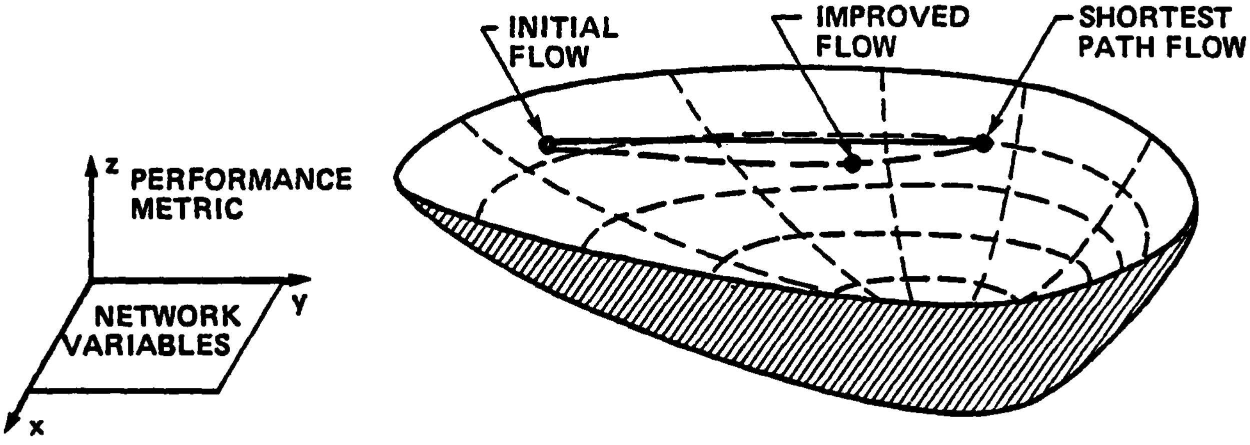


Figure 9 . N-dimensional Solution Space

The solution space is guaranteed to be concave by the mathematical properties of the performance function chosen. Once we find a feasible initial routing (a point on the 3-dimensional solution surface), the FD algorithm will bring us closer to the optimal solution with each iteration. As described earlier, the FD algorithm calculates a shortest path solution (a second point on the solution surface). The method of generating the shortest path solution guarantees that the line connecting it to the initial flow solution will be in the direction of maximum flow improvement (for incremental flows). What this means in terms of our 3-dimensional analogy is that the line connecting the initial and the shortest path flows will pass over a portion of the solution space that is convex (i.e., lower or better performance metric). The projection of the line connecting the initial flow and shortest path flow is shown as a dashed curve in Figure 9. We can find this point of improved performance by taking a linear combination of the initial flow and the shortest path flow; we are minimizing the performance function over the function defined by the dashed curve. The improved flow that is produced is shown at the minimum point of this curve.

We can now visualize cases where the convergence of the FD algorithm is quite slow. If the shape of the solution space is relatively flat, we may wander leisurely down a mild slope towards the optimum flow for many iterations of the FD algorithm. This is shown in Figure 10.

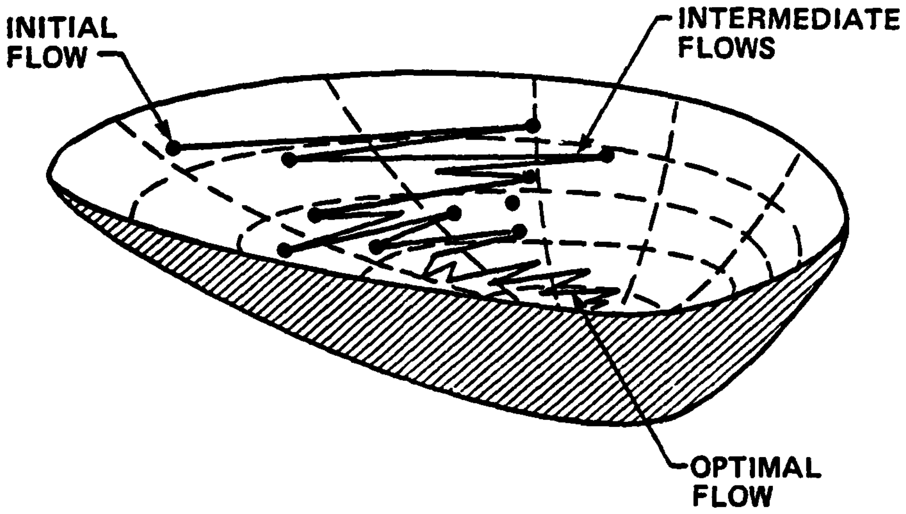


Figure 10 . Slow Convergence

To attempt a speedup in the rate of convergence, we could try using different performance metrics. In particular, we could try different values of k in our k-th power performance function. Values of k > 1 will cause the shape of the solution space to change. In addition, the slope of the solution space may also be steeper; however, it is not clear how this will affect convergence of the FD algorithm. Preliminary work along these lines was inconclusive. More work is needed to investigate ways of improving convergence via modifications to the performance metric.

We can also visualize why very small numerical errors can destroy the ability of FD to converge to a solution. If we are in a relatively flat portion of the solution space, small roundoff errors can cause the assumptions of the FD algorithm to be violated. In Figure 11 we have an initial flow and shortest path flow as before.

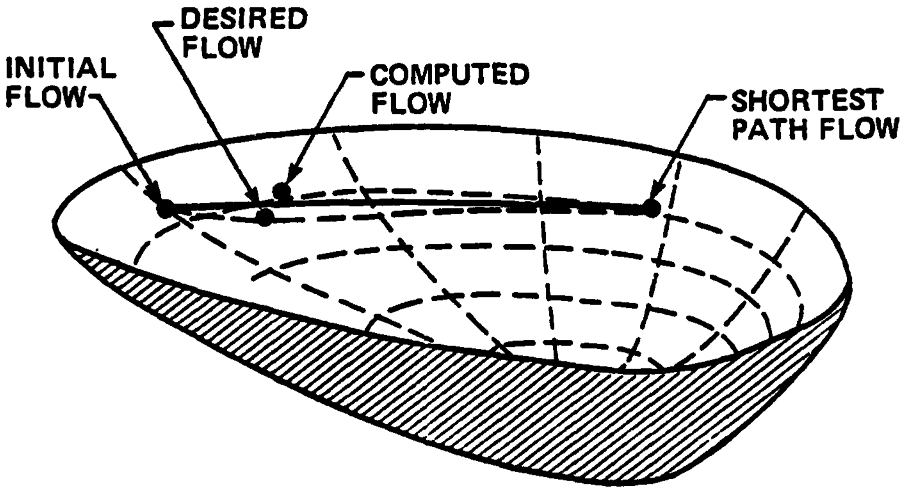


Figure 11 . Effects of Roundoff Error

The "desired flow" is the value that would be calculated if no roundoff error was involved in the FD algorithm. It improves the network performance slightly (i.e., it is "below" the initial and shortest path flows). The value we actually compute (includes roundoff error) is labelled "computed flow." It degrades network performance slightly (i.e., it is not below both the initial and shortest path flows). The two values are numerically close, differing only by a small roundoff error. Since the "actual flow" is what the algorithm uses, we see that the FD has not improved the performance of the network even though the flow is not yet optimal! This violates the assumptions on which FD is based. At the same time, we gain insight as to why the FD algorithm failed in some of the modified algorithms we described in previous paragraphs.

We have verified the operation of the FD algorithm as implemented in MRMAIN by comparison to numerous test cases and by comparison to results published by Schwartz [9]. MRMAIN was used to create optimal multi-path routings for a number of experiments that were also run using the ARPANET simulator. It was also used to model optimal single-path routing (i.e., optimal single-path routing was input to MRMAIN rather than generated by MRMAIN). These results are described in other sections of this report.

6 SIMULATION OF THE CONGESTION CONTROL ALGORITHM

In this chapter we will discuss our experience implementing congestion control in the simulator. The congestion control algorithm proposed by Eric Rosen in [3], will soon be implemented in the ARPANET.

Implementing a protocol is a considerably easier task in the simulator than it is in the real network. This is true for any number of reasons: use of a high-level language; sophisticated trace and debugging facilities; and the ability to ignore details such as interrupt handling, storage allocation, and mutual exclusion among cooperating processes. On the other hand, the protocol itself must be implemented completely: the action for any combination of events and protocol states must be defined. Therefore, implementation on the simulator provided an efficient way to examine the algorithm proposed in [3]. Not surprisingly, we found that, although the basic design was sound, many details had to be resolved. In the majority of cases, this could be done by appealing to the rationale underlying the initial design. The rest of this chapter will go through the design and discuss decisions made during the simulator implementation.

The design indicates that a line is classified into one of three congestion levels (unloaded, fully loaded, and congested) on the basis of three counters: a counter which is incremented each time the addition of a packet to the Task queue causes the length of the queue to be greater than a given threshold; a counter which is incremented each time Task refuses a packet because no logical channel is available; and a counter which is incremented by M2I each time it discards a packet because no buffer is available, and by Task each time it refuses a packet because no store-and-forward buffer is available. An outstanding question is whether the number of packets queued for Task includes routing updates and Hello/IHY's.

The design does not specify how to use the three counters to classify a line. In the present implementation a congestion averaging routine is run every n fast ticks (where n is a parameter). The routine adds the counters together and compares the total to two thresholds (for fully loaded and congested). The congestion level is stored in a table, and the counters are cleared. The design also mentions the possibility of further averaging, which is not included in this implementation.

The congestion level of each outgoing line in an IMP is reported to all other IMPs simply by adding the information to the routing update. A routing update is generated if it would have been before (i.e., based on the change in delay and the current delay threshold), or if the congestion level changes on any line.

Only the source IMP throttles the flow from a source IMP to a destination IMP. The throt'ling algorithm varies the limit on the number of packets for each destination which can be queued by Task for transmission by M2I. That is, there is a table giving the limit on the number of packets which can be queued for each destination. Task will refuse a packet from a local host which is to be forwarded to another IMP if the limit on the number of packets queued for its destination has already been reached. In the implementation, only packets queued for the same modem as the packet being checked are counted, and packets waiting for acknowledgment are included in the count.

The SPF routing algorithm will maintain the congestion level of the path to each destination, as well as the path itself. The congestion level of a path means the congestion level of the most congested link in the path. Note that this requires each IMP to remember the congestion level of each line in the network, as well as the delay.

For each destination there is a path to that destination, a modem (for the first hop in the path), the most congested link (or possibly more than one), the congestion level of that link, and a current limit on the number of packets which can be queued for the destination. For each destination, the limit on the number of queued packets must be less than or equal to the number of logical channels on the first line. There may also be separate upper and lower limits on the limits, for each outgoing line in each IMP, for each IMP, or for the network as a whole. This is not to say that the IMP must keep track of every one of these pieces of information -- we will discuss this below.

The design specifies that if the path to some destination becomes congested the limit will be decreased, and if the path becomes unloaded the limit will be increased. Periodically, the limit will be decreased if the path is still congested, or increased if the path is still unloaded. The limit is not changed while the path is fully loaded. Thus, the limit to each destination can be changed either periodically, or by the arrival of an update. We will discuss each in turn.

Every m fast ticks (where m is a parameter) fast timeout calls a routine which looks at the congestion level on the path to each destination. For each destination, if the path is congested the queue limit is decreased, and if the path is unloaded the limit is increased. The design discusses various ways to increase and decrease the limit; we will assume that the change is constant (i.e., a settable global parameter). There will also be upper and lower limits on the range over which the queue limit can vary. These will be parameters, but it is not clear what they will depend on. For example, they could be set for each outgoing line in each IMP, or for each IMP, or for the network as a whole.

When an update arrives, it may change any of the following (to each destination): the path, the congestion level on the path, or the set of most congested lines on the path. Note that any of the three can change independently of the others. We have to specify how the limit on queued packets changes in each case. The congestion control algorithm specifically ignores the delay along the path.

If the path changes, but the set of most congested lines and their congestion level stays the same, one consideration is that the outgoing line may (not must) change, possibly changing the number of logical channels, and hence the limit on queued packets.

Note that the SPF algorithm may not be able to detect the case where the path changes but the set of most congested lines and their congestion level does not. If so, any path change will have to be handled as though the set of most congested lines changed. This is discussed below.

If the limit on queued packets is larger than the (new) number of logical channels, then the limit can simply be reduced to the number of logical channels (or to the upper limit on the queue limit, if that is smaller). If the number of logical channels on the outgoing line increases, but the most congested lines and their congestion level do not, we want to maintain the effect of the current level of control. In the case where the path is unloaded and the queue limit is as large as possible, it may be that the correct strategy is to increase the queue limit, perhaps up to whatever limit is imposed by the (new) outgoing line. The conservative strategy is to leave the queue limit the same. In the case of a congested or fully loaded path, the queue limit should stay exactly the same. This is discussed again when we discuss updates which change the set of congested nodes.

If the path stays the same, but the set of congested lines or the congestion level changes, we should adjust the queue limit. This is the easy case: if the path becomes congested, the limit should be decreased; if the path becomes unloaded, the limit should be increased. There is a question of what to do about updates which report congestion at different lines along the path. The first will cause the congestion level of the path to change, so the queue limit will be decreased. Subsequent updates which report congestion along the path will not cause the congestion level of the path to change. Indeed, the SPF algorithm may not even report the change. In some sense, congestion is worse if more than one line is congested, but we will assume that extra controls do not need to be applied. If the congestion persists, the periodic process will increase control.

If the path changes, and the set of congested lines changes (or we cannot rule out the possibility that it has changed), the correct action depends on whether the flow under consideration is causing congestion or not. In general, we only know that congestion is occurring somewhere along the path from source to destination, not which flows are responsible. However, in theory we need to make a prediction about the effect of changing from one path to another. The change may or may not make congestion worse on the new path, and the congestion on the new path may or may not be the same as the congestion on the old path. The new queue limit on the new path should be based on the queue limit on the old path, or the congestion level on the old path, or the congestion level on the new path. In practice, we will be making a guess as to the strategy which will work best in most cases. This analysis does not pretend to take account of all possible pathological cases, but is suggested as a basis for simulation experiments.

If we do not adjust the queue limit when an update does not change the congestion level along a path which does not change, we can do the same thing when the congestion level on the new path is the same as the congestion level on the old path. Note, however, that this assumes that the old queue limit (from the previous path) is approximately correct. The periodic process will change the queue limit if appropriate.

It is quite unlikely that an update would cause the path to switch from an uncongested path to one which was congested. In this case, however, it seems prudent to apply more control and decrease the queue limit. Again, this assumes that the old queue limit is approximately correct.

Presumably, the most common case is when an update switches the path to one which is less congested. The problem is that we really have no information about the effect of the increased flow along the new path. The design makes the point that if the offered load is more than the capacity of any path, then the routing will oscillate, and the congestion control scheme should lower the queue limit until the load has been throttled to the capacity of the path, stopping oscillation. It is possible that the oscillations in flow caused by the oscillations in routing will also cause the congestion measurements to oscillate, so that just as the SPF scheme sees a shorter delay on the currently unused path, it will also see less congestion. This would stop any decrease in the queue limit. The congestion measurements must therefore be more damped than the delay measurements and routing changes. Assuming that this is so, under conditions of overload both paths will appear congested, and so the queue limit will gradually be decreased until the load is adequately throttled.

If no path is loaded, routing will be stable relative to the updating period. Therefore, if routing switches to an unloaded path, the correct strategy might be to relax congestion controls and reset the queue limit to its upper limit. On the other hand, since there is no way of ruling out the possibility that the change in route will cause congestion on the new path, the conservative strategy is to simply treat the update as it would be treated if the path had not changed, and increase the queue limit in the standard way. The cost of this strategy is that flow might be restricted unnecessarily. This would of course be temporary.

There is a problem concerning initializing the queue limit when a destination becomes accessible. When an update arrives, some destinations which were inaccessible may become accessible because the update may report one or more lines up which were down. A line which goes down will stay down for at least a minute, so the queue limit to an inaccessible destination will surely be out of date by the time the destination once again becomes accessible. We have to decide how to set the queue limit.

We believe that the congestion level on the line which comes up is irrelevant, except insofar as it determines the congestion level of the whole path. If the path is unloaded, we can safely set the queue limit to its maximum value (e.g., the number of logical channels on the outgoing line). If the path is fully loaded or congested, the conservative strategy would be to set the queue limit to its minimum value. However, this could be unfair, since the queue limit to the same destination at other competing sources could be much larger. Although the queue limits at these competing sources would be gradually dropping towards their equilibrium (i.e., fully loaded) values, the equilibrium value would in general be above the minimum. In this IMP, however, the queue limit would stay at the minimum.

We suggest that the solution is to set the queue limit to some intermediate value which is computed as a simple function of the minimum (and possibly maximum) value. It seems as though this is not very critical for 8 channel lines (there seems to be nothing to choose between 3, 4 and 5) but could be more difficult with more channels.

We will now summarize the conservative design. Changes in queue limit to any destination are caused by the periodic process if it finds the path to any destination either congested or unloaded, and by the arrival of any update which changes the congestion level of the path to the destination. Updates which change the path to the destination, or the set of congested nodes, but which do not change the congestion level of the path, have no effect on the queue limit.

When the IMP comes up, queue limits are initialized to the upper limit. The count of packets to compare against the queue limit includes packets waiting for retransmission, but not packets which are queued for other modems. The count of packets on the Task queue includes routing updates and Hello/IHY's as well as data packets.

This design requires that each IMP keep track of the congestion level of each line in the network (from the latest update), the congestion level on the path to each destination, and the queue limit to each destination.

Finally, there is an issue with satellite lines. The simple scheme outlined here depends on controlling a flow by controlling the number of queued packets. However, the number of packets which should be queued to sustain a particular flow level depends on the time needed to return a line protocol acknowledgment. Obviously this is much larger for satellite lines than it is for land lines, so this scheme may not work in situations where routing is switching between a satellite line and a land line. This problem could be solved by Rosen's design, which includes a scheme for directly controlling flows to a destination.

7 ISSUES IN INTERNET GATEWAY DESIGN

7.1 Internet Performance

The major activities in the work involving the Internet have centered on analysis of the design issues, and development of initial design ideas and approaches to be used as a basis for the next implementation cycle of gateways. The goal of this Internet effort has been to develop an environment which supports operational use and permits experimentation.

The results of the analysis work have been presented in a series of Internet Experimental Notes ([10], [11], [12], and [13]) and will not be replicated here. In this section, we present some elements of the specific design for a new implementation which applies the general model presented in the IENs. This design is being developed further as part of the Internet activities under a separate contract.

This section addresses three major issues. First, some specific architectural choices are discussed. Second, issues involved in the internal design of gateway process structure are presented. Third, tools and mechanisms for operational support and maintenance of a gateway system are detailed.

7.2 Architectural Issues

The basic architecture of an Internet system were presented in the IENs. In this section, we discuss four specific issues:

- host interface to the Internet,
- interoperability of autonomous gateway systems,
- congestion control, and
- logical addressing.

The following discussion presents current thoughts and outstanding questions concerning the design of modifications to the current gateway architecture.

7.2.1 Host Interface to the Internet

In AUTODIN II, the source nodes maintain structures called "Bookkeeping Blocks (BKBs)." Each source node has one bookkeeping block for each host-host "connection." The main purpose of these blocks is to maintain the information needed to enforce flow control on a host-host basis. These structures are also useful for accounting and instrumentation purposes. The source gateways will need to have similar structures -- we will call them "connection blocks" in this discussion. These blocks define the granularity of flow on which we do flow control. If there is one block per source/destination host pair, then we will be able to control the flow between each host pair without any interaction with flows from the same source host to other destination hosts. We could instead keep one block per source host, but then we would be unable to throttle any one host-host flow without also throttling all other flows from the same source host.

The tradeoff here is the amount of fine grain we have in throttling flows versus the amount of overhead (memory, and to some degree, cycles) we need to maintain the additional information needed to make the grain finer. The flow control with the finest grain might be based on the notion of a flow between a source/destination pair of PROCESSES; however, it is not clear that we will be able to make such fine distinctions in our flow control algorithms, and the overhead might be very large. The coarsest grain might come from using the notion of a source/destination pair of gateways. That results in much less overhead, but provides no fairness among different flows between the same pair of gateways. Applying flow controls on a host-host basis is probably the best compromise.

Host-to-host in this context can mean logical-address-to-logical-address, physical-address-to-physical-address, or some combination (such as defining a "host" to be a particular logical/physical address pair). Logical-address-to-logical-address might be the best way of identifying flows for the purposes of flow control, but that might give an advantage to hosts which happen to have a lot of logical addresses. Rather than precisely setting flow control based on purely a priori considerations, we propose to make the system flexible enough so that we can at any time (by altering the setting of a parameter and then restarting) change the way we set up the connection blocks (and hence the way we individuate flows for flow control purposes).

The most familiar type of flow control is based on a windowing scheme, as in TCP. Windowing schemes attempt to use a single mechanism, that of sequence numbers, for three different purposes: flow control, error control, and sequentiality. The use of a single mechanism for three disparate purposes is prone to introduce strange interactions in the inevitable cases in which no single action can serve each of the three separate purposes. In the Internet, we need to have flow control, we would like to have error control, and we do not want sequentiality (at least, not always), so windowing based on sequence numbers would be particularly inappropriate. We suggest the following tentative scheme for enforcing flow control. Each connection block should contain a number of packets P and a number of bits B so that no more than P packets per unit time or B bits per unit time could be accepted by the gateway on that particular host-host flow. If a source host tries to exceed this rate, any packets it sends in excess of the rate will be explicitly NAKed. The NAK will contain enough information to uniquely identify the NAKed packet, and the host will be advised of its maximum send rates. NAKed packets will be discarded by the gateway. (Of course we retain the option of not sending NAKs if we are out of resources; that is, if necessary, we can just drop packets with no notice, but, when possible, NAKing is better.)

Hosts should always be able to find out what flow control limits are being imposed. We need to have a control message which a host can send to a gateway to ascertain these limits; the gateway will reply if possible. One possible implementation is to let the host set a bit in the internet header of a data packet to ask the source gateway to explicitly acknowledge its receipt of the packet. The ACK can contain any flow control restrictions, and hosts can set this bit occasionally to see whether the source gateway is really taking packets. The source gateway might want to refrain from sending these ACKs until it forwards this packet on its next hop. This delays the sending of the ack, but gives it somewhat more meaning.

This flow control mechanism gives us a means for enforcing flow control which is independent of any particular flow control algorithm used within the Internet. Information from the routing algorithm, from specific congestion control techniques, and from monitoring of local resource utilization (e.g., buffer space in this gateway itself), can all be mapped into the rate limitations coded within the connection blocks, and we can experiment with the mapping functions without any need to modify host software.

Since we will have only a finite number of connection blocks, we will have to recycle them. We can recycle a block if its connection has been idle for a certain amount of time. If a packet comes in from source logical address S for destination logical address D, and there is no S-D block currently in use, one must be allocated. If no S-D block is free and available for allocation, we will have to NAK the packet. This is a consequence of the need to have flow control, and hence to maintain information about flows. Note that since we are keeping flow control information in these blocks, we do not want to be too free about recycling them or we lose important information.

We also suggest having error control on the host access protocol. One approach is to require the host to put in a software checksum; we could discard the packet if the checksum is incorrect. Experience in the ARPANET indicates that an access

Pathway checksum is really needed, and should be more powerful than the current additive checksum.

At any rate, we will need an end-end checksum (purely within the internet system, i.e., between the source and destination gateways), which we should probably make more powerful.

We will have to keep measurements of the utilization of the connection block resource. One approach is the following: when a request for a new connection block is made, increment by one a counter of requests, find out how many blocks are free and available, and add that number to another counter. Dividing the second counter by the first yields the average number of blocks available per request, which is a good measure of whether the utilization is high. We will also keep track of the rate at which blocks get recycled. There are some measurements which we could keep in the blocks themselves: packet and bit rates on the "connection," packet size histograms, number of packets for each protocol number, number of packets in each service class, etc.

In the current Internet, the protocol used among gateways is the same as that used among hosts and at the gateway-host interface. In the ARPANET, the protocol used at the host-IMP interface is all contained in the 1822 leader, and the protocol used between IMPs is contained in the packet headers. The actual leader is never carried across the network, but the header (which does not resemble the leader in format) contains enough information so that the leader can be reconstructed before the packet is passed to the destination host. Packet headers contain somewhat more information than the leaders do, but hosts never see the headers. We propose to introduce this distinction in the Internet where the IP header which exists between a host and gateway might differ from that which exists during transit between gateways. The information fields listed below do not have to be in the host-generated header, but are necessary in transit:

- Physical address of the source gateway. This will be the node number of the source gateway within the numbering scheme of the Internet. We need this for returning end-end control messages, such as DNA messages. It is also good to have such a field for debugging purposes. Yet this number may not even be known at the host level.

- Physical address of the destination gateway (needed for routing).

- Packet type. Is this a control message (routing, DNA, etc.) or a user-supplied data message. By "control message" I refer only to messages which are used for internal Internet control purposes.

- The "ACK this packet, neighbor" flag. The delay measurement procedure proposed for the routing algorithm requires that we be able to select certain packets for hop-hop acknowledgments.

- Compatibility/version flags, so that we can distinguish among different versions.

- Spare bits.

- If WWVB radio clocks are available, words for timing purposes (which can be optional).

- End-end checksum.

The following information is needed in the host specified header as well as in transit:

- Source host logical address. This should be verified by the gateway; if a certain logical address is not recognized by a gateway as that of a local host (i.e., a host that it knows how to reach "directly" without using the Internet), the packet should be discarded.

- Destination host logical address.

- Checksum for access Pathway only.

- Service class (including priority, precedence, etc.).

- Packet identifier.

- Protocol number.

- Information needed to enable fragmentation/reassembly.

- Instrumentation flags. This would include mechanisms like the trace bit in the ARPANET 1822 leader, which, when set, causes a packet to be traced on its path through the network. Issues in designing a trace package for the Internet similar to the one we have in the ARPANET are discussed later. Another possible instrumentation bit might correspond to the "tagged packet" bit of the ARPANET, which causes a packet (with no data) to have its data part filled with the identifier of each Switch it passes through, as well as the delay from that Switch to the next. Another sort of instrumentation might be a selective acknowledgment bit, which would cause an ack to be sent to the source gateway from either the destination gateway or from every intermediate gateway. This might be useful for timing.

- A flag which asks the source gateway to ack this packet and also furnish flow control information.

We also need a special host-gateway control message that will enable a host to ask the source gateway for its delay (in ms.) to the destination gateway which will be used for a particular specified destination host, as well as a reply for the source gateway to send back to the host.

7.2.2 Interoperability of Gateway Systems

As a general goal, the Internet should be capable of supporting a large number (possibly thousands) of networks interconnected via gateways. The traditional approach to routing and network management in general is likely to prove intractable in this environment. The approach we have been investigating involves creation of a number of autonomous gateway systems (each containing possibly 50 or 100 gateways) that interoperate to create an effective Internet which will be seen by the hosts as the union of all the gateways. This flexibility will allow configuration of networks and gateways into autonomous systems based on traffic patterns, without requiring hosts to be aware of the internal structure.

This section discusses some of the issues in gateway design and describes the gateway's interaction with hosts, other gateways in its own system, and gateways in other systems.

Consider the case where a host on Internet system A needs to communicate with a host on Internet system B. There must be some network C so that a gateway GA of internet A and a gateway GB of system B are both hosts on network C. Now GA can be regarded as the destination gateway of system A, since as far as GA is concerned, system B is a Pathway (no internal structure) to the destination host. Similarly, gateway GB can be regarded as the source gateway for system B, since it regards system A as a Pathway to the source host. When gateway GA gets some data for the destination host it only needs to strip off the network access protocol of system A, replace it with the network access protocol of system B, wrap it in the access protocol for network C (which is GA's Pathway to gateway GB, hence to system B), and then send it to gateway GB via network C. That is, GA accesses system B just as if it is an ordinary host on system B. However, instead of giving its own name as the source address in the internet protocol, it gives the name of the original source host. Of course, some simplification is possible if both systems use the same access protocol and the same logical addressing scheme, since then it is not necessary to remove one protocol envelope and replace it with another. Gateway GA will also have to have some way of knowing that certain addresses can only be reached through the other system; that information is properly stored in the logical-to-physical translation tables.

There are likely to be subtle problems with mechanisms such as flow control. If gateway GB wants to send flow control messages back to the source host, GA might want to let those messages go right back to the actual source host, or it might want to intercept them, and use them as input to its own flow control mechanism. From any particular gateway's perspective, it has to decide what to do when it receives flow control messages from another system which are destined for a host on its own system. One approach is to intercept these messages and feed them into the local congestion control system.

One other area of concern in this approach is fault isolation. One technique is to routinely keep track of the rate at which we receive data on each "connection" in order to be able to say whether someone's data has arrived at our system, or whether it must have gotten lost before reaching our system. Within any single system of gateways, we can use some of the instrumentation bits in the packet headers for similar purposes. If some particular source host claims that his data is not getting through, we can set an instrumentation bit in each packet that we see from him. Each of the gateways that sees a packet with this bit set can increment a count. This would help us to understand whether all packets entering a system are also leaving it, i.e., whether they are getting lost within some individual network. Keeping such counts in the intermediate gateways too would enable us to determine which individual network is at fault.

This suggests another feature to be included in the gateways. If particular networks in their own access protocols have bits which can be used for instrumentation within that particular network, we need to be able to turn them on selectively when sending packets into that network. For example, the gateways ought to be able to select certain packets going into the ARPANET (for example, every nth packet from a certain source to a certain destination) and turn on the ARPANET's trace bit in those packets. This could be very valuable for fault isolation.

7.2.3 Congestion Control

This section addresses some of the issues in an initial approach to congestion control. Basically, each gateway will determine, for each of its Pathways to neighboring gateways, whether that Pathway is "overloaded," "underloaded," or "fully loaded." Given this determination, it is not difficult to modify the SPF algorithm so that it determines, for every route between a source and destination gateway (where a route is just an ordered sequence of Pathways), whether that route is "overloaded" (i.e., has at least one overloaded Pathway"), "underloaded" (i.e., consists exclusively of underloaded Pathways), or "maximally loaded" (i.e., consists of at least one maximally loaded Pathway but no overloaded Pathways). That is, at a given gateway G, the routing algorithm would tell it that the route from itself to another gateway G' was in one of these three states. Throughput limits on hosts will vary dynamically, as follows:

a) If a host-host connection goes to a gateway along an overloaded route, the throughput limits for that connection should be reduced periodically by a certain (small) amount, until a minimum is reached, or until the route to that gateway is no longer overloaded.

b) If a host-host connection goes to a gateway along an underloaded route, the throughput limits should be increased periodically by a certain (small) amount, until the limits reach infinity or the route ceases to be underloaded.

c) Host-host connections using "maximally loaded" routes should have their throughput limits remain the same.

The basic notion is to have a disciplined and enforced throttling of input into the Internet, and in particular, to throttle all that traffic (but only that traffic) which would travel a congested route. Throttling limits would increase and decrease slowly until an appropriate level of load is reached. Note that controls are applied not merely to hosts which happen to get their packets dropped due to congestion, but rather to all hosts which are using congested Pathways.

By routinely keeping throughput statistics in the connection blocks, we can introduce mechanisms to increase fairness. When lowering the throughput limits, for example, we can decrease the maximum allowable throughput by a percentage of the actual measured throughput, so that the heaviest users get throttled the most. This might tend to lead to an equalization of flows when the internet is congested, which is an important fairness consideration.

One important issue is how a gateway can determine the congestion status of its outgoing Pathways. We will have to investigate this issue experimentally. Congestion in the individual networks will tend to cause certain symptoms (such as queues getting longer than a threshold and staying longer for a certain amount of time. or an excessively high loss rate) and we will have to determine experimentally what these symptoms actually are. There will also be internal gateway congestion to worry about, which would show up as buffer shortage or high CPU utilization or something similar. When the congestion is internal to the gateway, all Pathways leaving that gateway should be declared congested, since the congestion affects them all equally.

Note that in order to determine whether to change the throughput limitations applied to a certain connection, we must know the destination gateway of that connection. This means that connection blocks must be individuated not only by source and destination host, but also by destination gateway. (That is, if we are load splitting by alternating the traffic for a certain destination host to different destination gateways, then we need separate connection blocks.) Also, if different classes of service cause traffic for the same destination gateway to take two different routes, we will need a different connection block for each type of service that requires a different route.

7.2.4 Logical Addressing

This section addresses issues about the application of logical addressing to the Internet. We have defined in the IENA the notions of a logical address being "authorized" and "effective." In the ARPANET, when an IMP first comes up, the logical addresses of all directly connected hosts would be "ineffective" by default. When the connection between the IMP and the directly attached host first comes up, the host must send in LAD messages to indicate which of its logical addresses should become effective, and the IMP, if these logical addresses are authorized, will make them effective and will acknowledge this fact to the host. This works out nicely because host and IMP can easily tell when the other has gone down (or more precisely, when the connection between them has gone down).

This procedure would be too strict to apply to the Internet. That is, the gateways will not be able to keep track of all the host up/downs as a matter of course; rather, they will only be able to get this information on an exception basis, as needed. Certainly it is not possible to require every host to redeclare its logical addresses after every gateway crash. Therefore we will probably have to make all logical addresses be effective by default.

We can give the hosts the ability to declare certain logical addresses to be ineffective. or to declare that it only wants certain of its logical addresses to be effective and the others to be ineffective. That is, we can allow the hosts to use LAD messages as an option. The gateways receiving LAD messages would acknowledge them (possibly negatively, if a host asks to be addressed with an "unauthorized" name). A LAD message would retain its effect until superseded by another LAD message from the same host, or until the gateway restarts, in which case all logical addresses would become effective again, by default. This does mean that hosts may sometimes receive messages that are not intended for them (i.e., misdelivery of data). This can happen if two hosts share a port in some network but have different logical addresses. Since both logical addresses are effective by default, the Internet has no way of knowing automatically when one host is disconnected and the other one is connected in its place. If one of these hosts receives data intended for another, it must be willing to inform the Internet of the logical addresses effective at that time. This is not such a problem if the network itself has logical addressing, since then we can rely on the network to tell us which host is really there, and to prevent misdelivery.

It is interesting to consider the particular case of a gateway on a local net. In this case, because of the extremely high bandwidth and low delay of the network, it might indeed be feasible to have the gateway perform real-time monitoring of the up/down status of each host, and of the effectiveness of each authorized logical address for the local net. When a gateway comes up, all the logical addresses can be marked ineffective by default, and the gateway can query each host to see what logical addresses it is to be known by at that time. This suggests that we define a protocol by which a gateway could demand this information from the hosts on its network, and the hosts would be required to respond if they want to be able to receive Internet communications. (If a host sends a packet to the Internet with a particular source logical address, we might want to regard that as an implicit declaration that it is willing to receive data at that address.) Gateways on local nets would be expected to strictly monitor hosts on the local net, but gateways on lower bandwidth nets would not. The protocols need to be general enough to handle both cases.

These considerations suggest a possible way of connecting local nets to long haul nets like the ARPANET. We could have one gateway be a host on both the ARPANET and the local net. From the perspective of the ARPANET, the gateway would appear to be a single host with many logical addresses. That is, the ARPANET logical addresses of all hosts on the local net would map to the gateway's physical address. The gateway could keep track of which hosts on the local net are up or down, and keep the ARPANET informed of this. The gateway could handle all protocol needed to interface between the ARPANET and the local net. In this way, hosts on the local net could be treated as ordinary ARPANET hosts.

Another useful mechanism is some analogue of the ARPANET's "host going down" and "imp going down" messages. That is, hosts on a network should be able to inform all the gateways on that network that they are going to be down for a while, indicating a reason and a duration of the outage. Any time a host on some other network tries to communicate with a host that has declared itself to be down in this way, the destination gateway can send the source gateway a control packet with the reason for and duration of the outage. and the source gateway can feed this back to the source host. Once a host has declared itself down by sending a "host going down" message to a particular gateway, that gateway should consider the host to be down until it hears otherwise from the host.

7.3 Gateway Design Issues

This section represents an initial attempt to block out the process organization of the gateway, and to look at some of the problems of inter-process communications and resource management that arise. These considerations are only preliminary, to be used as a guide for further work.

7.3.1 Software Organization: Process Structure

Each network interface or access line will have one process associated with it (call this a "Pathway Access Process" or PAP), or maybe two, PAP Input and PAP Output. These processes will implement the access protocols of the individual networks to which the gateways are connected. In fact, we should think of four processes per network interface: PAP Input Level 2, PAP Input Level 3, PAP Output Level 2, and PAP Output Level 3. It is open for the present as to what the word "process" will actually mean in the implementation. Modules and interfaces should be clean enough so that we can freely intermix the possibly different Level 2 and Level 3 options of a given access protocol.

The PAP Input Level 3 module will pass each packet to a "Gateway Input" (GI) process for the particular access line (port) over which the packet was received. The GI process will have to decide. for each packet, which of four categories it falls into, and dispose of the packet on that basis. The four categories are:

a) Transit packets. These are packets for which this gateway is neither the source gateway nor the destination gateway. Such packets will be passed to a process which we might call "Dispatch" (see below).

b) Internet Control packets. These will be things like routing updates which need special processing by the gateway system. Such packets should be passed directly to whichever process or processes handle them. Thus routing updates might be passed to a process which manages the topological database and performs the SPF computation, while being simultaneously passed to all GO processes (see below) for flooding to other gateways. In some cases, where control packets need especially rapid processing (such as line up/down), GI might process the packets directly.

c) Internet Entry packets. These are packets from hosts which are entering the internet system at this point (or more accurately, those which are entering the

PARTICULAR internet system at this point). These should be passed to a process which we might call "Internet Arrival Process" (IAP).

d) Internet Departure packets. These are packets from hosts which will depart our internet system at this point. These should be passed to a process which we might call "Internet Departure Process" (IDP).

The GI process will tend to be responsible for actions at the level of the gateway-gateway packet headers, such as delay measurements, processing received piggybacked acknowledgments, etc. Similarly, there needs to be a Gateway Output process (GO) "above" every PAP Level 3 Output process, to set bits in the gateway-gateway header, etc. This allows the PAP modules to avoid doing things which are specific to gateway protocols, and hence preserves protocol layering. This mental organization of the gateway software might or might not map directly to an actual implementation.

Internal Pathway up/down considerations (i.e., which neighboring gateways are reachable over which access lines) will be handled by an interaction between GO and GI. Access Pathway up/down considerations (i.e., which hosts are reachable directly over which access lines) will be handled by IDP.

If a gateway is multi-homed to some network so that it has several access lines to that network, there are some subtleties. Each access line needs its own PAP Input and Output processes. We should take separate measures of delay to each neighboring gateway over each access line, as well as running a separate instance of the internal Pathway up/down protocol for each neighboring gateway over each access line. However. one possible approach is to have only one GO process and one GI process for each individual network to which the gateway is connected. The processes above GO and GI would not know whether or not a particular Pathway consisted of several access lines to the same network. The delay, congestion, and up/down information used by the routing algorithm would be reported by the GO or GI process for the composite set of access lines, and would be computed by these processes based on information passed up from the PAP processes. Dispatch (in the case of packets going further in the Internet) or IDP (in the case of packets leaving the Internet) would route packets to a particular GO process, leaving it up to the GO process to use its local knowledge to further route the packet to the most appropriate PAP Output process. The way in which GO computes routing information based on delays for each access line, and the way in which GO chooses from among several access lines to the same network, is something which needs further research.

There may be may be cases in which there are Pathways to a neighboring gateway through several distinct networks. This also requires further investigation.

The IAP process will look at the internet leader supplied by the source host, and use it to construct the internet header (there is a distinction between external leaders, used for protocol between host and gateway, and internet headers, used for protocol among gateways). IAP will also handle the flow control and connection block mechanisms. IAP may decide not to accept a packet, in which case it may create an error message or NAK and pass it to the GO process for the network from which the rejected packet was received. Such messages need not go out the same port or access line from which the rejected message was received. They should probably be returned by using the addressing information which can be gathered from the Pathway Access Protocol envelope, rather than by using the addressing information which can be gathered from the Internet Protocol envelope; hence the former information must be passed to IAP from PAP Input Level 3 (via the GI process). If IAP accepts the packet, it should be passed to the dispatch process.

The purpose of the Dispatch process is to decide what the packet's next hop is. That is, Dispatch looks up the next hop for this packet in the routing tables, which will be indexed by destination gateway and type of service. both of which can be found in the packet header. When the next hop is chosen, Dispatch must pass the packet to PAP Output Level 3 for transmission to the next gateway.

It is possible that Dispatch will find that the destination gateway is not reachable (this would be indicated in the routing tables). In this case, Dispatch should attempt a retranslation of the destination host address, to see if there is another destination gateway to which this packet can be sent. (This is possible if the destination host is multi-homed to a number of gateways.) If so, the header must be altered, and the packet passed to the appropriate PAP Output Level 3. If not, the packet can be discarded. It is not necessary to return any error message or NAK to the source host in this case.

We will also have some number of "internal hosts" ("fake hosts" in IMP parlance). These internal hosts will submit packets and receive packets to the gateway proper using the usual Internet Access Protocol, but no additional Pathway Access Protocol is needed. Hence the internal hosts can submit directly to IAP and receive directly from IDP. At least some of the internal hosts will implement higher level protocols for monitoring and control purposes. Others may deal in "raw messages" and not implement any higher level protocol. Certain internal hosts may require particular options that are not generally available to users, such as freedom from any congestion control constraints.

Some of our internal hosts will be used as "test nodes." That is, one way to test out the implementation of particular Pathway Access Protocols would be to implement both the host side (the PAP processes) and the network side (which would be in the test node). At its simplest, the test host could serve as a sort of software loopback facility. (Many network access protocols, unlike 1822, are non-symmetric, so they cannot be tested by simple hardware loopbacks.) If we are dealing in a connection-oriented network access protocol, the test host could participate in setting up connections. We could program the test host to send the gateway all of the strange control messages that networks are likely to send hosts during general operation, but which they never send when one is trying to test out host software.

In addition to these "event-oriented" processes, there might be any number of timer-based processes to handle various periodic or housekeeping functions, as well as lower priority background processes. The processes mentioned so far are not meant to be an exhaustive list.

This model of the gateway software architecture is based on the way the IMP is organized.

7.3.2 Inter-process Communication

The discussion has mentioned the act of one process "passing" a packet to another. To do this, the former process executes a system call which (a) places the packet on a queue for the latter process, and (b) causes the latter process to be scheduled (according to its assigned priority). This call should accommodate priority queuing. Sometimes processes may need to pass information to each other. This can either be done through commonly accessible data locations, or through the sending of messages.

With data moving among all these processes, we have a number of resource management problems, particularly buffer management. It is possible to provide some general principles of buffer management, but each new combination of protocols that we might have to deal with has to be considered individually, and its buffer management implications understood.

It is too early to produce a detailed buffer management specification at this time. but we expect the buffer management system to be very similar to that now being developed for the

IMP. That is, we might have a transit pool to hold those packets which move directly from GI to GO, and an end-end pool to hold those packets which move from GI to IAP or IDP. The buffers should also be organized into sub-pools to assure that each output device (GO process) can always obtain a certain number of buffers. Each input device (GI process) should also have a certain number of buffers assigned to it, to allow input with minimum latency. (We also have to save some buffers for internal host input, which bypasses the GI processes.) Since host input and output is much less tightly coupled in the gateway than in the IMP, we might want to separate the IAP and IDP pools entirely.

As packets move from process to process, we must make sure that no receiving process ever "blocks" the sending process; that is, all packet movement should be non-blocking. The receiving process should be ready to accept responsibility for a packet as soon as the sending process is ready to give it up. The receiving process can discard the packet, or queue it, or process it, or transmit it, as it sees fit, as long as this does not cause the sending process to back up. This sort of scheme is necessary to minimize interactions among processes. By extension, the gateways should not block the individual networks by refusing to take packets from those networks.

Of course any rule may have exceptions, but there should be a presumption in favor of the above principles. Such principles are difficult to adhere to unless each process that needs a pool of buffers is capable of getting a large enough pool to suit its needs most of the time; this assumes that the gateway will not be chronically buffer-short.

It is important to guarantee that all buffer management information is "globally" available. Buffer management should not be done invisibly by the system calls that move buffers from queue to queue; rather, the application needs full access to all information about the status of the buffer system. In particular, any process should be able to "charge" a buffer to any pool. For example. suppose a particular packet must move from process A to process B to process C, and that at the time process A sees the packet, it is already known that it will eventually get to process C. Process A should make sure that the buffer gets charged immediately to whatever buffer pool is associated with process C, and when process C finally receives the packet, it must have a way of knowing that the packet is already charged to its buffer pool, so that it does not get charged a second time (as in the IMP's "double counting" bugs). The need to make all this information accessible to all processes may be significant in the design of the system calls or subroutines that we use to move packets around; it certainly will be significant in the design of the buffer header formats. Concrete examples of the applications of these rules for completeness are described below.

The gateways should never block the networks by refusing to take packets from them. We can investigate how this applies in the case of the ARPANET. Each 1822 interface should have a buffer dedicated to it which is long enough to take an 8-packet message from the IMP. When a message is received over the 1822 interface, the PAP Input process should try to get a free buffer (of maximum message size) to put up immediately for the next input. If such a buffer cannot be obtained, then we should process the message ONLY if it is an internet control message (routing, up/down). If there are any piggybacked acks, or anything of that nature, we should process them. Then we should discard the message. and reuse its buffer for input. While this causes some loss of data, it is necessary so that we can still receive and process internet control messages, even when we are out of buffers for handling data messages. It is also necessary to prevent being declared tardy.

Since we do process internet control messages, there may be times when we cannot get enough buffer space to read in a full 8-packet message, but we can get enough buffer space to read in smaller messages. In this case we should still accept input from the ARPANET. If the input is too long, we should pull it through and discard it. This will at least enable us to receive control messages (which are likely to be relatively short). If this situation persists for any amount of time, the gateway which is sending to us over the ARPANET will end up thinking that the delays to us over the ARPANET are very long, and also that the Pathway to us is congested. This allows the routing and congestion control procedures to take effect and reduce the amount of traffic sent to us.

Another example: suppose that IAP sees a packet, and decides, by looking into the appropriate connection block, that it cannot send that packet yet because of congestion control restrictions. It makes sense for IAP to maintain a pool of buffers to use for holding such packets internally, until the congestion control procedure allows the packets to be sent through the Internet. However, when this IAP pool is filled, IAP must discard any further packets which it cannot send immediately. Any such packets must not be allowed to back up into the buffering needed to handle 1822 input.

Another example: we will want to limit the number of packets that can be queued to a single access line, so that one slow network does not utilize all the gateway's buffers, thereby preventing I/O from/to other networks. Further packets that arrive at the GO process will have to be thrown on the floor. If the packets are transit packets (from a neighboring gateway), this should cause the neighboring gateway to see a high delay to this gateway. This gateway should also reflect these discards in its determination of the congestion status of this access line. In the case where the discarded packet is an IAP packet, we should send a NAK to the host, IF we can get a buffer to do so.

A number of neighboring gateways may be reachable over a single network access line. That is, a gateway on the ARPANET may reach each of many neighbors over a single 1822 access line. In this case, we may want to reserve a certain amount of the buffering reserved for that access line for each neighboring gateway. Since the number of neighboring gateways may vary dynamically, this must be carefully implemented.

It is interesting to consider the extent to which this nodal software architecture can be considered to be "layered." This design attempts to make it possible to "mix and match" protocols at different layers more or less arbitrarily. That is, a process that handles a particular protocol should not need to know what protocols are in effect either above or below it. However, all processes must be cognizant of certain global or system-wide requirements and goals. In particular, every process may have buffer management responsibilities which accrue to it through its role in the system, and which therefore cannot be specified for that process when considered in isolation from other processes. Further, higher level protocol processes may need to know information (such as the source address of a packet in the local network) which is contained in the envelopes of lower level protocols. If we can produce an implementation following these guidelines, we have a chance of producing a system which preserves protocol layering but which does not introduce the inefficiencies and insularities that many excessively layered systems seem to have. However, producing this sort of implementation will be quite a challenge.

7.3.3 Measurements

The hierarchy of processes discussed above has implications for the way delay measurements are done and reported to the routing algorithm, and for the way up/down determinations are made.

The process structure relates to delay measurements as follows. Each GI/GO pair of processes "controls" the I/O to a particular network. The delay to EACH other gateway on that network must be computed separately for each access line to that network. This means, of course, that the GI/GO processes must be able to tell which access line a packet came from. A single value of delay to each neighbor must be computed from the delays on the various access lines. (For example. the single value of delay might just be the minimum over all access lines, or it might be a weighted average.) This generates a value of delay to each neighbor over the network "controlled" by that pair of GO/GI processes. This information must be fed up to some higher level process which computes a single value of delay to each neighbor, based on the delay values to that neighbor over the individual networks that can be used to reach it. It is this final value that will be put into the routing updates.

If a packet must be sent to a neighboring gateway and there are several different networks, or if several different access lines can be used to the same network, how do we decide which one to choose? This is not obvious and will require experimentation with a number of different algorithms. Possible selection criteria are:

1. Choose the network or access line which provides the

least delay. If we use this criterion, we will probably want to report the minimum delay of all the lines and networks as the Pathway delay to that neighbor. (Even though we will be sending all data on one access line to one network, we can use test traffic to measure delays on other access lines or networks.)

2. Round-robin the traffic among the access lines which have roughly the same delays (assuming that several lines have comparable delays, but several others have larger delays).

3. Split the traffic in proportion to the delays.

4. Use the shortest-delay access line or network until its queues exceed a certain threshold, then use the next shortest-delay line. etc. A variant: use the shortest-delay access line until the number of outstanding packets on it passes a threshold. This variant might be useful if we are dealing with HDLC lines, or in general with lines controlled by a low-level protocol which limits the number of unacknowledged packets that can be sent.

5. Base the decision on the type of service requested by the user.

Note that these criteria are by no means mutually exclusive; rather, we will probably use some combination depending on the circumstances or the configuration. Furthermore, some of these criteria require the higher level processes to be aware of the resource utilization of the lower level processes.

On occasion, a destination gateway may have to choose among several access lines or even several networks in deciding how to send a packet to a destination host. We would like to use the same sort of criteria we use in deciding how to send a packet to a neighboring gateway, except that we do not know the delay to the hosts. In some instances, the network will tell us its delay to a given host and we can use that as a guide. However, in such a case, we might want to plan for the contingency of the network giving us incorrect information.

Just as delay information needs to be passed up the levels of protocol and consolidated into a single quantity (or at least, fewer quantities than we had to begin with), the same is true of up/down information. First, the PAP Level 2 processes must make their up/down determinations for the individual access lines. PAP Lcvel 2 up/down information is generally restricted to saying whether the access line itself is up or down. That is, PAP Level 2 might be able to tell us that NO host or gateway at all is reachable over that access line, but will not be able to tell us that any particular host or gateway is reachable over that access line. Examples of PAP Level 2 up/down might be checking the Ready Line (1822) or performing the VDH line up/down protocol. PAP Level 3 up/down might also tell us that no node at all can be reached over the access line. For example. SATNET host access protocol has a line up/down determination at level 3 in addition to the VDH line up/down determination at level 2, and experience indicates that it is not at all unheard of for level 3 to declare the line down while level 2 declares it up. PAP Level 3 might also be able to say that particular destinations are not reachable over this access line. For example, an 1822 "destination dead" message might be received. Both the Level 2 and Level 3 up/down determinations would correspond to what are called "low level up/down protocols" in the IENs. The "high level up/down protocols" would be at the level of the internet protocol, and would be executed by GI/GO and by IDP (for neighboring gateways or hosts, respectively). Note that Pathway down determinations have to percolate up to higher level processes, so that the processes do not select networks or access lines which cannot reach a given destination. The processes must also AND together the down determinations of the processes "beneath" to determine whether a particular destination is completely unreachable.

When this determination is made, it must be reflected in a routing update (if there is no longer a Pathway to a particular neighboring gateway) or in the address translation tables (so that DNA messages can be returned for messages that cannot be delivered).

7.4 Measurements and Tools

7.4.1 End-End Traffic Measurements

In order to be able to configure the internet based on traffic requirements we will need to obtain some sort of end-end traffic matrix. In this section we like to consider what sort of measurements we need to get the best possible traffic matrix.

We plan to be able to count the traffic in units of internet datagrams, as well as in units of 16-bit words. For proper configuration of the system it is important to know the throughput requirement in both packets per second and bits per second, since these two measures are somewhat independent and encounter different bottlenecks throughout the system.

We would also like to know, for each source host, how much traffic it sends to each destination host, and what the requirements of that traffic are. That is, if there is a way for a host to request a particular type of service, we would like to measure how much of each type of traffic it sends to each destination host.

A more feasible (though less "precise") sort of end-end matrix could be computed on a "net-net" basis, rather than on a host-host basis. Each gateway can distinguish between ENTRY traffic (entering the Internet here, i.e., not being forwarded by any neighboring gateway), TRANSIT traffic (traffic coming from a neighboring gateway and going to a neighboring gateway), and EXIT traffic (traffic not going to a neighboring gateway). This distinction is possible since each gateway knows the local address of each of its neighbors, and can make the distinction by looking in the local leader as it comes off the input interface. Note that the same datagram can be both ENTRY and EXIT traffic at the same time, but something cannot be TRANSIT traffic at the same time as it is ENTRY or EXIT traffic. Each gateway can maintain sets of counters for each input interface. Each set of counters would correspond to the amount of traffic destined for some particular destination net which arrived over that input interface. Each set of counters should contain three words -- a single precision count of datagrams, and a double precision count of words. Periodically, each gateway should send all its counters to the collection point, zero the counters, and keep counting. When we collect and process all this data, we can produce a net-net traffic matrix. (Actually we get a somewhat more precise matrix, of "gateway interface"-to-"destination net".) This will be very interesting and will aid in understanding what is happening in the Internet.

As an extra source of information we could perform the dual measurement, i.e., count EXIT traffic at each output interface, by source net.

7.4.2 Internal Gateway Measurements

This section deals with what we might call "profile" measurements, i.e., measurements which provide a regular profile of gateway activity, performance, and resource utilization. The measurements described below are intended to be those that might be kept as a matter of course, and run continuously as part of the normal gateway code.

Queue Length Measurements

A measurement should be kept for EVERY queue in the gateway. For each queue, there should be the following three counters:

a) Total queue items -- each time an item is added to the queue, bump this counter by 1.

b) Accumulated queue length -- before adding an item to the queue, find the queue's current length, and add this to a counter.

c) Maximum queue length -- self-explanatory.

Dividing counter b by counter a gives a useful measure of average queue length. This is not the "average over time" (which in normal situations almost always turns out to be zero), but rather "the average number of items which a given item must wait for", or, more figuratively, "the number of items queued ahead of the average item." Counter c gives a rough and ready indication of the variance.

"Waiting Time" Measurements

This is similar to the queue length measurement, but is given in units of time. rather than queue items. When a packet enters the gateway, stamp it with its arrival time. When it reaches the front of its output queue, AND the gateway begins to transmit it to the network, compute the difference between the current time and the arrival time, and add this to a counter. Dividing this by the value of counter a above gives the average waiting time (time between arrival and transmission) of a packet.

This count should be kept in double precision. A maximum value should also be kept.

"Transmission Time" Measurement

This measurement indicates whether the gateway, when transmitting messages to the network, is being held off by the network. When the gateway begins to transmit a message out a network interface, take a time-stamp. When transmission completes, compute the total time spent in transmission.

This count should be kept in double precision. A single precision maximum value should also be maintained.

Routing Measurements

These measurements include:

a) a count of the number of routing updates generated by this gateway.

b) a count of the number of routing updates received from each "neighboring" gateway. (That is, keep a separate count for each neighbor.)

c) For each gateway in the Internet, a count of the number of times this gateway sees the Internet go down.

d) For each gateway in the Internet, total amount of time this gateway sees the Internet as down.

Interface Up/Down Statistics

These measurements include:

a) a count of the number of times each interface goes down.

It remains to be defined what "interface goes down" means for each network that might be connected to the gateway. For the ARPANET, probably we mean to count IMP ready line flaps.

b) an accumulated count of the total amount of time each interface is down.

Buffer Utilization

Whenever a request for a buffer of a particular kind is made, a counter is incremented. At the same time, the number of free buffers of that kind is added to a second counter. Dividing the second by the first gives the average number of available buffers of that kind per request. This provides a good indication of whether buffers are a scarce resource, on average.

Traffic Measurements

Messages sent out a particular network interface can be addressed either to another gateway on that network, or to some destination host on that network. Each of the following counters should be kept separately for each neighboring gateway reachable over an interface, and there should be a set of counters lumping in all the traffic going over an interface which will terminate at that network.

a) Number of data packets sent.

b) Number of data words sent.

c) Number of control (ICMP, GGP) packets sent.

d) Number of words in control packets sent.

e) Number of packets discarded because there was no room on the output queue.

f) Number of packets discarded because fragmentation was unsuccessful.

g) Number of packets discarded due to RFNM counting restrictions or similar network-specific flow control.

A similar set of measurements should be kept for each input interface, or rather, for each input interface there should be a set of counters for traffic from each neighboring gateway. as well as for traffic originating from the connection net:

a) Number of data packets received.

b) Number of data words received.

c) Number of control packets received.

d) Number of words in received control packets.

e) Number of packets discarded because there was no free buffer for the next input.

f) Number of RFNMs (or equivalent, e.g., SATNET ACCEPTs) received.

g) Number of DEADs received (or equivalent)

h) Number of INCs (or equivalent, e.g., SATNET REJECTs) received.

i) Count of packets dropped because of "net unreachable."

j) Count of packets dropped because of "host unreachable."

k) Count of packets dropped because of the time-to-live.

l) Count of packet dropped because of "parameter problem."

Generation of Control Messages

This measurement simply counts the number of each kind of ICMP message (including echo replies) generated.

Collecting This Data

This sort of data should be collected continuously under control of a NU monitoring system. We would specify a time-interval for each collection message. At the end of that interval, the counters would be packaged up and sent to NU in response to an HMP poll. When the counters are packaged up, they are then zeroed, and counting continues for the next interval. Note that every collection message will contain the data for a fixed time interval (which itself should be recorded in the message), so that no special initialization procedure is needed.

7.4.3 Tracing Facility

The ARPANET trace package is a means of gathering data about individual packets, as opposed to mere averages or totals. In considering the design of a trace package for gateways, there are three basic issues of concern: (a) what information should be gathered about each traced packet? (b) how should a particular class of packets be selected to gather this data? and (c) how should the data be collected?

Information to Gather

For each traced packet, we would probably like to know all of the following:

a) Everything about its IP leader.

b) Which interface it came in on.

c) Everything about its local leader on that interface.

d) Which interface it went out on.

e) Everything about its local leader on that interface.

f) Time packet was received at gateway.

g) Time gateway began transmission to network over output interface.

h) Time transmission was completed.

i) Time packet was "acked" by local network. In the ARPANET context, this means the time a RFNM, INC, or DEAD was received. In the SATNET context, this means the time an ACCEPT or REJECT was received.

Note that if a packet is discarded, we still want to collect its trace data; however, instead of output time stamps we need an indication that it was discarded and why.

Basically, this information tells us how the trip through a single gateway looks to a particular packet. It allows us to correlate the performance of each packet with almost anything about the packet that may be of any relevance, since we have all the relevant data about the packet.

Selection Criteria

a) Mask on any set of bits in the IP leader. That is, on a packet selected for tracing, we should be able to specify anything that might possibly appear in the IP leader, such as source host, destination host, type of service, a trace bit that we might define and make host-settable, or type of message (ICMP type, data, etc.).

b) Select all packets arriving over a particular input interface and leaving over a particular output interface (of course, we would have to specify a "don't care" value, so we could, for example, get all packets arriving over a certain interface but departing over any interface).

c) Mask on any set of bits in the input local leader and output local leader.

d) Gather information on discarded packets -- in this case the transmission time stamp should be replaced by a code indicating the reason for discard, and the ack time-stamp is unused.

We would probably generate too much trace traffic if we got a trace message for every packet satisfying certain criteria. We have to define a parameter N so that we get trace data about every Nth packet satisfying those criteria. We also want to make sure that we do not end up tracing the traces of trace data, which means we should probably mark trace data distinctively, in order to avoid tracing it.

Collection Procedure

In the ARPANET, each IMP has a fixed number of trace blocks allocated. When the decision is made to select a particular packet for tracing, a free trace block is obtained and associated with that packet. (If no free trace block is available. an overflow counter is bumped and that packet is not traced.) The appropriate fields in the trace block are filled in as the information becomes available. and the packet itself is marked to indicate that it is being traced. When the ack for the packet is received, the trace block is marked as finished. (This could be a little trickier to implement in the gateway than in the IMP, since the gateway does not hold the packets until acked.) Periodically, a scan is made of all the trace blocks to see whether any are finished. If so, the finished ones are sent (along with the overflow count, which is then zeroed) in a single message to the collection point. Presumably, the sending would be under the control of the HMP, which could poll for finished trace blocks, or else the finished trace blocks could be sent like traps.

Originally, the ARPANET trace package was intended to be able to trace a packet along its entire path, getting a trace block for the packet from each node that it traversed. In fact, it is hard to actually get all the trace data back, and difficult to collate all the traces of a single packet. Some traces always seem to be missing. The trace facility can be used to investigate the characteristics of intra-node delay on a per-packet basis. Tracing often reveals the large dynamic range and variability of the delay in a way which could not be revealed by averaging. It is also used to investigate long delay problems, by trying to see if the long delays in an IMP would correlate with some aspect of the packets experiencing the long delays.

7.4.4 Cross-Network Delay

One important facility will be the measurement of cross-network delays from one gateway to a neighboring gateway. To introduce delay-oriented routing into the Internet, we will first have to study the delay characteristics on a "gateway-to-neighboring-gateway" basis, and a measurement facility is an important first step.

We need some sort of "round-trip" measurement, which would enable us, insofar as is possible. to separate intra-gateway delays from delays experienced in the networks themselves. One approach is to mark certain packets as "special." and cause gateways receiving special packets to return replies for them to the gateway that sent them (i.e., to the gateway that was the previous hop, not the the "source gateway"). These replies would contain three pieces of data: (1) the time the the special packet to which this is a reply was received, (2) the time the transmission of this reply on its output interface was begun, and (3) an identifier for the special packet. Note that subtracting time (1) from time (2) gives the intra-gateway delay experienced by the reply.

A gateway which sends a special packet would keep a block of data for that packet. This block of data would contain the packet id, the time transmission out the output interface was begun, the time it was completed, the time a reply from the network (RFNM, DEAD, INC, SATNET REJECT or ACCEPT) was received, the time the reply from the neighboring gateway was received, and the intra-gateway delay of the reply (obtained by subtracting the two time-stamps contained in the reply). This data supplies good information about the cross-network round-trip time. and allows us to distinguish true network delays from intra-gateway delays. No synchronized clocks are needed.

When all the information is gathered, the block can be sent to some collection point. This is a slightly different sort of trace package than the one described previously, so the same collection mechanism or block allocation mechanism can be used.

In studying delay characteristics, it is important to be able to get samples of real user traffic, rather than just getting data about artificially-generated test traffic. In order to mark arbitrary packets as "special," we plan to steal a bit from the IP leader which we can set in any user packet (we might set it in every 10th packet, for example. though it is preferable to be able to get a random sample). This enables collection of round-trip data from real user packets. One issue is whether or not the gateway that receives a special packet turns off the "special" bit. We also want to be able to generate packets which already have the special bit turned on, either with message generators internal to the gateways or with message generators in ordinary hosts.

7.4.5 Message Generator

The ideal message generator would allow one to generate messages with an inter-message interval which varies at random, according to some probabilistic distribution. It would also allow one to vary the lengths of the generated messages according to some probabilistic distribution. Such generators should be included in the gateways, or in stand-alone or host-based traffic generators.

The message generators will be remotely controlled from the monitoring center. We include the ability to specify all fields of the IP leader that is to be carried in the generated message. but the message generator has to have built-in defaults for those fields which we do not specify during a particular run. In particular, we have to be able to set up the message generator to send any kind of ICMP message we choose.

The same is true of the "local" leader. We need to be able to specify all fields of the "local" leader which the generated message will carry. but defaults are also needed.

We need to choose a protocol number to indicate the "message generator" function of a gateway. Messages sent by the message generator will carry this number in the "protocol number" field of the IP leader, UNLESS we specify something else to go there. That is, this protocol number should just be a default.

Anything which arrives at a gateway and whose IP destination address is that gateway itself. but whose protocol number is the message generator number, will be discarded. This allows the message generator "fake host" (in IMP parlance) to double as a "discard fake host," and in particular ensures proper handling of echo packets sent by the message generator.

The message generator should keep a count of the number of IP datagrams it generates and the number it receives (when doubling as discard).

The message generator should NOT be turned on with a simple on/off flag, but rather with a timer. There should be NO way to turn the message generator on forever. It should always go off automatically after a certain time elapses; however, we do need flexibility in the amount of time specified. This is important for two reasons: first, it prevents a generator from running accidentally for weeks at a time. causing needless traffic; second, if we ever generate so much traffic that we cause severe congestion (which is a test we will probably want to do), we can be sure that the traffic will eventually stop even if we cannot get a "turn it off" message through because of the congestion.

Another issue is the inter-relation of the message generator with the buffer management. We must make sure that the message generator does not take the last buffer that is available for input from any real network interface; that is, we must continue to serve all the input interfaces. It would be interesting to know how many generated packets would not be sent because of lack of buffers. This suggests that the message generator should have two counters: one for "messages generated," and one for "messages actually queued for output." Suppose that a packet arriving over some input interface for the "discard fake host" is in the last available buffer for that interface. If the packet were really destined for some output interface, it would be discarded so that we could keep on doing input on all interfaces. If the packet were destined for this gateway's "discard," it would be discarded anyway of course, but perhaps we should not count it as a packet received by "discard." If it were a real packet, it would get lost here, so by not counting it we would get a more representative count of what real traffic would see. We should have two counts: one for the total number of packets arriving for "discard fake host," and one for the subset of those packets which would be discarded for buffer management reasons, even if they had not been destined for the discard fake host.

APPENDIX A. THE SIMULATION COMMAND LANGUAGE

This appendix describes how to run the simulator. It refers to version 1.0 (May 1, 1981) of the simulator. There are sections on creating a network, specifying traffic and routing matrices, producing simulation output. and actually running a simulation.

When you start up the simulator, it will type out a line giving the version number, then a prompt:

```
Arpanet Simulator Version 1.0, May 1, 1981
Command:
```

and then you can type commands. *Commands* are identified by their first word. A number of arguments follow the first word; these arguments may be numbers, words or filenames. For example:

```
Command: TRACEFILE MYFILE.TRA
Command: QUEUE 0 0 1 0 0
Command: IMP 12 TIMEOUT 0.001
```

The three commands shown are the TRACEFILE command, the QUEUE command, and the IMP command. Note that you do not type in "Command: "; it is printed out by the simulator. From now on, the examples will show only what you type. Most commands must fit on a single line; assume that this restriction holds except where you are told otherwise. Under certain circumstances, the IMP, HOST. LINE and MODEL commands can continue over several lines. This will be explained below.

The simulator ignores the distinction between upper and lower case; all input is first converted to upper case. Simula imposes the restriction that numbers should not end with a period. Use "1.0" instead of "1.". Simula also imposes the restriction that file names must be no more than 6 characters, with an extension no more than 3 characters.

A.1 Creating a Network

The simulation provides a number of commands for describing the network that you would like to simulate. The simulator allows you to set up a network consisting of a number of nodes, called IMPs, connected together by lines. Each IMP may have a number of hosts connected to it. Any host in the network can generate traffic for any other host in the network.

The simulation provides a way to specify, create, and modify IMPs, lines, and hosts. It also allows defaults to be set up in a convenient manner.

The first thing you must do when describing the network is to tell the simulator how large the network is. The INIT command specifies the number of IMPs and the number of lines:

```
INIT 64 162
```

This says that the network contains 64 IMPs and 162 lines. A line is a one-directional (or simplex) connection from one IMP to another. The maximum number of IMPs in the simulation is 100. This restriction is quite arbitrary. and can be changed fairly easily. There is only one other such restriction in the simulation; it has nothing to do with the size of the simulation. It will be described in the next section.

The general format of commands for defining and creating IMPs, hosts and lines is:

```
<command> <object-specification> <arguments>
```

The argument list is always optional. Objects are created when they are mentioned for the first time. The next time they are mentioned the argument list simply specifies changes in attributes (such as the error rate on a line). Unless otherwise indicated, you can change the attributes of a host, IMP or line at any time in the simulation.

For IMPs, you must give the IMP number:

```
IMP 53 <arguments>
```

except that when you mention the IMP for the first time you must specify the number of lines and number of hosts attached to the IMP:

```
IMP 53 3 5 <arguments>
```

which creates IMP 53 and specifies that it has 3 lines and 5 local hosts. For hosts, the host and IMP number must be specified:

```
HOST 2/53 <arguments>
```

which refers to host 2 on IMP 53. To connect two IMPs together one simply gives their numbers:

```
LINE 53 60 <arguments>
```

which refers to the line from IMP 53 to IMP 60. The line in the other direction, from IMP 60 to IMP 53, must be created and referred to separately.

Since the IMP has dozens of attributes, and many of them are the same for all IMPs, it makes no sense to force you to repeat every attribute for each IMP you create. You can set up default values for attributes of IMPs, hosts and lines by using models. You can think of a model in the sense of "model 316 IMP," but a model is really just a set of defaults. A model is created in much the same way as an IMP, host, or line:

```
MODEL <object> <number> <arguments>
```

where object is "IMP," "HOST," or "LINE," number is the model number, and the arguments specify the model defaults. For example:

```
MODEL IMP 0 <arguments>
MODEL HOST 3 <arguments>
MODEL LINE 2 <arguments>
```

Model IMPs are quite separate from model hosts and model lines. There can be up to 11 of each, numbered from 0 through 10. This is one of two arbitrary restrictions in the simulation. (The other is the maximum number of IMPs in the network, namely 100.)

Since the attributes of a model IMP are exactly the same as those of a particular IMP, the arguments you can give in a model IMP command are exactly the same as the arguments you give for a particular IMP. This is also true of model hosts and model lines. The attributes and arguments are described below.

Every IMP, host and line in the simulation has a model. The model can be specified explicitly when the IMP, host, or line is created:

```
IMP 53 2 5 MODEL 2 <arguments>
HOST 2/53 MODEL 1 <arguments>
LINE 53 60 MODEL 2 <arguments>
```

As shown, the model must be specified first. before any other arguments. The model should be specified when the IMP (or host or line) is created, and not otherwise. If it is not specified, model 0 is assumed. The effect of the model is to copy ALL the attributes from the model to the IMP, host, or line.

Every model also has a model. It can be specified explicitly:

```
MODEL IMP 2 MODEL 1 <arguments>
```

or it will default to model 0. The effect of the above example is to make model 2 exactly like model 1, except for the attributes of model 2 which are given in the argument list.

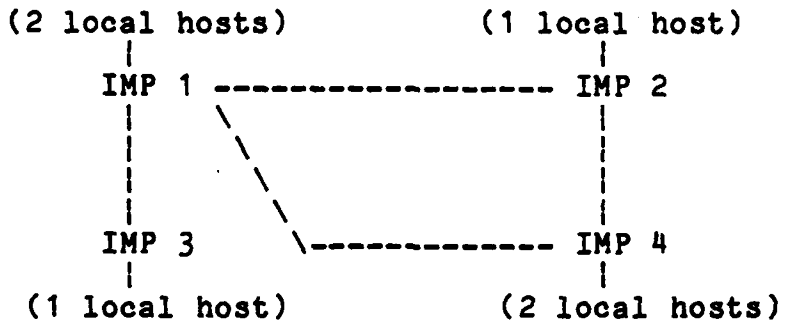
Since any unspecified attributes always default to those given in model 0, it makes sense to define model 0 immediately after the INIT command:

```
INIT 64 162
MODEL IMP 0 <arguments>
MODEL HOST 0 <arguments>
MODEL LINE 0 <arguments>
```

Then you should create any other models, then the IMPs, then the hosts and lines.

A.2 An Example Network

Suppose we want to create the following network:



Note that the connections between IMPs represent a line in each direction. This network has 4 IMPs and 8 lines:

```
INIT 4 8
```

Assume that the IMPs all have the same attributes, which have been given in a model statement; this is similar for hosts and lines:

```
MODEL IMP 0 ...
MODEL HOST 0 ...
MODEL LINE 0 ...
```

Then we can specify the topology with a series of IMP, host, and line commands which do not have to give any attributes. There are four IMPs; we must specify the number of lines and number of hosts for each:

```
IMP 1 3 2
IMP 2 2 1
IMP 3 1 1
IMP 4 2 2
```

Here all attributes will take the model 0 defaults. It is similar for the hosts. we just have to create them without any special attributes:

```
HOST 1/1
HOST 2/1
HOST 1/2
HOST 1/3
HOST 1/4
HOST 2/4
```

For the lines, we have to give the IMP on each end:

```
LINE 1 2
LINE 1 3
LINE 1 4
LINE 2 1
LINE 2 4
LINE 3 1
LINE 4 1
LINE 4 2
```

This completes the description of the topology.

A.3 Format of IMP, Host, and Line Arguments

The <arguments> section of the command is used to specify or change attributes of the IMP, host or line. There are dozens of attributes for the IMP, and a few each for the hosts and lines. The argument list is made up of a sequence of subcommands:

```
IMP 53 <subcommand> <subcommand> ...
```

Each subcommand has the general form:

```
<subcommand name> <subcommand arguments>
```

With a few exceptions, each subcommand has a single argument. Typically, they will refer to timing parameters, error rates, or output flags:

```
DEBUG ON
TIMEOUT 0.025
TASK 0.001
LUD 3 20 60
```

In the case of attributes of input or output processes, which may be different for different lines or hosts, an alternate command form can be used:

```
<subcommand name>/<index> <subcommand arguments>
```

where index is the neighboring IMP number that specifies which input or output process is to be affected. For example:

```
RETRANSMIT/60 1.6
```

The retransmit command sets the interval between retransmissions of an unacknowledged packet. The example above sets this time to 1.6 (seconds), but only for the output line (and process) which connects this IMP to IMP 60. The "/<index>" may be omitted, in which case the subcommand refers to every instance of the attribute in the specified IMP.

A.4 IMP Attributes

TASK t	task processing time
MODEMIN t	modemin processing time
MODEOUT t	modeout processing time
TIMEOUT t	timeout period
RETRANSMIT t	retransmission period
HOSTIN t	HostIn processing time
HOSTOUT t	HostOut processing time
RATE r	IMP clock runs at r times real time
FASTOFFSET t	IMP clock will start running at time t
DELAYOFFSET t	routing will start at time t
THRESHOLD t	initial threshold for delay is t seconds

DECAY t	threshold will decay t seconds each period
PACKET ON/OFF	turn packet tracing on or off
DEBUG ON/OFF	turn debugging output on or off
NODE ON/OFF	turn CPU utilization tracing on or off
QUEUE ON/OFF	turn queue length tracing on or off
LATENCY t	ModemIn latency (between packets)
SLOWTIMEOUT t	SlowTimeout processing time
FASTTIMEOUT t	FastTimeout processing time
DELAYAVGPERIOD	fast ticks between delay computations
LINEUDPERIOD	fast ticks between line up/down ticks
UPDATEAGINGPERIOD	slow ticks between updating aging
REROUTE t	rerouting processing time
SINK t	fake host input processing time
SOURCE t	fake host output processing time
DUMMY t	dummy background process processing time
MSGRATE r	message generator message rate: slow ticks/msg
PRIORITY p	message generator message priority
LENGTH l	message generator message length
DESTIMP i	message generator destination IMP
DESTHOST h	message generator destination host
DELUNITS x	units for quantized delay
AVGDELMAX x	maximum reportable average delay
AVGUNITS x	units for quantized average delay
WORDSIZE w	number of bits in average delay computation

PACKETLENGTH	maximum length b packets in bits
NUMBUFFERS n	number of buffers in IMP
HOSTDISCARD ON/OFF	vestigial: do not use

A.5 Host Attributes

SINK t	PacketSink processing time
PACKET ON/OFF	turn packet tracing on or off
QUEUE ON/OFF	turn queue tracing on or off
DEBUG ON/OFF	turn debugging output on or off
RATE d	message rate distribution
LENGTH d	message length distribution

A.6 Line Attributes

ERROR r	bit error rate is r
LAG t	propagation delay is t seconds
SPEED s	line speed is s bits/second
PACKET ON/OFF	turn packet tracing on or off
DEBUG ON/OFF	turn debugging output on or off
NUMCH n	number of channels in line protocol
LUD k n nup	line up/down protocol parameters
FRAMING b	hardware framing in bits

All times t, the error rate r, and the line speed s must be positive. Where ON/OFF is shown, the command must have ON or

OFF. The message rate and message length distributions can be either NEGEXPONENTIAL or DETERMINISTIC.

A.7 Global Network Parameters

In this section we will describe network parameters which are set with their own commands, rather than being set separately for each IMP, host, or line in a subcommand. Some commands have already been described; some are described in this section; some are described in the following sections on routing, files, and tracing and debugging.

SEED i

Set the value of the random number seed to i. Except in exceptional circumstances, this command only makes sense before any other commands (e.g., immediately before the INIT command). Do not use this command AT ALL unless you are SURE you know what you are doing. The default value is 314159.

UPDATE 1
LINEUPDOWN 1
NULL 1

Set the length of routing updates, line up/down protocol or null packets to 1 (bits). This does not include protocol overhead or hardware framing. This may be set or changed at any time during the simulation. The default value is 0.

OVERHEAD n

Protocol overhead is n bits per packet. This is the same for the whole network. The other sort of overhead is hardware framing, which is set individually for each line by the FRAMING command. When the simulation reports the length of a packet, the protocol overhead is included. The only thing in the simulation which is affected by packet length is the transmission time on the lines. For this purpose, the packet length (including overhead) is taken from the packet and the framing from the line, and the two are added together to give the total length.

A.8 Routing

Routing is the procedure used to determine the path which a packet takes from its source to its destination. The simulation allows three possibilities: fixed, random, and shortest path first (or SPF). SPF routing, the algorithm used in the ARPANET, is the default for the simulation; it is described in Sections 4.13 through 4.15.

The SPF routing protocols can be run in either of two modes: enabled or disabled. The simulator starts with SPF routing running disabled; it may be enabled using:

SPFROUTING ON

When the protocol is running disabled, the average delay out each line is still computed, and updates are still flooded through the network, but IMPs do not recompute their forwarding tables when an update arrives (or is generated). Thus, although delay information in each IMP is kept current, the routing is fixed. This is useful at the beginning of the simulation because, otherwise, the algorithm will use the minimum hop route. In the case of traffic loads near saturation, this choice of routes may overload one or more lines, probably causing the routing algorithm to oscillate, a condition from which it may never recover. The solution is to initialize the IMPs' forwarding tables to some set of routes which do not cause a line to overload (using the "ROUTE" command, described below), run traffic through the network for one or more delay averaging periods with SPF routing running disabled, then enable the protocol. Running traffic through the network with the protocol running disabled allows accurate average delay information to be accumulated without the danger that routing will be affected by incomplete or transient data.

There are two sorts of fixed routing: single-path and multiple-path. Single-path routing means that all packets at a given IMP for a given destination are routed the same way. Note that this applies to all IMPs in the path, not just the source. You must indicate that you want fixed routing, and also specify the routing matrix, which for each IMP gives the next IMP in the path to a given destination. The routing specified in our example network is shown below:

```
FIXEDROUTING
ROUTE 1   0 2 3 4
ROUTE 2   1 0 1 4
ROUTE 3   1 1 0 1
ROUTE 4   1 2 1 0
```

The command "SROUTE" is synonymous to "ROUTE." Note that the diagonal of the matrix is all zeroes. The simulator is clever enough to realize that once a packet IMP i gets to IMP i, routing is no longer needed. It does not matter what you put in those entries of the matrix, by convention they are set to zero.

Consider the line which reads:

```
ROUTE 4   1 2 1 0
```

and refer back to the diagram of the network. This says that when a packet reaches IMP 4, it can be forwarded as follows:

destination	forwarded
1	directly to 1
2	directly to 2
3	via 1
4	- doesn't matter -

Multiple-path routing means that a fraction of the packets at a given IMP for a given destination are routed to each of that IMP's neighbors. This means that for a given source-destination flow, each of a number of different routes will carry a fraction of the flow. A single entry in the routing table specifies the fraction going to each neighbor for a particular IMP and destination. For example:

```
MROUTE 4 1  (1 0.5)  (2 0.5)
```

in our example network means that of the packets at IMP 4 which are destined for IMP 1, half should be routed directly to 1, and half routed via IMP 2. Note that to complete IMP 4's routing table, we would also have to specify the splitting fraction for packets to IMP 2 and IMP 3. In practice, not all flows will split at every node, so it is possible to define the network routing by giving a single-path routing matrix for flows which do not split, then giving MROUTE commands for those that do. It is legal to combine SROUTE and MROUTE commands in any order, and the last one mentioned overwrites any preceding definitions for each IMP and destination. (That is, an MROUTE command defining the routes for a particular IMP and destination will not affect the routes for that IMP and other destinations.)

Random routing simply means that when a packet arrives at an IMP, if that IMP is not its destination, it is forwarded to a neighbor of that IMP chosen at random, possibly back where it came from. To set random routing, simply give the command:

```
RANDOMROUTING
```

and probability will take it from there. This form of routing is easier to analyze.

If you use SPF routing, it is useful to be able to find out what routing is being used in the network. The command:

```
PRINTROUTING
```

causes the routing tables in every IMP to be printed out on the debugging file. This table is in the same format as you would use to input the routing for fixed routing using ROUTE commands. The PRINTROUTING command may be given at any time. If single-path routing is in use, you will get back exactly what you put in; if random routing or multiple-path routing is in use, the output will make no sense at all.

If you want the routing table in just one IMP, give an argument to PRINTROUTING. For example:

```
PRINTROUTING 10
```

will print out only the routing table in IMP 10.

A.9 Disabling Protocols and Buffer Limits

In Section A.8 we described various options for the routing protocol. In this section we describe how to disable the IMP-IMP reliable transmission protocol and buffer management limits. The intention is that the simulator have two different operating modes: realistic, and analytic. In the analytic operating mode, the above protocols are disabled, making the simulator roughly comparable to analytic models, which cannot handle adaptive routing, or buffer and line protocol limits.

It is a feature of the present buffer scheme that the number of buffers allocated to the store-and-forward function depends only on the number of lines attached to the IMP, not on the total number of buffers available. Thus, the number of store-and-forward buffers will be limited even if the number of buffers in an IMP is set to a very large number. The command:

```
NOBUFFERLIMIT
```

sets the buffer limit for each function in every IMP to a very large number.

The reliable transmission protocol which runs between each pair of (connected) IMPs imposes a limit on the number of packets which can be waiting for acknowledgment in the sending IMP. It also includes acknowledgment of packets which have been accepted by the receiving IMP, and retransmission by the sending IMP if a packet is not acknowledged within a certain time. To effectively remove the limit on the number of unacknowledged packets, the limit can be set to a large number:

```
LINE 1 3  NUMCH 100
```

However, 100 is too small a number to rule out the possibility that the limit will occasionally be reached, but too big to allow every line in the simulation to use this limit without running out of space for the necessary data structures. In addition, this approach does not eliminate the processing and line bandwidth caused by acknowledgments and retransmissions. A line subcommand is therefore provided to disable the protocol completely:

```
LINE 1 3  LINEPROTOCOL OFF
```

This has the effect of stopping acknowledgments and retransmissions, and removing the limit on the number of unacknowledged packets. In order to save space, the number of channels on the line should be set to 1:

```
LINE 1 3  LINEPROTOCOL OFF NUMCH 1
```

This command must be given separately for each direction on each line (or the MODEL default mechanism must be used when the line is created).

A.10 Network Traffic

Traffic is generated in the hosts in the form of messages. A message is simply a given number of bits of data, from a particular host and IMP, to be sent to a particular host and IMP at a specified priority level. When the host that generates a message gives the message to its IMP, the message is broken up into a number of data packets. Each packet is sent through the network independently and delivered to the destination IMP and host. A message may be any length; the maximum length of the packet is an attribute of the IMP. It can be set separately for each host.

A message flow is simply a sequence of messages from a single IMP and host, to a single IMP and host, with a specified priority. There is no limit on the number of message flows which you can start. You must specify the source and destination, the rate at which messages are to be generated, and the average message length. The priority is optional; it is defaulted to 4. Unless you are a sophisticated user, you should only use priorities numerically greater than or equal to 3. Note that priority 3 is higher than priority 4.

Message flows are initiated by using the START command. For example:

```
START 2/1 1/4 1.5 2000 5
```

starts a flow from host 2 on IMP 1 to host 1 on IMP 4, at an average rate of 1.5 messages per second, and an average length of 2000 bits, at priority 5. The time between messages may be either fixed or random. Similarly, message lengths may be either fixed or random. The only random distribution which is implemented by the simulator is the negative-exponential distribution. The choices are specified using host subcommands: RATE for message rate, and LENGTH for message length. The two possible values are DETERMINISTIC and NEGEXPONENTIAL. For example, to specify messages arriving at fixed time intervals with randomly chosen lengths, you type:

```
HOST 2/1 RATE DETERMINISTIC LENGTH NEGEXPONENTIAL
```

Although you can have many message flows from the same host, with different average values, the distribution must be the same.

For certain analytic models, or to test the network, it is convenient to generate uniform traffic from each IMP to every other IMP. The command:

```
UNISTART r l p
```

starts equal message flows from each IMP to every other IMP, at a total rate of r messages/second, with an average length of l bits. The argument p specifies the message priority: it is optional -- the default is 4.

Each flow from IMP i to IMP j is from host 1 on IMP i to host 1 on IMP j. A subsequent UNISTART or START command will start a flow or flows separate from these flows.

It is not possible to change message flows once they have been started. To get that effect, you should stop the message flows and restart them with the new values. To stop a message flow, use the STOP command:

```
STOP s d
```

which stops all message flows from IMP s to IMP d. Both s and d are optional: if d is omitted all flows from IMP s are stopped;

if both s and d are omitted all flows in the simulation are stopped. Note that you cannot specify individual hosts; this command stops the message flows to or from all hosts on the specified IMPs.

A.11 Files, Input and Output

The simulation uses standard Simula input/output facilities, which are very bad. This section will describe how to fight the system and not lose.

The simulation uses three output files: a file for echoing input, and files for tracing and debugging output. The echo file is simply a file to which the simulator echoes all input; tracing and debugging are described in the following sections. The files can be changed using the commands:

```
ECHOFILE test.ech
DEBUGFILE test.deb
TRACEFILE test.tra
```

Filenames must be less than six characters with three-character extensions. The default for each output file is "NUL:", which discards all output.

At any particular time, the simulator is reading commands from a single input file, but input files may be nested. If you give the command:

```
READ test.dat
```

the simulator leaves the current input file and begins reading commands from "test.dat." If test.dat contained another READ command, the simulator would switch input files again. When the simulator gets to the end of an input file, it finishes the READ command and continues with the next command.

Simula allows ordinary file input/output to and from the terminal, but with some restrictions. It does not seem possible to close the terminal and save the program so that it can be restarted later. Simula will not allow several output streams (e.g., both tracing and debugging output) to be directed to the terminal. For these reasons, the simulator does all terminal input/output via a file called "ME:". For the simulator to work at all, you must define this file to be the terminal (which is called "TTY:"):

```
@DEFINE ME: TTY:
```

This is a command to TOPS-20, not to the simulator. You must give this command to TOPS-20 before running the simulator. Having done that, you can now use "ME:" whenever you want to refer to the terminal. It is possible to have more than one input stream, or more than one output stream refer to the terminal in this way.

When you direct output to a particular file. using TRACEFILE, DEBUGFILE or ECHOFILE. a new version of the file is created. When you specify a particular file for input. using READ, the current version of the file is used.

The output files specified in TRACEFILE. DEBUGFILE and ECHOFILE must all be different, except that more than one of them can be "NUL:" or "ME:".

A.12 Controlling Tracing and Debugging

Tracing output. usually just a string of numbers, is generated by the simulation for analysis by the statistics package. This package reads a file of numbers and computes such things as means and confidence limits. The tracing output is designed to be easy to process rather than easy to read. A typical piece of trace output might be generated when a packet is delivered to its destination host. and contains information on the route taken by the packet and the time taken to cross the network. A list of the tracing output currently implemented is given below.

Debugging output. on the other hand, is designed to be easily readable. It contains English text as well as numbers. It can be used to produce a step-by-step account of the progress of every packet through the simulator, or to check on the correct operation of the protocols. Debugging output is provided for every process, and is generated both when the process wakes up and discovers something to do, and when it has done it and is about to go to sleep. A list of the debugging output currently implemented is given below.

Tracing and debugging output are generated by identical. parallel mechanisms. There is a file for tracing output. specified by the TRACEFILE statement, and a file for debugging output, specified by the DEBUGFILE statement. Each IMP, host, and line has 4 flags which turn tracing on and off (controlled by the PACKET, BUFFER, QUEUE and NODE subcommands), and also a flag which turns debugging on and off (controlled by the DEBUG subcommand). There are global flags for each sort of packet and queue tracing; these are specified by the PACKET and QUEUE commands. If an event occurs in an IMP, host, or line which has its local tracing flag set, tracing output is generated; however, this will occur only if the global tracing flag for that event is set. That is, output is generated only when both the local flag and the global flag are set. The exception to this is tracing of buffer allocation and node CPU utilization: these two traces have local flags in each IMP but no global flag(s) -- in that sense they are always "on." There is a similar set of global flags for debugging output; this set is controlled by the DEBUG command. A list of flags is given below.

The default file for both tracing output and debugging output is the device "NUL:", which discards all output. The initial value of all flags is off. Thus, in order to get any tracing output at all, one must specify an output file by using the TRACEFILE command, set the global flags for the events one is interested in tracing by using the PACKET or QUEUE command, and set the local flags in some number of IMPs, hosts, or lines. Note that one can use the model statement to change the default value of the local flags from off to on. In order to get debugging output, one must use the corresponding commands, DEBUGFILE and DEBUG.

The file for output, the global flags, and the local flag in each IMP, host, or line, may all be set or changed in the middle of a simulation run. The new setting will take effect immediately (i.e., at the simulation time at which the simulation returned to command level).

Note: In the tracing and debugging output it is neccessary to be able to refer unambiguously to IMPs and hosts. Unfortunately, IMPs are numbered consecutively from 1, and h sts are numbered consecutively from 0 (host 0 refers to the so-called fake host, which is simulated by the IMP itself). Therefore, when host numbers are output as part of tracing or debugging, they are output as negative numbers. The fake host, host 0, is still 0. Thus 1 refers to IMP 1, 0 to the fake host, and -1 to host 1 (on some IMP).

A.13 Tracing and Debugging Flags

In each IMP, debugging can either be on or *off*. It is controlled by the DEBUG subcommand. For example:

```
IMP 1 DEBUG OFF

IMP 2 DEBUG ON
```

will turn debugging off in IMP 1 and on in IMP 2. Similarly for hosts and lines:

```
HOST 1/2 DEBUG ON

LINE 5 3 DEBUG ON
```

Tracing in the IMP is controlled by 4 separate flags: packet tracing, node tracing (for CPU utilization), queue tracing and buffer tracing. The corresponding subcommands are PACKET, NODE, QUEUE and BUFFER. For example:

```
IMP 1 PACKET ON

IMP 2 NODE ON

IMP 3 QUEUE ON BUFFER OFF
```

Lines have packet tracing only, and hosts have packet and queue tracing. For example:

```
HOST 1/2 PACKET ON QUEUE OFF
LINE 5 3 PACKET OFF
```

As usual, the MODEL command can be used to set default values for these flags before the IMPs, hosts or lines are created.

Each piece of debugging output also has a global flag, which indicates whether that particular piece of output will be generated. These flags are set all at once by the DEBUG command (not subcommand). The flags are specified by index:

1	Task process (IMP)
2	Modem input process (IMP)
3	Modem output process (IMP)
4	Host input process (IMP)
5	Host output process (IMP)
6	Timeout process (IMP)
7	Input process (line)
8	Output process (line)
9	Delay measurements (IMP)
10	Routing updates (IMP)
11	Routing table (IMP)
12	Message output process (host)
13	Packet input process (host)
14	IMP-to-IMP link protocol (IMP)
15	Line up/down protocol (IMP)
16	Buffer Management (IMP)

17 Background (IMP)

18 Routing changes (IMP)
19 Congestion control (IMP)

For example, the command:

```
DEBUG 0 0 0 1 1 0 0 0 0 0 0 1 1
```

would set flags 4, 5, 12 and 13 so that debugging output would be generated for host input and host output in any IMP or host whose local debugging flag was on.

Packet and queue tracing (but not node or buffer tracing) are also controlled by global flags, set by the PACKET and QUEUE commands. The packet flags are:

1 end-end delay, route etc. (host)
2 node delay
3 null packets
4 discarded packets
5 line utilization (line)
6 changes in line protocol state.

For example, the command:

```
PACKET 0 1 0 1
```

would turn on tracing of node delays and discarded packets.

The queue flags control which IMP and host queues are traced.

Details of tracing and debugging output are given in the following sections.

A.14 Debugging Output

Most of the debugging output is in a standard form:

```
prccess  i [ j ] time  n [ m ] event [ other ]
```

Square brackets mean that the field is optional. Process is a tag for the debugging record. The tags are as follows:

```
TASK      MODEMIN   MODEMOUT  HOSTIN    HOSTOUT
TIMEOUT   LINEIN    LINEOUT   DELAY     UPDATE
FORWDING  TO IMP    FROM IMP  LINE PR.  LINE U/D
BUFF.MGT  BACKGRND  ROUT.CHG  CONG CTL
```

The next two fields give the IMP number, and either another IMP number, or the host number. Time is the current simulation time. The next two fields describe the packet being processed. If there is no such packet, the field contains "[]". For data packets, n is the packet number and m is omitted; for update packets, n is the originating IMP and m is the serial number; for hello's and IHY's, n is either "HELLO" or "IHY" and m is either "UP" or "DN." Event is a descriptive field; a packet may be logged as it is taken off a queue. or forwarded to another process, or dropped. In each case the event field describes exactly what happened. The final fields. if present, give such information as network delay or logical channel number.

The exceptions to the above format are delay (9), routing table (11), routing changes (18), and congestion control (19). The fields up to the time are as described, and the trailing fields are as follows:

delay	the exact and rounded delay out each line
routing table	the routing table
routing changes	the number of the previous and current next hop IMP
congestion control	the previous and current path congestion levels and queue limits

A.15 Tracing Output

There are four separate types of trace output: packet, node, buffer, and queue tracing. They are controlled by separate trace flags, both globally and in each host, IMP, and line. All output goes to the file given in the most recent TRACEFILE command.

Packet tracing traces the progress of a packet through a node or the network. It is turned on and off by the global command PACKET, and the host or IMP subcommand PACKET. Packet flag 1 controls tracing of network delay. If it and the host packet tracing flag are set when a packet is accepted by the destination host, the following will be output (the numbers in brackets are column numbers):

(1)	the trace type (i.e., 1)
(6,9)	the destination IMP and host numbers
(13)	the current time
(25,28)	the source IMP and host numbers
(31)	the priority
(34)	the time the message was created
(46)	the time the message entered the network
(58)	the total delay
(66)	the network delay
(75)	the packet length
(81)	the total number of retransmissions of the packet
(86)	the number of IMPs in the route taken
(89)	the route taken by the packet

Note that the times given for delay are slightly different when delivering to real and fake hosts. For a real host, the times do not include processing by the host; for fake hosts, processing by the fake host background routine is included.

Packet flag 2 controls the tracing of individual node delays. If it and the IMP packet tracing flag are set when a packet is acknowledged or accepted by a host, then the following will be output:

(1)	the trace type (i.e., 2)
(6)	the IMP number
(9)	the neighboring IMP number or local host number
(13)	the time (of acknowledgment by neighboring IMP

or delivery to local host)
(25,28) the source IMP and host numbers
(32,35) the destination IMP and host numbers
(39,42) priority and packet number
(50) the time at which the packet entered the node
(62) the node delay (includes transmission time and propagation delay, but only for packets sent to neighboring IMPs, not local hosts)
(74) the number of transmissions before acknowledgment (NOT the number of transmissions before successful receipt)

The last field is only given for packets sent to a neighboring IMP, not packets delivered to a local host.

Packet flag 3 controls the tracing of null packets. If it and the IMP packet tracing flag are set when a null packet is sent, the following is output:

(1) the trace type (i.e., 3)
(6) the IMP number
(9) the neighboring IMP number
(13) the time

Packet flag 4 controls the tracing of discarded packets. If it and the IMP packet tracing flag are set when a packet is discarded, the following will be output:

(1) the trace type (i.e., 4)
(6) the IMP number
(9) the host or other IMP number or 0
(13) the time
(25) the packet number or 0
(33) the reason for discarding the packet

The third field would be 0 if, for example, the packet was discarded by TASK. The packet number would be 0 if an incoming message is dropped before being given a packet number, or if the packet is not a data packet, and hence has a packet number of 0.

The possible values of the reason field are:

1	Unacceptable update (ProcessUpdate)
2	Duplicatepacket (TASK)
3	No reassembly buffer (TASK)
4	No line availabe (TASK - random routing)
5	No store-and-forward buffer (TASK)
6	Inaccessible destination (TASK or HostIN)
7	No channel (TASK)
8	Not ready (ModemIn)
9	Line error (ModemIn)
10	Discard bit set (ModemIn)
11	No buffer (ModemIn)
12	Timeout (HostIn)
13	Congestion control (Task)
14	Reset state (ModemInterface)

Packetflag 5 controls the tracing of line utilization. If it and the line packet tracing flag are set when a packet starts or finishes being transmitted on the line, the following will be output:

(1)	the trace type (i.e., 5)
(6)	the source IMP number
(9)	the destination IMP number
(13)	the time
(25)	1 (for start), or 0 (for finish)
(30)	the packet length

Packet flag 6 controls the tracing of changes in the line protocol state. If it and the IMP packet tracing flag are set when the line protocol for one of the IMP's lines changes state, the following will be output:

(1)	the trace type (i.e., 6)
(6)	the IMP number
(9)	the neighboring IMP number
(13)	the time
(25)	0 (down) or 1 (up)
(28)	the protocol state (0-4)

Node tracing traces which process is executing on a given CPU. It is controlled only by a local flag in each IMP, set using the NODE subcommand. If it is set, every time a process starts executing on the CPU, the following will be output:

(1)	the trace type
(6)	the IMP number
(9)	the time

The trace type is 2000 plus the priority level of the process. In general, each process has its own priority level. An idle CPU is indicated by a priority of 999.

Buffer tracing traces buffer allocation in a given IMP. It is controlled only by a local flag in each IMP, which is set by using the BUFFER subcommand. If it is set, every time an attempt is made to allocate a buffer, or when a buffer is freed, the following is output:

(1)	the trace type
(6)	the IMP number

(9) the change in allocation
(13) the time
(25) the number of free buffers
(30,36,40,...)
the number and maximum number of buffers allocated in each buffer type.

The trace type is 3000 plus the buffer type. The possible buffer types are:

1 Reassembly
2 Store and Forward
3 Uncounted.

The change in allocation field is not entirely accurate, since it is given as +1 for a successful allocation, 0 for an unsuccessful allocation, and -1 for a buffer being freed. Since no account is taken of the *previous* buffer type of a buffer. a successful allocation need not increase the allocation.

Queue tracing traces the length of queues in the IMP and host. It is controlled in each IMP and host by a local flag which is set by the QUEUE subcommand, and by an array of global flags which are set by the QUEUE command. Six queues are traced:

1 task ordinary queue
2 task special queue
3 modemOut ordinary queue
4 modemOut special queue
5 hostOut ordinary queue
6 messageOut queue (host)

The ordinary queues contain data packets or messages; the special queues contain routing update packets and line up/down protocol packets. The messageOut queue in the host contains messages waiting to be submitted to the network. If the local IMP or host flag and the appropriate global flag is set when a packet arrives at or is removed from a queue, the following is output:

(1)	the trace type
(6)	the IMP number
(9)	the host or other IMP number or 0
(13)	the time
(25)	arrival/removal flag
(29)	the queue length

The tracetype is 1000 plus the flag index given above. The arrival/removal flag is +1 for arrivals and -1 for removals. The queue length is measured after the packet has been added or removed.

A.16 Tracing Average Line Utilization

If you are interested in tracing line utilization, you can use trace flag 5, which reports the beginning and end of each packet on each line. However, this generates a very large amount of output. Usually, you would not be interested in the timing of every single packet on a line, but rather would want to observe how the "average" line utilization changes over time. For this purpose, the simulation provides a way to trace line utilizations averaged over a specified interval:

```
AVERAGELINEUTILIZATION p
```

This command causes the utilization of every line to be averaged and reported every p simulated seconds. The information is written to the current trace file in the following format:

(1)	the trace type (i.e., 7)
(6)	the sending IMP number
(9)	the receiving IMP number
(13)	the current time
(25)	the average line utilization

Note that a single packet may be split between averaging intervals. The choice of averaging interval p is not particularly critical. Too long an interval will tend to increase confidence limits, but less output will be generated. Because the data is highly correlated, a shorter interval will increase the amount of output generated without necessarily decreasing confidence limits. We have found that an averaging interval of 1 second produces reasonable results.

A.17 Running the Simulation

Almost all the commands in the simulator simply describe the network which is to be simulated, or the traffic which is to be generated. They do not actually DO anything. In particular, in the simulated network, time does not pass. To make time pass in the simulation, use the RUN command:

```
RUN t w
```

which runs the simulation for t seconds of simulated time. The argument w is optional; if it is present, a message is printed every w seconds of simulated time to indicate that the simulation is progressing.

You can repeat the RUN command as many times as you like. The simulation will proceed each time. For example:

```
RUN 10.0
```

is the same as:

```
RUN 5.0
RUN 5.0
```

You can change IMP, host, or line attributes, or traffic patterns, at any time:

```
RUN 1.0
LINE 53 60 ERROR 5.0E-4
RUN 2.0
STOP 1 2
START 1/1 1/2 5.0 100
RUN 7.0
```

This sequence of commands will also cause 10.0 seconds of simulated time to elapse, but of course the result will be different.

To stop the simulation, and exit the simulator, give the QUIT command:

```
QUIT
```

This command closes all files and returns to TOPS-20. You can give this command in a command file. even a nested command file. in which case ALL input files are closed and any subsequent commands are ignored.

The simulator provides a command to save the state of the simulation in a file. If you give the command:

```
DUMP f
```

the simulation will be saved in the file f. All current input files are closed, and commands in command files after the DUMP command are ignored. When the DUMP command has completed, the next command will be read from the terminal. All current output files are closed and reopened, and the simulator continues as though nothing had happened. Here. f is NOT restricted to 6 characters with a 3-character extension. Since the file produced is very large. only two versions of f are kept. and the directory is expunged each time.

The simulator can be continued from the file by typing:

```
RUN f
```

to TOPS-20 (not the simulation). Since the debugging, tracing, and echo files are closed before the program is saved, and reopened in append mode. if you continue the simulation from f they will only be correct if either:

a) you QUIT the simulator immediately after the DUMP command, or

b) the system or simulation crashes before the next DUMP command.

That is, you can always continue from the last DUMP file after a system crash. but if you want to continue after a QUIT, you should DUMP immediately beforehand.

If you are running the simulator for a long time. it is advisable to save the state of the simulation regularly. The sequence of commands:

```
RUN t
DUMP f
 ...
RUN t
DUMP f
 ...
RUN t
```

```
DUMP f
QUIT
```

will ensure that the saved program is no more than t seconds of simulated time out of date. The simulator can be restarted from f if the system crashes, or if the simulator completes successfully but you discover you need a longer simulation run.

Because of a problem in closing and reopening the terminal. the SIMULA debugger may not work on a program saved by DUMP. This means that if you are running a program saved by DUMP and the simulator encounters an error, it will print an error message and then may die trying to enter the debugger. You will be deposited back at TOPS-20. There is no problem with using the terminal from the simulator itself -- see Section A.11, Files, Input and Output.

A.18 Examples

This section simply contains examples of the complete input for some runs of the simulator. Suppose we have put the following model definitions in a file "model.dat":

```
MODEL IMP 0 (
   MODEMIN 0.00030          MODEMOUT 0.00031
   HOSTIN  0.00053          HOSTOUT 0.00035
   TASK     0.00084         FASTTIMEOUT 0.001
   SLOWTIMEOUT 0.001        REROUTE 0.001
   DUMMY 0.007              SOURCE 0.001
   SINK 0.001

   TIMEOUT 0.0256          DELAYAVGPERIOD 400
   LINEUDPERIOD 25         UPDATEAGINGPERIOD 12

   RETRANSMIT 0.125        WORDSIZE 16
   THRESHOLD 0.064         DECAY 0.0128
   DELUNITS 0.0064         AVGUNITS 0.0064
   PACKETLENGTH 1008       NUMBUFFERS 40
   FASTOFFSET 0.0          DELAYOFFSET 0.0
   RATE 1.0                AVGDELMAX 1.6384

   PACKET OFF         BUFFER OFF        QUEUE OFF
   NODE OFF           DEBUG OFF
   )
MODEL LINE 0 (
   ERROR 0.0          SPEED 50E3        LAG 1.0E-3
   PACKET OFF         DEBUG OFF         FRAMING 72
LINEPROTOCOL ON
   NUMCH 8            LUD 3 20 60
   )
MODEL HOST 0 (
   SINK 0.0           PACKET OFF        DEBUG OFF
   LENGTH NEGEXPONENTIAL   RATE NEGEXPONENTIAL
   )
```

If we wish to debug the line protocol. we only need a network with two nodes:

```
INIT 2 2
READ model.dat
IMP 1 1 1
IMP 2 1 1
LINE 1 2
LINE 2 1
HOST 1/1
HOST 1/2
FIXEDROUTING
ROUTE 1  0 2
ROUTE 2  1 0
START 1/1 1/2 20.0 1000
DEBUG 1 1 1 0 0 0 0 0 0 0 0 0 0 1
DEBUGFILE lnprot.deb
RUN 10
```

```
LINE 1 2 ERROR 1.0E-4
RUN 10
QUIT
```

Now we will put together the fragments of the four-node network given in Section A.2. An Example Network. Suppose we put the network description in a file "square.dat":

```
IMP 1 3 2
IMP 2 2 1
IMP 3 1 1
IMP 4 2 2
HOST 1/1
HOST 2/1
HOST 1/2
HOST 1/3
HOST 1/4
HOST 2/4
LINE 1 2
LINE 1 3
LINE 1 4
LINE 2 1
LINE 2 4
LINE 3 1
LINE 4 1
LINE 4 2
```

We can set up a simulation run to measure delays across this network as follows:

```
INIT 4 8
READ model.dat
MODEL IMP 0 LINEUDPERIOD 9999999 PACKET ON
MODEL HOST 0 PACKET ON
READ square.dat
FIXEDROUTING
ROUTE 1   0 2 3 4
ROUTE 2   1 0 1 4
ROUTE 3   1 1 0 1
ROUTE 4  1 2 1 0
OVERHEAD 128
PACKET 1 1
TRACEFILE delay.tra
UNISTART 100.0 1000 4
RUN 100
QUIT
```

We have switched off the line up/down protocol by specifying:

```
MODEL IMP 0 LINEUDPERIOD 9999999
```

and fixed routing has been chosen. The tracing chosen just traces node and network delays. If we want to investigate the operation of the SPF routing algorithm and the line up/down protocol, we can use the same network with slightly different parameters:

```
INIT 4 8
READ model.dat
MODEL IMP 0 TRACE ON DEBUG ON
READ square.dat
LINEUPDOWN 16
UPDATE 80
OVERHEAD 128
PACKET 0 1 0 0 0 1
TRACEFILE route.tra
DEBUG 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1
DEBUGFILE route.deb
START 1/2 1/3 10.0 1000 4
RUN 25
PRINTROUTING
START 1/2 1/1 20.0 1000 3
DUMP exampl.dmp
RUN 25
PRINTROUTING
LINE 1 2 ERROR 1.0E-4
DUMP exampl.dmp
RUN 25
```

```
PRINTROUTING
DUMP exampl.dmp
QUIT
```

Note that the error rate was increased on a line which was not carrying any traffic at all (at least not in that direction). but the line protocol will still bring the line down.

APPENDIX B. ANALYTIC MODEL USER MANUAL

This apppendix describes the procedures and commands that are necessary to run the multi-path routing analytic model (MRMAIN). The commands are grouped into sections based on their function and on the order in which the user will be most likely to encounter them when first becoming familiar with MRMAIN. Experienced users should consult the Alphabetical List of Commands (Section B.10) which gives brief descriptions of each command's function. It may also be useful to study the examples shown in Section B.9.

B.1 Operating System Considerations

As it is currently defined, MRMAIN runs on systems supporting Simula version 4A (Aug 1978) under the TOPS-20 operating system. To invoke MRMAIN, give the monitor command:

```
RUN MRMAIN
```

This will start the program running, and cause it to print a header message with the MRMAIN version number and general information about that version. When this has finished, the following prompt will appear:

```
Command:
```

At this point, you are ready to start entering commands to MRMAIN. These commands are described in the following sections.

For more detailed information about the peculiarities of MRMAIN's Simula language interface to the TOPS-20 operating system. refer to [8] and [14].

B.2 Input Data

Information about the traffic of the network to be modelled must be provided before MRMAIN can predict delays or optimize routing. The traffic is read from a file which contains one or more records in the following format:

```
<src> <dst> <rate> <length> <time> <prio>
```

The source and destination nodes of this particular flow are specified by <src> and <dst>. The rate packets are sent from <src> to <dst> is <rate> packets/second. Each packet in this flow is <length> bits long (including any protocol or other overhead). One exception: line protocol overhead is modelled separately for each line and is defined at the time each line is created. The <time> field is an integer value which can be used to simultaneously model network flows that occur at independent "times of day." If we have one flow with <time>=3 and a second flow with <time>=12, the two will not interfere with each other (we think of them as occurring at different times of the day...perhaps 3am and 12pm). The allowable range of <time> is 1:24 (for each hour of the day); if more internal table space is needed to model large networks, we can reduce the number of different times (of day) that can be modelled simultaneously.

The first time the traffic file is read, it is parsed and stored internally to MRMAIN. This permits faster operation of any command that causes the traffic file to be read. Note that the OPTIMIZE command causes the traffic file to be read at least three times per iteration; thus, parsed internal representation of the traffic file permits a significant saving in CPU time. If the traffic file is large. this feature can be disabled so that more room will remain for storing the node and line data structures in large networks.

B.3 Defining Topology, Traffic, and Routes

The topology and traffic of the network to be studied must be specified before modelling or optimization can take place. The topology is defined by entering commands to specify nodes and lines of the network. The traffic is defined by entering the name of a file that contains the formatted traffic rates. The commands that are used to define topology, traffic, and routes are: INIT, IMPS, IMPTIMES, LINE. SLINE, MROUTE, SROUTE, TRAFFICFILE. TRAFFICFACTOR, DISTRIBUTION, DROP, and EDIT. These are described in detail below.

INIT <# nodes>

Initialize the number of nodes to be modelled to <# nodes>. This is used to allocate space for the internal data structures of MRMAIN. If a previous INIT command has been given, all previously allocated nodes are garbage collected.

IMPS <#imps> <ln> [<task> <M2I> <I2M> <H2I> <I2H> <F2I> <I2F]

Define <#imps> different IMPs each having a maximum of <ln> lines connected to them. The maximum # of lines is needed so that the internal data structures of each IMP can be allocated efficiently. IMPs are numbered sequentially starting from 1 as they are defined by one or more IMPS commands. The IMP processing times are (optionally) specified by the parameters <M2I>..<F2I>. See the IMPTIMES command for a description of each of these parameters.

IMPTIMES <imp#> [<task> <M2I> <I2M> <H2I> <I2H> <F2I> <I2F]

Set the internal processing times for each part of IMP number <imp#> to the indicated values. If fewer than the 7 parameters are specified, the remainder are set to zero. If "#" is specified instead of an IMP number, the processing times are changed for all IMPs. Each parameter corresponds to the delay in a separate process of the IMP: <task>=task process, M2I=modem input. I2M=modem output. H2I=host input. I2H host output. F2I=fake host input (source), and I2F=fake host output (sink).

LINE <imp1> <imp2> [<cap>[<mult>[<framing>[<propDelay>]]]]

Define a full-duplex line connecting IMPs <imp1> and <imp2>. A number of parameters can optionally be specified. The line has a capacity of <cap> bits/second in each direction (default=50000). There are <mult> lines each of capacity <cap> in this line (default=1). Thus, the command "LINE 3 7 50000 2" creates a full duplex line between nodes 3 and 7 that has a total capacity of 100KB/sec (implemented as 2

50KB/sec lines in parallel). The number of bits of framing added on this line by the line protocol is <framing> (default=0). The propagation delay of this line is <propDelay> seconds (default=0). Related commands: SLINE.

SLINE <imp1> <imp2> [<cap>[<mult>[<framing>[<propDelay>]]]]

Define a half-duplex line from IMP <imp1> to IMP <imp2>. A number of parameters can optionally be specified. The line has a capacity of <cap> bits/second in each direction (default=50000). There are <mult> lines each of capacity <cap> in this line (default=1). The number of bits of framing added on this line by the line protocol is <framing> (default=0). The propagation delay of this line is <propDelay> seconds (default=0). Related commands: LINE.

MROUTE <src> <dst> (<via1> <fract1>) ... (<viaN> <fractN>)

Specify routing in multi-path format. The fraction <fract1> of the traffic from <src> to <dst> is sent via node <via1>. Similarly, fraction <fractN> of the traffic from <src> to <dst> is sent via node <viaN>. Nodes <via1>..<viaN> must all be neighbors of <src>. The sum of all the <fract1>..<fractN> values must be 1.0 for routing to be valid. The VERIFY command can be used to check this automatically. Fractions for neighbors which are not explicitly described by (<via> <fract>) pairs are set to zero. Related commands: SROUTE, VERIFY.

SROUTE <src> <via1> <via2> ... <viaN>

Specify routing for node <src> in single-path format. Traffic destined for node 1 is sent via node <via1>, while traffic destined for node N is sent via node <viaN>. A zero entry for any <via> parameter signifies that no value is to be specified at the current time. Typically, the entry for routing to itself at each node is set to zero. Each <via> node must be a neighbor of <src>. Related commands: MROUTE, VERIFY.

TRAFFICFILE <filename>

Traffic for the network to be modelled can be found in file <filename>. The format of the trafficfile is described in Section B.2.

TRAFFICFACTOR <real#>

Multiply the packet rates of all flows in the trafficfile by <real#> before they are used. This provides a convenient method of uniformly scaling traffic up or down without changing the trafficfile itself. The default trafficfactor is 1.0 (i.e., use the traffic exactly as it appears in the trafficfile).

DISTRIBUTION [DETERMINISTIC, EXPONENTIAL <parameter>]

Specify the type of packet length distribution to use for all flows. If DETERMINISTIC is selected, the packet lengths given in the traffic file for each flow are used. If EXPONENTIAL is selected, the length of each packet in each flow is selected from an exponential distribution with mean packet length of <parameter> bits. The values specified in the traffic file for packet lengths are ignored if EXPONENTIAL distribution is used. Default DISTRIBUTION = DETERMINISTIC.

DROP <node1> <node2>

Remove all lines connecting <node1> and <node2>. Related commands: LINE, SLINE.

EDIT <node> <z>

Edit the routing table of node <node> in routing table <z>. The value of <z> is one of "ZC," "ZS." or "ZA" corresponding to the "current," "shortest path," or "alternate" routing tables respectively. When this command is given, the prompt changes to "Edit: " and subcommands can be given to change and inspect the routing table. The subcommands are:

CH <dst> <via> <fract>

Change the fraction of traffic this node sends to node <dst> via node <via> from its current value to <fract>.

SHOW [<dst>]

Display the routing table entry for destination node <dst>. If <dst> is absent or "*", show routing table entries for all destinations.

DONE

Leave "Edit: " subcommand mode and return to normal command level. The VERIFY command can now be used to insure that the routing tables are (still) consistent.

B.4 Computing Flows and Delays

Once the network topology, traffic, and routing have been defined, it is possible to read the traffic file. distribute the flows through the links in the network, and compute parameters for the entire network. In particular, we can compute node and line loadings, average hopcount. average network delay and statistics for each source-destination pair; these are computed for each priority of traffic and for each time of day that is being modelled. The commands COMPUTE and MODEL are used to invoke these functions.

COMPUTE [<which flow>[<which routes>]]

Read the traffic in file TRAFFICFILE and distribute it throughout the flow tables <which flow> according to the routing in tables <which routes>. This also updates all the internal counters and partial statistics that are used in computing the network delay. Default values for <which flow> and <which routing> are the current routing ("ZC").

MODEL [<which flow>[<which routes>]]

Model the network performance by summing node and line delays over all lines in the network. Performance is computed using the flow in tables <which flow> and routing tables <which routes>. Default values for both of these are the current routing ("ZC"). The format of the output is:

```
time high(o) low(o) high(b) low(b) combined hopcount
 T     HO      LO     HB      LB       C        N
```

Where:

T = time of day

HO = delay for High priority packets if they were the Only ones in the network.

LO = delay for Low priority packets if they were the Only ones in the network.

HB = delay for High priority packets when the effect of Both high and low priority packets is taken into account.

LB = delay for Low priority packets when the effect of Both high and low priority packets is taken into account.

C = combined delay. Tnis is the weighted average of the HB and LB delays according to the total amount of high and low priority traffic.

N = network average hopcount; the average number of lines a packet must traverse before it reaches its destination.

Normally, the MODEL command is immediately preceded by a COMPUTE command to insure that the currently specified network variables are accurately represented in the internal data structures of the analytic model. Related commands: COMPUTE, PRINT PARAMETERS.

B.5 Reporting Results

A wide variety of information is available about all aspects of the network model. Node and line parameters can be printed in a variety of formats. Output suitable for use by the ARPANET simulator can be produced and stored in a file. Data can be printed on the user's terminal or redirected to a file for long-term storage. The commands in this section describe how to invoke the appropriate functions: PRINT, LIST, SIMULATION.

PRINT [<type> <which> <node range> <time range>]

Prints information about the current state of the model on the user's terminal. The data structures to use are specified by <which> to be one of current (ZC), shortest path (ZS), or alternate (ZA). The <node range> and <time range> parameters specify an optional range of nodes and times of day for which data is to be reported. Default values are "all nodes" and "all times of day". The <type> field specifies the information to be printed. Values of the <type> field and the information that each prints is shown below. If all the optional parameters are not relevant for all <type> options, they are ignored.

ALL

This is the equivalent of a series of different PRINT commands with the following options: PARAMETERS, TOPOLOGY, ROUTES, PEAK-RATE, PEAK-LOAD, PEAK-UTIL, PEAK-PROC, and ROUTESUMMARY.

PEAK

This is equivalent to a series of PRINT commands with the options PEAK-RATE and PEAK-UTIL.

PEAK-RATE, PEAK-LOAD, PEAK-UTIL, PEAK-PROC

Report information about the peak hour packet rate (pkt/sec), line load (bits/sec), line utilization, or processor statistics. Notation of the form "[2]" indicates that data is sent to node 2, while "@t" means that "t" is the peak time of day. A "[C]", "[S]", or "[A]" notation indicates the ZC, ZS or ZA data tables respectively. Thus, the line:

```
Node  1[C]: bits 2.250E+04[ 2]@12 2.250E+04[ 4]@12
```

means that for the ZC data structures, node 1 sends 22500 bits/sec to node 2 at time 12 and 22500 bits/sec to node 4 at time 12.

ALPHA

Report the value of the internal parameter ALPHA that is used when optimizing network performance. ALPHA is the fraction of the ZC and ZS routing that is used to produce a new and improved routing at each iteration (ZA). This command is useful only when running the OPTIMIZE command in manual (incremental) mode. Related commands: TRACE with the option for tracing ALPHA turned on.

ROUTES

Print the current state of the multi-path routing tables.

ROUTESUMMARY

Print a summary for each source-destination pair showing the average packet delay along each path from source to destination and the fraction of the flow that takes each path. Also reports statistics for the entire network. This is especially useful when compared to the routesummary output of the simulator.

PARAMETERS

Reports the current values of all parameters that may affect the operation of the model or the optimizer. This includes values such as node processing delays for each process, trace flags set, etc.

TOPOLOGY

Describes the topology of the network. This consists of a list of each node and its neighbors.

LIST <filename> <type> <which> <node range> <time range>

Identical to the PRINT command except that output is sent to file <filename> rather than to the user's terminal. All other fields are exactly the same as for the PRINT command. Stated in another way. the PRINT command is really just a

"LIST TTY: <other fields>" command.

SIMULATION [FILE <filename>, OUTPUT ROUTES]

Affects the way output is produced for input to the ARPANET Simulator. The "SIMULATION FILE <filename>" command specifies that output for the simulator is to be written to file <filename>. The "SIMULATION OUTPUT ROUTES" command causes the current routing tables to be written to a file specified by a previous "SIMULATION FILE <filename>" command. The file that is written is in the same format as that used as input for MRMAIN (MROUTE command); this format is also accepted by the simulator.

B.6 Performing Optimizations

MRMAIN can create optimal multi-path routing for a given topology and traffic. The Flow Deviation (FD) algorithm described by Kleinrock [4] and Gerla [5] is implemented. In order to find a feasible routing where none exists, or where the existing routing is not feasible for the current network, the initial feasible flow (IFF) algorithm described by Gerla [5] is also implemented. The commands in this section describe how IFF and FD can be used in MRMAIN to create optimal multi-path routings.

To perform routing optimization using MRMAIN, the user should first specify the topology and traffic of the network with commands described in Section B.3. In particular, the IMPS. LINEs and TRAFFICFILE of the network should be specified. If the routing is also known, it can be specifieJ at this time (using

MROUTE or SROUTE commands). The COMPUTE and MODEL commands can be used to get an initial estimate of the network performance. If the reported delay is INFINITE, the traffic is currently too large for some line in the network. The PRINT PEAK-UTIL command can be used to discover which lines are overloaded. If the reported delay is not INFINITE, then we can use the OPTIMIZE command without any arguments to improve the network performance.

If the initial routing is not known, or if the initial flow is not feasible. the OPTIMIZE IFF command should be used to force MRMAIN to compute an initial feasible flow (and routing). After it has created such a flow. it will proceed as if an OPTIMIZE command had been given. The commands relating to network optimization are described below. It will be helpful to study the examples shown in Section B.9 while referring to this section.

MRMAIN uses three completely separate sets of tables internally for storing the node and line data structures. These are used by the FD algorithm to produce improved flow and routing from the initial and shortest path flows. The tables are all accessible by the MRMAIN user; they can be referred to as "ZC" (the current routing tables and flow). "ZS" (the shortest path routing tables and flow). and "ZA" (the alternate routing tables and flow). In general. the user will only be interested in ZC unless the OPTIMIZE command is being run in incremental mode with the BREAK flags set (see Section B.8). ZC is the default for all commands that require one of the three sets of data structures to be specified.

A number of commands are also available to control the progress of the OPTIMIZE algorithm; these should be set before the OPTIMIZE command is given. In some cases, it also makes sense to change the values of the OPTIMIZE parameters _during_ a simulation run (e.g., we may want to reduce the tolerances of convergence as we get closer to the optimal solution). The default values of the OPTIMIZE parameters have been set so that the OPTIMIZE command works adequately over a wide range of network conditions. A detailed understanding of how the algorithms are implemented is needed in order to make intelligent changes to most parameters -- you have been warned!

MAXFLOW

Compute the maximum TRAFFICFACTOR that the network can support with the current topology and routing. This command is not particularly related to the other optimization commands except that it allows the user to scale up traffic automatically until the network is near saturation. The TRAFFICFACTOR is reset to the new value that is computed by the MAXFLOW command.

ITERATIONS [<opt> [<iff>]]

Set the maximum number of iterations of the FD algorithm to <opt> (Default = 15), and the maximum number of iterations of the IFF algorithm to <iff> (Default = 15). If the optimal routing or initial feasible flow is not reached after this many iterations, the algorithm is aborted. Note that OPTIMIZE will effectively run in incremental mode if ITERATIONS 1 1 is used.

K-POWER <k>

Set the value of "k" in the performance function to the (integer) value <k>. Default = 1.

OPTIMIZE [IFF]

Run the FD algorithm for the number of iterations specified by the ITERATIONS command. If IFF is specified, run the IFF algorithm first to create an initial feasible flow. To run the IFF algorithm only in incremental mode (stops after each iteration), use ITERATIONS 0 1; to run both IFF and OPT in incremental mode, use ITERATIONS 1 1. The TRACE command can be used to produce detailed information about the internal workings of the IFF and FD algorithms. Related commands: ITERATIONS, K-POWER, TOLERANCE, and TRACE.

B.7 General Function Commands

There are a number of commands that provide general functions not specifically related to analyzing or optimizing networks. The HELP command, for example, prints a summary of legal commands. The general function commands are: COMMENT, PUSH, TYPE, HELP, READ, WARNINGS, EXIT, POP, QUIT, and DUMP.

COMMENT <anything>

This command is printed, but otherwise ignored. It is useful for inserting comments and documentation into files that are to be read (using the READ command) by MRMAIN. Lines starting with ";" are also interpreted as comments for compatibility with other programs.

PUSH

Suspend the current operation of MRMAIN and enter an inferior fork running the standard operating system monitor. Any regular operating system command can be given; when done, type "POP" to return to MRMAIN at the point you left off. If you want to save the state of MRMAIN in a file without creating an inferior fork, use the DUMP command.

TYPE <filename>

Print the contents of the file <filename> on the user's terminal. This is the same as giving a "type" command to the operating system, except that it saves the trouble of giving the PUSH, TYPE <filename>, POP sequence of commands.

HELP

Prints a list of valid commands. Typing "?" will also cause the HELP command to be invoked.

READ <filename>

The contents of <filename> are read and interpreted as commands to MRMAIN. When end-of-file is reached, control returns to the user at the Command level. Nested READ commands (i.e., READ commands appearing in <filename>) are allowed.

WARNINGS [ON,OFF]

Causes warning messages to be turned ON or OFF. Warning messages indicate possible problems in algorithms or data structures (e.g., warnings are produced when the routing tables result in damped oscillations of packets, or when the optimizer finds that the function to be minimized is not concave).

[EXIT, POP, QUIT]

All these commands cause MRMAIN to exit without saving the current state of the model. To exit and save the current state, see the DUMP command. Related commands: ABORT, CONTINUE (for debugging).

DUMP <filename>

The current state of MRMAIN is saved in a file called <filename>. Now, if you EXIT, POP, or QUIT from MRMAIN, you can resume execution from the point of the DUMP command by giving the system-level command "RUN <filename>" (instead of the usual "RUN MRMAIN"). Note that the Simula debugger may not work properly when running a DUMPed file -- this is due to a bug in the Simula debugger!

B.8 Tracing and Internal Validity Checking

Features are provided for tracing the internal operation of the algorithms in MRMAIN. In addition, utility commands allow the user to manually verify the consistency of the internal data structures. This is particularly useful for checking data initially, or for rechecking data after it has been changed at some point during the operation of the program. The commands described are: VERIFY, ABORT, CONTINUE. TRACE. BREAK, and TEST.

VERIFY [ON, OFF, SINGLE. MULTI [ZC,ZS,ZA]]

Check to see that the routing tables are valid. VERIFY ON causes the routing to be checked automatically at each stage of the OPTIMIZE loop in which they are modified. VERIFY OFF switches off this checking and causes the algorithms to run much more quickly. Default is VERIFY ON. The VERIFY command can also be invoked manually to check the routing tables (e.g., after you have used the EDIT command). VERIFY MULTI is used to insure the consistency of the multi-path routing tables. The data structures to be checked can be specified explicitly; the default is VERIFY MULTI ZC. VERIFY SINGLE is used to check the single-path routing tables. The single-path routing tables are used for the shortest path computation of the FD algorithm. The user is normally concerned only with the ZC multi-path routing tables. Note that both the SROUTE command and the MROUTE command use the ZC multi-path routing tables!

ABORT

Leave the current (recursive) command level and return to the previous one. If you are at the basic command level ("Command:" prompt), this command has the same effect as QUIT, EXIT, or POP. If you are in a recursive command level such as "breakpoint" or "error," it causes you to immediately return to the basic "Command:" level, aborting any debug or breakpoint operation in progress. To continue with the current debugging, use the CONTINUE command.

CONTINUE

Leave the current (recursive) command level and return to the program at the point where it was interrupted. To exit from the tracing (recursive) command level, use the ABORT command.

TRACE <f1> <f2> <f3> <f4> <f5> <f6> <f7> <f8> <f9> <f10>

Turn one or more trace flags ON or OFF. To turn a trace flag ON, the value of the corresponding flag should be set to "1"; to turn it OFF, the value of the flag should be set to "0". Each of the flags <f#> controls a different value to be traced as described below.

<f1> IFF tracing. Prints progress of the IFF algorithm as it searches for an initial feasible flow. Default = 1.

<f2> OPTIMIZE tracing. Prints progress of the FD algorithm as it completes each iteration. Default = 1.

<f3> FLOW tracing. Print flow (bits/sec) in addition to routing information at each stage of the OPT or IFF iterations. Default = 1.

<f4> ADDFLOW tracing. Displays the lines each flow is sent over as the traffic file is read and packets are distributed throughout the network. Default = 0.

<f5> COST tracing. Print the cost information (from performance metric) that is used to construct the shortest path routing for each FD iteration. Default = 1.

<f6> SHORTEST PATH ROUTING tracing. Print information about the routine that produces the shortest path routing for each iteration of the FD algorithm. Default = 0.

<f7> LINEAR split tracing. Trace the function that calculates the linear combination of ZC and ZS routings (producing new routing in ZA) that is then passed to the minimizer function. Default = 0.

<f8> ALPHA tracing. Print the fraction that is finally returned after minimizing over the linear combination of ZC and ZS routings. The value returned is ALPHA: ALPHA*(ZC flow) + (1-ALPHA)*(ZS flow) = (ZA flow) which then becomes the new ZC flow for subsequent iterations of the FD algorithm. Default = 1.

<f9> SINGLE-PATH ROUTING tracing. Print the single-path routing that is created by each iteration of the FD algorithm. Default = 1.

<f10> MULTI-PATH ROUTING tracing. Print the multi-path routing that is created by each iteration of the FD algorithm. Default = 1.

BREAK [+,-] [IFF,OPT.COST.SHORT,FD]

Breakpoint tracing for the indicated option is turned ON (when "+" is specified) or OFF (when "-" is specified). If breakpoint tracing is turned on, the program will stop after the specified option has completed (e.g., stop after costs for each line in the network have been computed if the COST breakpoint has been set by a BREAK +COST command). At this point, a recursive command level is entered so that data structures can be inspected and other commands can be given. When ready to proceed, give the CONTINUE command; alternatively, type ABORT to exit from the breakpoint tracing. The possible stopping points are:

```
IFF   - initial feasible flow algorithm has completed
OPT   - optimize loop of FD has completed
COST  - line cost performance calculation is done
SHORT - shortest paths based on cost is complete
FD    - flow deviation iteration is complete
```

Default values for all breakpoints are OFF ("-"). When each breakpoint is encountered, the recursive command level will replace the normal "Command:" prompt with a prompt that indicates which breakpoint was encountered.

TEST <options>

Direct testing of individual subroutines from "Command:" level is possible using this command. This command is used only for initial development or debugging; its proper use requires a good understanding of the source code and hardcopy listing of the MRMAIN program and subroutines.

B.9 Examples of MRMAIN Use

In this section, we present two examples of how MRMAIN can be used to model network behavior and optimize performance. Both examples are based on the simple 4-node network shown below:

```
45KB/sec ==> 1---------2
             |         |
             |         |
             |         |
             4---------3  ==> 45KB/sec
```

We will send traffic from node 1 to node 3 at a rate of 45KB/sec (or more precisely, we will send 1000-bit packets at 45 pkt/sec). Initially, we will specify the routing to be via the path 1-2-3. The results shown are from an actual run of MRMAIN. Lines are numbered at the left so that they can be referred to conveniently in the text that follows each section of code.

EXAMPLE 1

In the first example we load the network, estimate its end-to-end delay. and print information about the nodes and lines. First, we must define the network topology and traffic.

```
 1  TOPS-20 Command processor 4(546)
 2  Friday. October 9, 1981 22:37:52
 3 @run n:mrmain
 4
 5 ** Multipath Routing Analytic Model...Version of 08-Oct-81 **
 6      (Priority queueing fixed; Non-bifurcated OPTIMIZE)
 7
 8 Command: read exampl.dat
 9 >INIT 4
10 >IMPS 4 3
11 >LINE 1 2 50E3
12 >LINE 2 3 50E3
13 >LINE 3 4 50E3
14 >LINE 1 4 50E3
15 >DISTRIBUTION DETERMINISTIC
16 >SROUTE 1 1 2 2 4
17 >SROUTE 2 1 2 3 3
19 >SROUTE 3 4 2 3 4
19 >SROUTE 4 1 1 3 4
20 >TRAFFICFILE EXAMPL.TRF
21 Command: type exampl.trf
22   1  3  45.0  1000  12  0
23 /*
```

Lines 1-7 show how we start MRMAIN running on the system. On line 8 we tell MRMAIN to read a file we have previously created that defines the network we would line to study. The lines in file exampl.dat are read and echoed on lines 9-20. Line 15 specifies that the distribution of all packet lengths will be deterministic with the values specified in the traffic file (this command is redundant since the default distribution is DETERMINISTIC). On line 21 we give the "type" command so that we can quickly look at the contents of the traffic file that is specified on line 20; the contents are displayed on line 22, while line 23 shows the end of the file mark.

```
24 Command: print parameters
25 ****************************************************************
26 *** Tolerance Settings ***
27   Stop IFF/OPT Rel    1.000E-02  Minimizer Accuracy 1.000E-03
28   Stop IFF/OPT Model 1.000E-07  Flow-Rte Mismatch  1.000E-03
29   IFF Epsilon Multi  5.000E-02  IFF Cost Infeas    1.000E-05
30   IFF Ratio Infeas   1.000E-03  Rounding Error     1.000E-06
31   MinBitRate         1.000E-03  % to stop flow     1.000E-02
32 *** Kleinrock Kth power model with K =    1
33 *** Distribution Type: DETERMINISTIC
34 *** Iteration maximums: OPT.IFF =    15   15 ***
35 *** Trace Flags ***
36 IFF,OPT.FLOW,ADDFLOW,COST,SHORT,LINEAR,ALPHA.PRTS,PRTM
37  1 1 0 1 0 0 1 1 1
38 *********************************************************
39 Command: compute
40 Command: model
41 time  high(o)   low(o)  high(b)   low(b) combined hopcount
42
43  12   0.22000  0.00000  0.22000  0.00000  0.22000    2.000
44 Command: print bit-rate
45 Node  1[C]: bits 4.500E+04[ 2]@12 0.000E+00[ 4]@12
46 Node  2[C]: bits 0.000E+00[ 1]@12 4.500E+04[ 3]@12
47 Node  3[C]: bits 0.000E+00[ 2]@12 0.000E+00[ 4]@12
48 Node  4[C]: bits 0.000E+00[ 3]@12 0.000E+00[ 1]@12
```

The PRINT PARAMETERS command on line 24 causes MRMAIN to print the value of all parameters that it uses in modelling the network and optimizing flows. These values are shown on lines 25-38. The compute command on line 39 causes the traffic file to be read and flows distributed throughout the network. On line 40 we ask MRMAIN to model the network performance; the results are reported on lines 41-43. Note that we have only high priority packets at "time of day" = 12. All other times of day have no traffic. On lines 44-48 we see that the traffic has indeed been distributed in the network. Line 45 reports the values for "Node 1[C]," where the "C" means the current or "ZC" routing tables (the default); "bits 4.500E+04[2]@12" indicates that there are 45000 bits/sec on the line going to node 2 at time 12. Time 12 is shown as the default because this is the time of peak traffic (or in this particular example. the only traffic).

EXAMPLE 2

In the second example we continue from where we left off in the first example. The object now is to optimize the routing in out sample 4-node network that is currently using a simple single-path routing along the path 1-2-3. We can do this quite simply (in spite of the fact that MRMAIN produces a lot of output with the default parameter settings).

```
1 Command: optimize
2 *************************************************************
```

```
*** Tolerance Settings ***
  Stop IFF/OPT Rel   1.000E-02  Minimizer Accuracy 1.000E-03
  Stop IFF/OPT Model 1.000E-07  Flow-Rte Mismatch  1.000E-03
  IFF Epsilon Multi  5.000E-02  IFF Cost Infeas    1.000E-05
  IFF Ratio Infeas   1.000E-03  Rounding Error     1.000E-06
  MinBitRate         1.000E-03  % to stop flow     1.000E-02
*** Kleinrock Kth power model with K =    1
*** Distribution Type: DETERMINISTIC
*** Iteration maximums: OPT.IFF =    15   15 ***
*** Trace Flags ***
IFF,OPT,FLOW,ADDFLOW,COST.SHORT.LINEAR,ALPHA.PRTS,PRTM
 1 1 1 0 1 0 0 1 1 1
************************************************************
Node  1[C]: bits 4.500E+04[ 2]@12 0.000E+00[ 4]@12
Node  2[C]: bits 0.000E+00[ 1]@12 4.500E+04[ 3]@12
Node  3[C]: bits 0.000E+00[ 2]@12 0.000E+00[ 4]@12
Node  4[C]: bits 0.000E+00[ 3]@12 0.000E+00[ 1]@12
*** OPT: Starting...
  1...******************************************************
*** Model(Current Flow) =  2.2000000E-01
*** Costs for t.prio = 12  0...Kleinrock model; k=  1
*** ComputeCost: Multiplier =         0.0222222223
Node  1: 2.2444E-05[ 2]..4.4444E-07[ 4]..
Node  2: 4.4444E-07[ 1]..2.2444E-05[ 3]..
Node  3: 4.4444E-07[ 2]..4.4444E-07[ 4]..
Node  4: 4.4444E-07[ 3]..4.4444E-07[ 1]..
*** Single Path Routing Matrix ***
Node  1:   1  4  4  4
Node  2:   1  2  1  1
Node  3:   2  2  3  4
Node  4:   1  3  3  4
Node  1[S]: bits 0.000E+00[ 2]@12 4.500E+04[ 4]@12
Node  2[S]: bits 0.000E+00[ 1]@12 0.000E+00[ 3]@12
Node  3[S]: bits 0.000E+00[ 2]@12 0.000E+00[ 4]@12
Node  4[S]: bits 4.500E+04[ 3]@12 0.000E+00[ 1]@12
*** Model(Shortest Path) =  2.2000000E-01
*** DONE OPTIMIZING: relative stoppingTolerance = 1.000E-02
*** Absolute stopping (theta) =  1.9800000E+00
*** Relative stopping (Gerla) =  8.9999591E+00
*** alpha is   5.0000000E-01
*** New Routing is:
*** Node  1[A] Routing Table:
 Dst node:      1      2      3      4
via node:   2  0.000  1.000  0.500  0.000
via node:   4  0.000  0.000  0.500  1.000
*** Node  2[A] Routing Table:
 Dst node:      1      2      3      4
```

```
via node:   1  1.000  0.000  0.000  0.000
via node:   3  0.000  0.000  1.000  1.000
*** Node  3[A] Routing Table:
 Dst node:      1      2      3      4
via node:   2  0.000  1.000  0.000  0.000
via node:   4  1.000  0.000  0.000  1.000
*** Node  4[A] Routing Table:
 Dst node:      1      2      3      4
via node:   3  0.000  0.000  1.000  0.000
via node:   1  1.000  1.000  0.000  0.000
Node  1[A]: bits 2.250E+04[ 2]@12 2.250E+04[ 4]@12
Node  2[A]: bits 0.000E+00[ 1]@12 2.250E+04[ 3]@12
Node  3[A]: bits 0.000E+00[ 2]@12 0.000E+00[ 4]@12
Node  4[A]: bits 2.250E+04[ 3]@12 0.000E+00[ 1]@12
Model based on flow    (in ZC) =   5.6363636E-02
Model based on routing (in ZA) =  -1.0000000E+00
*** Model improvement from last iteration =   1.2345679E+10
  2...**********************************************************
*** Model(Current Flow) =  5.6363636E-02
*** Costs for t,prio = 12  0...Kleinrock model; k=  1
*** ComputeCost: Multiplier =           0.0222222223
Node  1: 9.5684E-07[ 2]..9.5684E-07[ 4]..
Node  2: 4.4444E-07[ 1]..9.5684E-07[ 3]..
Node  3: 4.4444E-07[ 2]..4.4444E-07[ 4]..
Node  4: 9.5684E-07[ 3]..4.4444E-07[ 1]..
*** Single Path Routing Matrix ***
Node  1:    1  2  2  4
Node  2:    1  2  3  1
Node  3:    2  2  3  4
Node  4:    1  1  3  4
Node  1[S]: bits 4.500E+04[ 2]@12 0.000E+00[ 4]@12
Node  2[S]: bits 0.000E+00[ 1]@12 4.500E+04[ 3]@12
Node  3[S]: bits 0.000E+00[ 2]@12 0.000E+00[ 4]@12
Node  4[S]: bits 0.000E+00[ 3]@12 0.000E+00[ 1]@12
*** Model(Shortest Path) =  2.2000000E-01
*** DONE OPTIMIZING: relative stoppingTolerance = 1.000E-02
*** Absolute stopping (theta) =  0.0000000E+00
*** Relative stopping (Gerla) =  0.0000000E+00

***************************************************************
*** OPT Done: number of iterations =        1
*** Delay using optimal flow   =   5.6363636E-02
*** Delay using optimal routes =   5.6363636E-02
*** New Routing is:
*** Node  1[C] Routing Table:
 Dst node:      1      2      3      4
via node:   2  0.000  1.000  0.500  0.000
```

```
 97 via node:   4  0.000  0.000  0.500  1.000
 98 *** Node   2[C] Routing Table:
 99  Dst node:      1      2      3      4
100 via node:   1  1.000  0.000  0.000  0.000
101 via node:   3  0.000  0.000  1.000  1.000
102 *** Node   3[C] Routing Table:
103  Dst node:      1      2      3      4
104 via node:   2  0.000  1.000  0.000  0.000
105 via node:   4  1.000  0.000  0.000  1.000
106 *** Node   4[C] Routing Table:
107  Dst node:      1      2      3      4
108 via node:   3  0.000  0.000  1.000  0.000
109 via node:   1  1.000  1.000  0.000  0.000
110 Node  1[C]: bits 2.250E+04[ 2]@12 2.250E+04[ 4]@12
111 Node  2[C]: bits 0.000E+00[ 1]@12 2.250E+04[ 3]@12
112 Node  3[C]: bits 0.000E+00[ 2]@12 0.000E+00[ 4]@12
113 Node  4[C]: bits 2.250E+04[ 3]@12 0.000E+00[ 1]@12
114 ****************************************************************
115 Command: compute
116 Command: model
117 time  high(o)   low(o)  high(b)   low(b) combined hopcount
118
119  12   0.05636  0.00000  0.05636  0.00000  0.05636    2.000
120 Command:
```

The OPTIMIZE command on line 1 is used to cause the current routing and flow to be optimized using the FD algorithm. Lines 2-114 show the result of the FD iterations using the default TRACE flags. Although this output is rather long, it gives detailed information about the internal workings of the FD algorithm. The parameters used in the optimization are shown on lines 2-15. The initial bit rates are shown on lines 16-19. The first iteration of the FD algorithm starts on line 21 ("1..." indicates first iteration). The delay for the initial flow is reported on line 22 as 220ms. Lines 23-28 show the costs that are computed for each line using the performance metric. These values are used to construct the single-path routing tables that are shown on lines 29-33. The bit rates in the shortest-path flow are shown next on lines 34-37. Note that the shortest-path flow routes the 45KB/sec along the path 1-4-3. The delay of this shortest-path flow is also 220ms (see line 38). Lines 39-41 report tolerances which are used to determine whether the FD algorithm has come close enough to the optimal routing to stop. The splitting fraction between the current flow (ZC) and the shortest-path flow (ZS) is shown to be 0.5 (line 42). The new routing which has been computed from the ZC and ZS flows is stored in the alternate (ZA) set of tables (indicated by the "Node 1[A]" designation) and is displayed on lines 43-59. The bit rates for this new routing are shown on lines 60-63, and the network delay based on the new flow (which is now adopted as the ZC flow for the next iteration) is shown on line 64. Lines 65-66 are not meaningful in this context and should be ignored. The second iteration of FD starts on line 67. Note that the initial network delay is now 56.4ms. On lines 67-89 FD tries once more to improve network performance. This time the performance does not improve (because we are already at the optimal routing). FD signals that is done on line 90, and prints the final value of the network delay. routing tables, and bit rates. Note that the routing table for node 1 shows that .5 of the traffic destined to node 3 is sent via node 2, and .5 of the traffic is sent via node

4. Finally in lines 115-119, we repeat the COMPUTE and MODEL sequence we used earl'.er to confirm that the network delay has indeed improved (56.4ms now vs 220ms before optimization). At last, we are done! Note that 90% of the lines described above can be eliminated by turning off most of the trace flags (especially the flags that cause routing tables to be printed); give the command "TRACE 0 0 0 0 0 0 0 0 0" to turn off all tracing flags.

B.10 Alphabetical List of Commands

ABORT	- leave breakpoint or error tracing command level
BREAK	- set a breakpoint
COMMENT	- add comment line
COMPUTE	- read traffic and load network nodes and lines
CONTINUE	- continue debugging or tracing after breakpoint
DISTRIBUTION	- define packet length distribution
DROP	- remove lines between nodes
DUMP	- save the state of MRMAIN for later resumption
EDIT	- change a routing table
EXIT	- leave the MRMAIN program
HELP	- print list of available commands
IMPS	- define network nodes
IMPTIMES	- specify node processing times
INIT	- initialize number of nodes
ITERATIONS	- specify maximum number of IFF and OPT iterations
K-POWER	- define value for "k" in performance metric
LINE	- create a line between two nodes
LIST	- send information about the model to a file
MAXFLOW	- compute maximum network TRAFFICFACTOR
MODEL	- report average network delays and hopcount
MROUTE	- specify part of a multi-path routing table
OPTIMIZE	- optimize flow and routing of current network
POP	- leave the MRMAIN program
PRINT	- send information about the model to user's terminal
PUSH	- push to inferior exec running system monitor (EXEC)
QUIT	- leave the MRMAIN program
READ	- take input from a file
ROUTING	- specify routing to use (single-path or multi-path)
SIMULATION	- produce output for ARPANET simulator
SLINE	- define a single duplex line between two nodes
SROUTE	- define routing in single-path format
TEST	- MRMAIN interface for testing and debugging
TOLERANCE	- specify convergence tolerances for FD algorithm
TRACE	- trace internal operation of FD and IFF algorithms
TRAFFICFACTOR	- scale traffic up or down
TRAFFICFILE	- specify file where traffic information resides
TYPE	- print contents of a file on user's terminal
VERIFY	- check that routing tables are valid
WARNINGS	- enable or suppress warning messages

APPENDIX C. SAMPLE RUN OF SIMULATOR

This appendix shows a sample set of input and output for the simulator. The example we use is experiment 106. which was one of the experiments on a four-node ring described in Section 2. The input to the simulator consists of commands which describe the network topology. the protocols and the traffic to be simulated, and the commands which control simulator execution and output. This input is in the form of a single command file which contains references to other command files. To run the experiment. the following commands are given to the simulator:

```
echofile me:
read model0.dat
model imp 0 ( debug on
              lineUDperiod 99999999 )
read x30.top
read x106.off
seed 21784653525
spfrouting off
route 1 1 3 3 4
route 2 3 2 3 4
route 3 1 2 3 2
route 4 2 2 2 4
start 1/3 1/1 10.0 1000
start 1/4 1/1 10.0 1000
run 10 1
debug 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
debugfile x106.deb
spfrouting on
run 300 10
quit
```

The first command specifies that echoing of input should be directed to the terminal (or batch log file). The second command reads in a set of defaults for the IMP, host, and line from the file shown below. The next two lines turn on debugging output and switch off the line up/down protocol. X30.TOP contains the description of the network topology; X106.OFF contains commands for desynchronizing the IMP clocks. The SEED command sets the simulator random number seed. The SPFROUTING and ROUTE commands specify SPF routing, disable routing changes, and initialize each IMP's forwarding table. The START commands specify the traffic on the network. The RUN command directs the simulator to run for 10 seconds of simulated time. After 10 seconds, debugging flag 18 is switched on and debugging output is directed to the file X106.DEB. Routing changes are enabled, and the simulator is run for 300 seconds, then stops.

The file MODELO.DAT specifies the default values of parameters for the IMP, host, and line. Since this simulation is quite simple, these parameters are used unchanged for the 4 IMPs in the network. The commands in MODELO.DAT are:

```
model imp 0 (
   modemin 0.00052        modemout 0.00044
   hostin  0.00087        hostout 0.00053
   task     0.00170       fasttimeout 0.0032
   slowtimeout 0.0036     reroute 0.001
   source 0.001           sink 0.001
   dummy 0.007

   delayAvgPeriod 400     lineUDperiod 25
   updateAgingPeriod 12   timeout 0.0256
```

```
      retransmit 0.125      rate 1.0
      avgdelmax 1.6         threshold 0.064
      decay 0.0128          avgunits 0.0064
      delunits 0.0008       wordsize 16
      delayoffset 0.0       fastoffset 0.0

      packetlength 1008     numbuffers 35

      packet off            buffer off
      node off              debug off
      queue off
      )
   model line 0 (
      error 0.0             speed 50e3
      lag 1.0e-3            packet off
      debug off             framing 72
      numch 8               lud 3 20 60
      lineprotocol on
      )
   model host 0 (
      sink 0.0
      packet off            debug off
      length deterministic  rate negexponential
      )
   lineupdown 16
   update 80
   null 80
   overhead 128
```

The first set of parameters specifies the execution speed of each IMP process, the next set specifies the rate of various periodic events in the IMP, and the next set specifies parameters for the link and routing protocols. PACKETLENGTH gives the maximum length of a packet, and NUMBUFFERS gives the number of buffers in an IMP. The remainder of the IMP parameters turn off tracing and debugging output. The remainder of the file specifies default values for line parameters, default values for host parameters, and sizes for control packets and protocol framing overhead.

The file X30.TOP, shown below. describes the topology of the network. It is as follows:

```
INIT 4 8
IMP 1 2 1
IMP 2 2 1
IMP 3 2 1
IMP 4 2 1
HOST 1/1
HOST 1/2
HOST 1/3
HOST 1/4
LINE 1 3
LINE 1 4
LINE 2 3
LINE 2 4
LINE 3 1
LINE 3 2
LINE 4 1
LINE 4 2
```

The first command specifies a network with 4 IMPs and 8 (simplex) lines. The IMP commands create 4 IMPs, each with 2 lines and 1 local host. The HOST commands create those hosts. The LINE commands connect the IMPs with 4 bi-directional lines.

The file X106.OFF, shown below. gives random offsets to the clocks in each IMP, so they are effectively desynchronized.

```
imp   1 delayOffset   9.6512 fastOffset   0.0190
imp   2 delayOffset   1.2800 fastOffset   0.0188
imp   3 delayOffset   2.6624 fastOffset   0.0213
imp   4 delayOffset   9.1904 fastOffset   0.0110
```

For each IMP, the parameters DELAYOFFSET and FASTOFFSET are set to random values. These parameters set the relative time at which delay averaging takes place in each IMP.

The output of the simulation is directed to file X106.DEB, shown below:

```
Rout.Chg  4   1    11.52354  2  1
Rout.Chg  3   1    53.86344  1  2
Rout.Chg  2   1    53.86800  3  4
Rout.Chg  2   1    60.38890  4  3
Rout.Chg  3   1    60.39560  2  1
Rout.Chg  2   1   217.70844  3  4
Rout.Chg  2   1   268.90844  4  3
```

This is the entire output of the simulation run! Each line describes a routing change in a particular IMP. The first number is the IMP number, and the second is the destination to which this routing change refers. The third number gives the simulation time. The last two numbers give the IMP number of the neighbor for the old and new choice.

The initial routing is the worst-case routing. The output shows that the routing changes to the optimal routing at 11.5 seconds (i.e., 1.5 seconds after routing changes were enabled) and switches back to worst-case between 53.9 and 60.4 seconds. For the purposes of this experiment, the routing changes in IMP 2 are irrelevant. The routing is therefore optimal for all but 8 seconds of the 300 second run, or 97%.

References

[1] J.M. McQuillan. I. Richer. E.C. Rosen. D.P. Bertsekas. ARPANET Routing Algorithm Improvements. Second Semiannual Technical Report. BBN Report No. 3940, October 1978.

[2] E.C. Rosen. J.G. Herman. I. Richer, J.M. McQuillan. ARPANET Routing Algorithm Improvements. Third Semiannual Technical Report. BBN Report 4088, March 1979.

[3] E.C. Rosen. J. Mayersohn. P.J. Sevcik, G.J. Williams, R. Attar, ARPANET Routing Algorithm Improvements. Volume I. BBN Report No. 4473. August 1980.

[4] L. Kleinrock, Queueing Systems. Volume 2: Computer Applications. John Wiley & Sons. 1976.

[5] M. Gerla. The Design of Store-and-Forward (S/F) Networks for Computer Communications, Report No. UCLA-ENG-7319, School of Engineering and Applied Science. University of California. Los Angeles, January 1973.

[6] B. Meister, H.R. Mueller. and H.R. Rudin. Jr., "On the Optimization of Message-Switching Networks." IEEE Transactions on Communications. COM-20(1):8-14, February 1972.

[7] W. Chou and H. Frank, "Routing strategies for computer network design." presented at Symp. Computer-Communications Networks and Teletraffic, Polytechnic Inst. of Brooklyn, Brooklyn, NY, April 4-5. 1972.

[8] G. Birtwistle. L. Enderin. M. Ohlin. J. Palme. DECSYSTEM-10 SIMULA Language Handbook. Swedish National Defense Institute and the Norwegian Computing Center, NTIS # PB-243-064.

[9] M. Schwartz. Computer Communication Network Design and Analysis. Prentice-Hall. Inc. 1977.

[10] E.C. Rosen. Issues In Internetting Part 1: Modelling The Internet, Internet Experimental Note Number 184, May 1981.

[11] E.C. Rosen. Issues in Internetting Part 2: Accessing The Internet. Internet Experimental Note Number 187. June 1981.

[12] E.C. Rosen. Issues in Internetting Part 3: Addressing.

Internet Experimental Note Number 188, June. 1981.

[13] E.C. Rosen. Issues in Internetting Part 4: Routing. Internet Experimental Note Number 189, June 1981.

[14] "Simula and TOPS-20 Notes." On line documentation of bugs and features in file [BBNG]PS1:<BHITSON>SIMULA.NOTES. Available from the authors of this report.