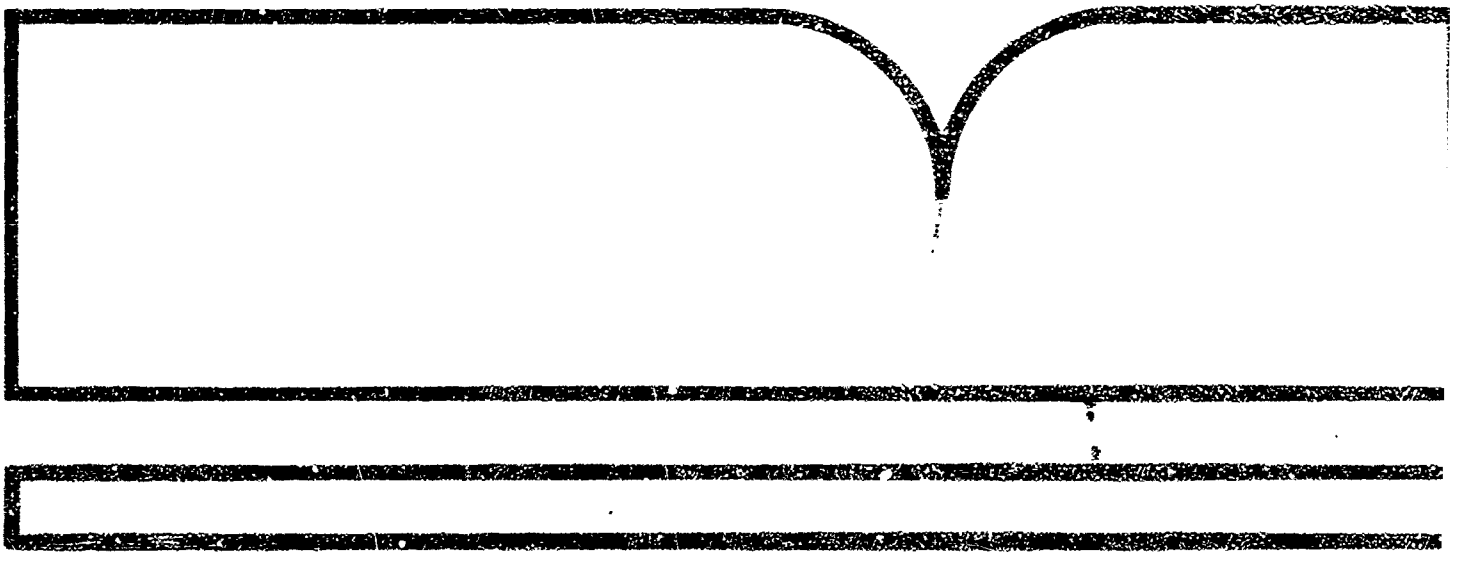
AD/A-092 065

ARPANET ROUTING ALGORITHM IMPROVEMENTS. VOLUME I

E. C. Rosen, et al

Bolt Beranek and Newman, Incorporated
Cambridge, Massachusetts

August 1980



BBN Report No. 4473

ARPANET Routing Algorithm Improvements

Volume I

August 1980

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: Dr. R.E. Lyons

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 4473 — 2. GOVT ACCESSION NO. AD-A092065	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ARPANET Routing Algorithm Improvements, Volume I	5. TYPE OF REPORT & PERIOD COVERED Annual Technical Report 8/1/79 - 9/1/80
	6. PERFORMING ORG. REPORT NUMBER 4473
7. AUTHOR(s) E.C. Rosen, P.J. Sevcik, R. Attar, J. Mayersohn, G.J. 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 August 1980
	13. NUMBER OF PAGES 498
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, buffer management, logical addressing, routing algorithms, AUTODIN II, multi-path routing, congestion control, simulation SIMULA, statistical analysis of simulation data, network simulation.

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

This report covers work performed during the first year of the extension of the ARPANET Routing Algorithm Improvements Contract. Network buffer management issues are discussed and a new buffer managemet scheme for the ARPANET is designed. Logical addressing is discussed, and a design is given for a logical addressing scheme suitable for ARPANET or DIN II. The applicability of ARPANET Routing to DIN II is evaluaced. The possibility of extending ARPANET's routing algorithm to provide multiple(cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (Continued)

routes between a given pair of nodes is discussed, and a preliminary design is proposed, which, however, still contains a number of unsolved problems. A set of metrics for evaluating congestion control algorithms is proposed, and AUTODIN II congestion control scheme is evaluated. A new congestion control scheme, suitable for networks containing ARPANET routing, is proposed. BBN's network simulator is described, and its command language is specified. Various simulation design decisions are discussed. The statistical properties of simulation data are discussed and various techniques for analyzing and interpreting simulation data are proposed.

TABLE OF CONTENTS

INTRODUCTION

This report covers work performed during the first year of the extension of the ARPANET Routing Algorithm Improvements contract. It describes technical work in progress. Each chapter is an independent treatment of a technical area of investigation.

In Chapter 1, we discuss the issues involved in designing a good buffer management scheme for a packet switching node. This is an area of nodal design which, although somewhat neglected, can have a significant impact on network performance, especially under heavy load. We begin with a discussion of the requirements a buffer management scheme must meet, both when there are few buffers and when there are many buffers. Some general principles of buffer management design are then developed. These principles are applied to the particular case of the ARPANET, and an improved buffer management scheme is presented. It is intended that these improvements be implemented during the second year of the contract, if memory constraints allow it.

Chapter 2 is concerned with logical addressing in computer networks. It begins with an explanation of the notion of logical addressing and a discussion of the functional requirements which a logical addressing scheme must meet. There is an extensive discussion of such issues as: how to translate

logical to physical addresses, where to do the translation, how to organize, initialize, maintain, and update the translation tables, and how to use logical addressing as a basis for multi-homing of hosts. A design of a logical addressing scheme is proposed, which is intended to be suitable either for the ARPANET or for AUTODIN II.

In Chapter 3, we discuss several issues concerning the applicability of the ARPANET's SPF routing algorithm to AUTODIN II. The bandwidth requirements of SPF routing in AUTODIN II are shown to be quite modest. We argue that the most effective implementation of SPF routing in AUTODIN II would treat each SCM (as opposed to each PSN) as a separate node. We discuss ways of treating the intra-PSN bus and the parallel trunking, and show that these features of AUTODIN II are easily handled by SPF routing.

In Chapter 4, we consider the possibility of extending and modifying the SPF routing algorithm so that it can maximize network throughput by providing multiple paths between each pair of nodes. Such an algorithm would have several components, including: (a) path-selection computation, (b) measurement of link and node utilization, (c) updating, (d) apportioning flows to paths, and (e) forwarding packets along a particular path. All these components are extensively discussed, and algorithms are proposed for each one.

In Chapter 5, a number of criteria, or metrics, for evaluating congestion control algorithms are discussed. These criteria are based upon qualitative practical network design considerations, rather than theoretical quantitative considerations. The AUTODIN II congestion control technique (RAFT) is evaluated against these criteria, and is found to be seriously deficient in several important respects. Based on this critique, various improvements are suggested.

In Chapter 6, we propose a new congestion control technique which is suitable for networks containing SPF routing. In this new technique, a measurement is performed on each link to determine whether that link is underloaded, maximally loaded, or overloaded. This information is fed back to source nodes via the SPF routing updates, where a slight modification of the SPF algorithm enables each node to determine whether its path to each other node is underloaded or overloaded. Based on this information, each source node can meter its traffic to avoid sending too much on paths which are overloaded.

One of our main tasks under this contract has been to develop a network simulator that can be used to test various routing and congestion control strategies. While development is still proceeding, Chapter 7 describes the current state of the simulator, specifying the way in which the simulator models a network node. Chapter 8 specifies the command language used to

run the simulator. Appendix 1 discusses some of the features of the SIMULA programming language, and explains why we thought it to be particularly appropriate for our simulator. A detailed implementation guide to the simulator is given in Appendix 2.

Chapter 9 discusses theoretical issues in the statistical analysis of simulation data. Two major problems are discussed. The first problem is the computation of confidence intervals for network performance parameters as estimated from the simulation output data. The second problem addressed is the bias which is present in the output data due to the initial conditions of the simulated system. We show that analysis of simulation data must be done quite carefully and with attention to the statistical niceties, if we are to avoid drawing false or ungeneralizable conclusions.

Chapter 10 discusses the specific algorithms which are implemented in our simulation analysis routines in order to estimate confidence intervals and eliminate the bias due to the initial conditions. A complete description of the simulation analysis software is provided. The routines are applied to simple simulations and very good results are obtained.

In chapter 11, we discuss the procedures we used to validate the random number generator which is used in our simulator. It is quite important to be sure of the properties of

the random number generator, since any problems with the generator will infect all the results obtained from the simulator. We conclude that the SIMULA random number generator is sufficiently random for our purposes. This chapter also provides an example of the techniques we will use to validate other parts of the simulator.

1. BUFFER MANAGEMENT IN THE IMP

1.1 Introduction

In Chapter 7 of BBN Report No. 4088 (ARPANET Routing Algorithm Improvements -- Third Seminannual Technical Report), we described the buffer management procedures currently in effect in the Honeywell 316/516 IMP. In a section on possible improvements, we pointed out several defects in the current scheme. Some of these defects may be thought of as "bugs," in the sense that they are due to details of the implementation which are in violation of the principles which underlie the design of the scheme. For example, under certain conditions, a buffer may be counted simultaneously as both a "reassembly" buffer and a "store-and-forward" buffer. Under other conditions, a single buffer may be counted as two store-and-forward buffers. Viewed conceptually, this is meaningless. It makes no sense to count a single buffer twice. As discussed in BBN Report No. 4088, the only effect of the "double counting" is to increase the holding time of buffers, thereby slowing the flow of data through the IMP. The source of these problems is that the TASK routine, when determining whether a particular buffer may be devoted to a "store-and-forward" function or a "reassembly" function, does not take into consideration the present state of the buffer; it simply assumes the buffer to be "uncounted" (this term is explained in BBN Report No. 4088). This defect arose because the

design principles of the buffer management scheme were unwisely sacrificed to simplify the implementation. The simplification is not great, however, and it would not be difficult to re-code the procedure to bring its implementation into agreement with the design principles.

However, we did not, in our previous report, make any attempt to make explicit the principles underlying the design of the ARPANET buffer management scheme, or to evaluate those principles critically. It would be unwise to fix implementation bugs in the current procedures without first considering whether there may also be design bugs in the procedures. That is the purpose of this chapter. We will begin by considering, in general, the function of a buffer management scheme in a packet-switching network. We will discuss the way in which such a procedure might be designed in an "ideal" network, where there is an ample supply of buffers. We will see that, no matter how many buffers there are, careful buffer management is essential to good performance. We will then discuss the way in which procedures designed for an ideal network need to be modified for a network (like the ARPANET and most other networks) in which buffer space is a scarce resource. Finally, we will compare the current ARPANET buffer management procedures to the procedures we develop, and will recommend changes to the former.

1.2 General Considerations of Buffer Management

A network node must execute many different functions for which it requires buffers. Among these functions are:

1) Transmitting packets on the various output devices (inter-node trunks or host access lines). Packets must be buffered while queuing for these devices, while in transmission on these devices, and (sometimes) while awaiting acknowledgment from the node or host on the other side of the device.

2) Receiving packets from the various input devices.

3) Reassembling messages so they can be transmitted to the destination host.

4) Processing packets. Packets must be buffered while the CPU is processing them, and they may have to occupy buffers while queuing for a busy processor.

5) Creating protocol or control messages. The IMPs often need to create control messages in order to run the many protocols necessary for proper network operation.

It should be clear that, no matter how many buffers exist in a node, a "laissez-faire" approach to buffer management cannot possibly succeed. In a laissez-faire approach, buffers are

allocated to the various processes that need them on a first-come, first-serve basis. Any process, at any time, can obtain any number of buffers that are available at that time. No import is given to considerations of fairness or of overall network performance. Therefore, a laissez-faire scheme will be prone to lock-up. Suppose, for example, that the output processes in some node have taken all the buffers. Then no input can be done. If, as is often the case, the output processes cannot free their buffers until an acknowledgment is received from some other node, and if acknowledgments cannot be received because no buffers are available for input, then there is a deadlock, and the buffers will never be freed. It is important to understand that this sort of deadlock is not caused by a shortage of buffer space. No matter how much buffer space is available, it is always possible, for example, that the network will try to utilize some output device at a higher capacity than it is capable of handling. With a laissez-faire approach to buffer management, there is no bound on the number of buffers which may end up holding packets for the overloaded device. The possibility of deadlock cannot be eliminated by adding more buffers.

This particular sort of deadlock is just one example of a more general situation. For the network to perform well, all the processes in the nodes must be able to run at an adequate rate.

This cannot be guaranteed unless each process is guaranteed the resources that it needs. Unless each process is explicitly prevented from "hogging" resources, other processes may be unable to run, and the network will not, in general, be able to give adequate performance. It must be understood, of course, that the buffer supply is not the only resource which must be managed in order to prevent hogging. Similar sorts of deadlocks can occur if some processes are allowed unrestricted access to CPU cycles, thereby preventing others from ever running at all. Although this chapter is primarily concerned only with management of the buffer space resource, management of the CPU resource is equally important. Furthermore, it must not be imagined that deadlocks are the only sort of performance degradation against which a buffer management scheme must protect. Freedom from deadlocks is only a necessary, not a sufficient, condition of adequate network performance. A scheme which dedicates some small number of buffers to each process, while taking a laissez-faire approach to the large majority of the buffers, may prevent deadlocks, since it will permit each process to run at some slow but non-zero rate. However, such an approach may not allow all the processes to run at "adequate" speeds; if some processes are running "too slowly," then ordinary users of the network may not be able to distinguish that situation from the situation where there is a deadlock. The problem is the general one of "fairness." The purpose of a buffer management scheme is to ensure that no

process gets either more or less than its fair share of the buffer resource. (It is worth noting that simply specifying a protocol in some formal language, i.e., in a way which is not implementation-specific, and proving it to be deadlock-free, does not guarantee that the protocol will perform fairly. Such formal specifications almost never address such important issues as buffer management or fairness. In fact, by abstracting the protocol specification from implementation considerations, such issues are only obscured and made easier to overlook.) Of course, such notions as "adequate performance," "too slow," and "fair share" are hopelessly qualitative. Implementing a buffer management scheme in an actual network would require giving some quantitative interpretation to these notions. The precise way in which these notions are quantified would depend on the design objectives of the particular network, as well as its performance characteristics, and it would probably require a large degree of arbitrariness. This does not mean, though, that the qualitative considerations cannot guide the development of a buffer management procedure, but only that any such procedure should be sufficiently parameterized so that it can be tuned to meet the particular requirements of a particular network.

The considerations raised above do not mean that there should be no sharing of buffers among processes, but only that the sharing should be controlled so that considerations of

fairness and overall network performance can play a role. There is, of course, a disadvantage to restricting the amount of sharing of buffers among processes. If a buffer is available for process A, but not for process B, then there will be situations in which a buffer must lie idle, because process A does not need it, even though process B really has a use for it. In these particular situations, the performance of process B (and possibly of the whole node) may be degraded. The justification for keeping the buffer idle though is that it is possible that process A will have a need for the buffer before process B would finish with it, and that if such a situation were to arise, overall performance would be improved by keeping the buffer idle until needed by process A. The validity of the justification depends on the probability that process A really will need the buffer before process B would finish with it. This sort of probability is very difficult to evaluate a priori. Furthermore, the probability may change as network conditions change. This suggests that we might want to vary the number of buffers reserved for particular processes as a function of the utilization of resources by the various processes. That is, the buffer management scheme may need feedback from a more general congestion control scheme which can measure the pattern of resource utilization and determine whether it is satisfactory. This is only natural. The purpose of a congestion control scheme is to ensure that the demands placed on resources in the network

do not exceed the capacity of the resources, and that the resources are allocated to the demands in the way that yields best overall network service. In order to achieve these goals, the algorithm (or at least the parameters of the algorithm) used to assign resources to demands may need to change as the pattern of demands changes. A buffer management scheme is an algorithm for assigning one particular kind of resource (buffers) to the demands made on that resource. Hence it is just a part of a congestion control scheme, and may need to interact with the other parts of the scheme for best overall performance.

1.3 Buffer Management with an Ample Supply of Buffers

If we were designing a new network, with an ample amount of buffer space, one of the important desiderata of the buffer management scheme would be to enable all output devices (i.e., hosts and inter-node trunks) to run at their rated capacity. Transmission of packets over an output device is usually controlled by means of a protocol which requires the packet to remain buffered until a positive acknowledgment is received. The number of buffers needed to run such a device at full capacity is a function both of the transmission speed of the device and of the time it takes (on the average) for acknowledgments to return, which itself is a function of the physical length of the transmission line (speed-of-light propagation delay) and the processing latencies of the device which is receiving the output.

For each output device it is relatively straightforward to compute this number of buffers, at least approximately. To ensure that each output device can always run at its rated capacity, the buffer management scheme must "dedicate" that number of buffers to the particular output device in question.

It is important to understand just what it means to "dedicate n buffers" to a particular device or process. It does not means that certain physical buffers (i.e., physical areas of memory) are set aside for use only by that process. It means only that the process should always be able to obtain n buffers whenever it has a need for n buffers. There is no reason at all why the same n physical buffers should be used each time. To see exactly what this means in practice, we must consider the mechanism whereby a buffer is (logically) moved from a source process to a destination process. At any given time, a buffer which is not free is considered to be under the control of some process. When that process has completed its processing of the buffer, it must somehow release control of it. In some cases (e.g., a packet has been transmitted on an inter-node trunk and an acknowledgment for it received) the packet which is in the buffer is no longer needed at that node, and the buffer can be freed. In other cases, however, control of the buffer must be turned over to some other process. An example is a packet which is under control of the forwarding process of the routing

algorithm. Once the routing algorithm decides where to forward the packet, the buffer in which it resides must be turned over to some output process which will ensure its transmission over the appropriate output device. Before turning the buffer over to the next process, it must be determined whether doing so would prevent any other process from obtaining the number of buffers that have been "dedicated" to it. If so, the buffer cannot be turned over to that destination process. If the packet residing in the buffer is under control of some sort of reliable transmission procedure (e.g., the ARPANET's IMP-IMP protocol), the buffer can simply be freed. This will not result in loss of the packet, since the reliable transmission procedure will ensure that the packet is seen again, and again, until it is finally accepted. This is usually the case in the ARPANET with a packet that has been received from a neighboring node. If the receiving node discards the packet without sending an acknowledgment to the transmitting node, the latter node can usually be relied upon to send the packet again. (Note that this implies that, in the ARPANET, the receiving node cannot send an inter-node acknowledgment for a packet until that packet has been turned over to its final output process.) On the other hand, some packets may not be under the control of a reliable transmission procedure. This may be the case with control packets that are created in the node itself and which must be transmitted to some other node for reasons determined by some end-end protocol.

Freeing the buffer occupied by such a packet may result in loss of the packet. Since this is undesirable, if the buffer cannot be given to its destination process, it must be returned to the source process, where it must sit on some queue until some future time when it can be accepted by the destination process.

In general, when making the determination as to whether a buffer can be turned over to a particular process, it is not sufficient merely to consider the number of buffers already in control of the destination process. One must also take into consideration the source process of the buffer. After all, there may be cases in which the source process and the destination process share a common pool of buffers. In such cases, buffer management considerations can never cause the destination process to refuse the buffer, no matter how many buffers are already under its control. It follows that the correct decision as to whether a buffer ought to be refused cannot be made without knowledge of its source process. Also, only by considering the buffer's source process can it be determined whether or not the buffer, if refused, will be freed. This is important to know, since once it has been decided that a particular packet cannot be discarded at will, no process should ever reject the packet as a result of buffer management considerations. Any process that will not be able to obtain an adequate number of buffers if the packet is accepted will also be unable to obtain an adequate

number of buffers if the packet is rejected. After all, rejection of the packet will merely cause its buffer to be held in a queue somewhere else in the node until it can be accepted. Since the buffer cannot be freed, it will not become available for use by any other process, so there is no point in refusing it. Rejecting the packet will serve only to increase its delay, without any countervailing advantage. This may mean that the number of buffers under the control of a given process exceeds the nominal maximum which we have decided to allow to that process. The point of the buffer management scheme, however, is not so much to prevent a process from obtaining more than some maximum number of buffers as to ensure that a process can always obtain some minimum number of buffers. In the situation just described, holding one process to a certain maximum number of buffers does not help any other process to obtain its minimum. And while moving the buffer from the source process to the destination process in this situation may cause the source process to have less than its minimum number of buffers, it cannot hurt the performance of the source process, which, after all, has already finished with its use of the buffer. There is certainly no point in forcing a process to keep control of a buffer with which it is finished; that could serve only to degrade overall performance.

To put the point another way, once the node has committed itself not to discard the packet, all buffer management considerations are otiose. Of course, this is not to say that a packet to which the node is committed ought never to be refused by any process in the node, but only that considerations of buffer management can play no role in the refusal. There are many resources other than buffer space which may be in short supply; management of these resources may well dictate the rejection of a packet to which the node is committed. However, the same considerations apply. A packet should never be rejected due to resource management considerations unless rejecting it will free resources which would not be free were the packet accepted.

Of course, this principle may have unfortunate side-effects that must be controlled. If two packets are competing for buffer space, and one of the packets is discardable while the other is not, the non-discardable packet has an advantage, since it cannot be refused. For example, in the ARPANET, packets which an IMP receives from a neighboring IMP are discardable, since they are controlled by a reliable transmission procedure (the IMP-IMP protocol) and will be retransmitted if dropped. Packets received from a host, however, are controlled by the 1822 protocol, which does not provide for retransmissions, and which in fact assumes that the IMP will not drop a packet once it has fully received

it. This fact gives packets received from hosts an unfair advantage over packets received from neighboring IMPs in the competition for buffer space. This is a particularly unhappy situation, since it can lead to the violation of one of the basic principles of congestion control, namely that packets already in the network should be favored over packets just entering the network. The correct solution to this problem, of course, is to refrain from using protocols which force a node to treat a packet as non-discardable before all the resources needed to process that packet have been obtained. We will return to this issue when we discuss the particular case of buffer management in the ARPANET.

It should also be noted that moving a buffer from a source process to a destination process typically requires the mediation of a third process which serves as the Dispatcher. In the ARPANET, this is the function of the TASK process. While a buffer is queued for or being processed by the Dispatcher, it is still considered to be under the control of the source process, for purposes of buffer management. The reason, of course, is that the decision as to whether a particular destination process must refuse the buffer is independent of whether the buffer is being passed to it directly by the source process, or whether it is being passed to it by the Dispatcher. Therefore, it makes no sense to treat the Dispatcher itself as a source process.

Similarly, since the Dispatcher itself can never refuse a buffer, it makes no sense to treat it as a destination process either. The use of a dispatching process should be transparent to the buffer management scheme.

Sometimes a buffer may need to be under the simultaneous control of two distinct processes in order for its packet to be processed. If this is ever the case, the buffer management scheme must ensure that whenever the buffer can be assigned to one process, it can also be assigned to the other. If the buffer cannot be processed unless controlled by both processes, then a situation where it can be controlled by one process but not the other makes no sense at all. Such a situation would simply result in a waste of space, by allowing a buffer to be occupied by a packet which cannot be processed. This illustrates a most important point in the design of a buffer management scheme. The purpose of buffer management is to ensure good overall network performance. Therefore, one cannot determine how many buffers need to be dedicated to a process by considering that process in isolation. Rather, one must consider the role that that particular process plays in determining overall network performance.

1.3.1 Buffering for Output

We now consider, in general, which sorts of processes in the network nodes need to have buffers dedicated to them. Whenever a particular device is running at close to its maximum capacity and the demands on the device vary stochastically, the device will sometimes be overloaded. That is, although the device is fully utilized during some interval by the presence of n packets, a larger number of packets destined for that device will arrive during that interval. If the device is overloaded in the steady state, then some sort of congestion control procedure must be brought into effect to reduce the demand for that particular device. We are presently assuming, though, that the device is not overloaded in the steady state, and that any intervals of overload are caused by the variance in the demand. In such a situation, it is desirable to smooth the effects of the temporary overload by buffering the excess packets. So the buffer management system should allow more buffers to be assigned to an output device at a given time than are strictly needed to run that device at full capacity. The question is whether a certain number of excess buffers should be "dedicated" to each device (in the sense described above), or whether the excess buffers should be in a common pool, sharable among all the output devices on a first-come, first-served basis. In this case, it seems that the buffers ought to be sharable. If all these buffers end up queued

to a single output device, no other device is thereby prevented from running at full speed, since each device still has its own supply of dedicated buffers. Therefore there is no reason to strictly partition this additional buffer space.

One might argue that the number of buffers dedicated to a particular device should only be enough to run the device at its average rate, not at its maximum or peak rate. After all, the purpose of having a sharable pool of excess buffers is to smooth the effects of stochastic peaks. But stochastic peaks occur whenever the average utilization of a device is exceeded, not necessarily when its maximum utilization is exceeded. This argument, however, ignores the fact that several devices may exceed their average utilization at the same time. If this happens, and if there are not enough buffers dedicated to each device to run it at full speed, then some devices may be under-utilized while others will be over-utilized, which is what the buffer management scheme ought to try avoid as far as possible (at least, if the supply of buffers is ample).

1.3.2 Buffering for Input

We have yet to discuss the issue of whether it is necessary to dedicate buffers to the input devices, as well as to the output devices. Packets may arrive at a node either from a neighboring node, or from a locally-attached host. Receiving and

processing a packet requires a buffer. Even if all output devices are running at full speed and have their full complement of buffers, it is still necessary to dedicate a certain number of additional buffers to the input devices. Failure to do so can result in the stopping of all input whenever all the output devices are fully utilized. At first glance, this might seem like a desirable effect. After all, there is no point in accepting input when the output devices are already overloaded; to do so only leads to congestion. However, there are two problems with this argument:

1) Not all packets which arrive at a node as input will necessarily leave the node as output. Some packets are control packets which may cause the processor to take some action other than simply forwarding the packet somewhere else. The node should always be able to process these packets, no matter what the utilization of its output devices.

2) Packets cannot be processed instantaneously; there is always some latency. It may be the case that although no output buffers are available at the time a packet arrives, there will be buffers available by the time the packet is processed (e.g., by the time the processor determines which output device to route the packet to). If no buffers are available at the time the packet is

received, it has to be discarded and re-transmitted, thus introducing a potentially large amount of additional delay. This additional delay can be eliminated by having a supply of buffers for input.

These arguments show that there should be some buffers available for input over and above those which can be used for output. We have not yet dealt with the issues of how many buffers there should be, and whether they should be sharable among all the input devices. It is sometimes suggested that there should be two buffers dedicated to each input device, to allow "double buffering." However, this is something of a confusion. The point of double buffering is to allow an input to be received while the previous input is being processed. This makes sense if the time it takes to process the previous input is less than the time it takes to receive the current input. Then by the time the input is received, processing of the previous one has been completed, and the buffer which held the previous input can be re-used to receive the next input, while the current input is being processed. The purpose of such a scheme is to ensure that reception of an input is not delayed by the time it takes to process the previous input. It is easy to see though that this scheme is not directly applicable to a packet-switching node. There is no way to guarantee that the time needed to process one packet is less than the time needed to receive the next packet.

If the processor is busy, so that many packets are queued for it, and the inter-node trunks run at a high speed, so that packets are received very rapidly, merely dedicating two buffers to an input device will not ensure that a buffer is always available to receive the next packet.

One might think that this means that a larger number of buffers must be dedicated to each input line. By making the number large enough, we can make the probability of missing an input due to lack of buffers as small as we like. But it would be a mistake to do so. In general (though not invariably), after a packet is input and processed, it will be routed to some output device. There cannot be a shortage of buffers for input unless either all the output devices are heavily loaded (i.e., all the output-dedicated buffers are in use), or the processor itself is overloaded (so that many buffers are queued for the processor). A certain number of input-dedicated buffers are needed to permit input to flow smoothly under such situations, as well as to ensure that control packets can be processed. However, if the node is really congested (i.e., either the output devices or the CPU are overutilized in the steady state), having a large number of input buffers will not smooth the flow; it will result only in larger queues. The number of input-dedicated buffers need only be large enough to enable the processor to run at its full capacity while the output devices are also running at full

capacity. In order for an output device to run at full capacity, it should always be able to get enough buffers so that it can buffer all in-flight packets for the required period of time while still having a small queue of packets waiting to be sent. Running the processor at full speed requires only enough buffers so that a small number of packets can always be on the queue for the processor. This does not require a large number of buffers to be dedicated to input; even less does it require a large number of buffers to be dedicated to a particular input device. However, as we have pointed out, it does require some dedicated buffers.

We have now determined that there need not be a very large number of buffers dedicated to input. We have not yet resolved the question of whether these buffers should be sharable among all the input devices, or whether a certain number of buffers should be dedicated to each input device. To answer this question we must determine whether, if the buffers are sharable, some one input device can monopolize the buffer pool, preventing input from any of the other devices. This might well be the case, for three reasons. First, one input device might run at a higher speed than the others. Second, one input device might be more heavily utilized than the others, or might receive shorter packets than the others. Third, some artifact of the interrupt structure of the node might tend to favor certain devices over

others. (Thus in the ARPANET, each inter-IMP trunk is serviced at a different priority level; naturally, the one that is serviced with the highest priority is favored. This is due to the interrupt structure of the 316, rather than the software.) If any of these conditions hold, some input devices may be able to utilize so many buffers that the others are slowed down. Therefore a small number of buffers should be dedicated to each input device.

Another reason for dedicating a few buffers to each input device is the following. Certain inputs are processed at a very high priority level, without any queuing for the processor. These inputs are always control packets, which are not going to be routed to any output device. Furthermore, they are only those few types of control packets which must be processed very quickly. An example is the line up/down protocol packet of the ARPANET. When one IMP sends one of these packets to another, it expects a reply back within a few hundred milliseconds, no matter how congested the processor of the receiving IMP is. The receiving IMP must always be able to receive such packets and to process them immediately, without having to queue them. If this is not done, the line may be brought down spuriously, resulting in a significant and needless degradation of network service. In order to ensure rapid processing, at least one buffer must be dedicated to each input device from which control packets of this

sort may be received. Furthermore, the use of these buffers is even more restricted than that of other buffers which are input-dedicated. Ordinarily, to say that n buffers are dedicated to input is to say that there must always be n buffers which cannot be given to any process which is not input related. These buffers can, however, be queued to the processor (i.e., to the Dispatcher) after being filled with an input. After all, the main point of having input-dedicated buffers is to enable the processor to continue to look at inputs even if all output devices are running at full capacity. This goal cannot be achieved unless the input buffers can be queued for the processor. The point of this paragraph, on the other hand, is that there be certain sorts of control packets which require immediate processing. In order to ensure that a buffer is always available to each input device to process such packets, each input device should have one buffer dedicated to it which is not queueable to any other process, including the Dispatcher. Is a single such buffer enough? The feasibility of having protocols which require immediate processing of control packets is clearly dependent on the constraint that such packets be few and far-between. Otherwise, there may just be too many of them to process them all "immediately," and the protocol will not work. As long as this constraint is met, a single buffer should be enough.

It must be pointed out that the proper use of the non-queueable buffer is often a matter of some subtlety. Suppose a packet is received from some inter-node trunk, and that packet contains node-node acknowledgments (possibly piggybacked on an ordinary data packet) for packets that were transmitted (in the opposite direction) over the same trunk. Suppose further that after the packet is received, there are no more free buffers in the nodes. Clearly, any data in the packet cannot be processed; doing so would require queuing the packet for the processor, thereby violating the rule that each input device have a non-queueable buffer dedicated to it. But what of the acknowledgments -- should they be processed? In the ARPANET, received node-node acknowledgments are processed at the highest priority level, with no queuing. So they can be processed without violating the buffer management law that we have advanced. Furthermore, one might argue that it is really important to process the acknowledgments as soon as possible. After all, processing received acknowledgments can result in freeing buffers. Since, ex hypothesi, there are very few free buffers in the machine, processing the acknowledgments is of great importance, and should be done immediately. This argument, however, does not hold under all conditions. When there are very few free buffers in the node, it may be that a large number of buffers are holding packets which have already been transmitted on inter-node trunks, and which are awaiting acknowledgment. In

this case, processing the acknowledgments as quickly as possible has a salutary effect on the node's performance. However, there are other conditions which may result in a short supply of free buffers. Suppose, for example, that the node is CPU-bound, i.e., that the processor is overloaded. Then one would expect to find the majority of buffers queued for the processor. (This situation is very common in certain of the more heavily loaded ARPANET nodes.) Since these buffers contain packets which have not yet been transmitted out any inter-node trunk, the buffers cannot possibly be freed as a result of processing acknowledgments. The only way to expedite the freeing of these buffers is to reduce the demand on the processor, especially the demand at the higher priority levels. Thus the best strategy here may be to not process the acknowledgments, thereby reducing the processing load. Deciding whether a certain packet should be processed immediately may depend not only on the function of the packet, but on the conditions in the node at that time. This shows again that a buffer management scheme is only part of a more general congestion control strategy, and cannot be expected to do the whole job by itself.

It must be understood, of course, that although the number of buffers dedicated to input may be small, the number of buffers controlled by the input processes (i.e. the number of buffers containing input packets which have not yet been dispatched) may

be much larger. In fact, all the buffers that are dedicated to output processes may be under the control of input processes at some time. This may seem paradoxical, but it is easy to see why it is the case. In general, a packet cannot be output unless it has first been input. It makes no sense to refuse to use a buffer for input because one wants to save it for output -- it will never be used for output unless it is used for input first. Therefore, all buffers must be available for input, regardless of the number which are "dedicated" to other processes. (There is one exception to this rule. It may be desirable to save a few buffers for creating control messages, which, being created in the node, are never actually input. These buffers would then be unavailable for input. This is discussed below in greater detail.) To restate the point -- while only a small number of buffers need to be dedicated to input, a large number of buffers need to be available to input.

1.3.3 Buffering for Generating Control Messages

There are other functions besides input and output for which buffers are required. One such function is the creation of the control messages needed to run the various protocols used by the node. Every so often, the node will have to respond to a certain event by creating a control packet and transmitting it to some destination. Often one node will contain buffers which cannot be freed until a control packet from some other node is received.

If a node cannot create the necessary control packets because it cannot get buffers for them, then deadlocks are possible. Even if deadlocks are avoided, good network performance can depend on the timely creation and transmission of control packets. Nodes which have high buffer utilization because they are handling many data packets ought not to be at a disadvantage when it comes to obtaining buffers in which to create control packets. Indeed, it is just such nodes which are most likely to have the largest number of protocol-imposed responsibilities, and hence to have the greatest need for buffers in which to create control messages. In order to ensure that the flow of control messages is not slowed by the flow of data packets, each node should have a supply of buffers dedicated solely to the function of creating control messages.

1.3.4 Buffering Data at the Source Node

In many packet-switching networks, packets received from a host are buffered at the source node until an end-end acknowledgment is received. (This is true of single-packet messages in the ARPANET.) An insufficient supply of buffers for this purpose will hold the throughput of the locally attached hosts to an artificially low level. Furthermore, the holding time of a buffer which must await an end-end acknowledgment is very long, relative to the holding time of other buffers. This implies that the number of buffers needed to serve the function

might be quite large, if an adequate level of throughput is to be maintained. A basic principle of congestion control in packet switching networks is that packets which are already in the network should not be unduly interfered with by packets which are entering the network. The buffer management scheme we have been outlining applies this principle by dedicating pools of buffers to each output device and to the various protocol functions. That is, the scheme ensures that local inputs cannot hog the buffer space at a node, which would result in degrading the flow of traffic through the node. There is a question, however, as to whether there should be a pool of buffers dedicated to buffering input packets at the source node, or whether this function should compete with other functions for a sharable buffer pool. Since we have already assigned dedicated buffer pools to those other functions that need them, the only possible bad result of not dedicating a pool of buffers for source buffering of local inputs would be that these other functions would be able to hold down the throughput due to local hosts, by taking most of the buffering for themselves. It is sometimes thought that this is actually a good feature. That is, if the node is so heavily loaded with transit traffic and with traffic destined for output to local hosts, perhaps it is good to reduce the amount of buffer space available for source buffering. After all, when the network is heavily loaded, one does want to reduce the input rate, and reducing the buffer space available for source

buffering of input will have this effect. This argument, however, ignores fairness considerations. In the ARPANET, for example, there are a few nodes which, because they are on the major cross-country paths, have a much greater load of transit traffic than does the vast majority of nodes. However, these nodes which are heavily loaded with transit traffic also have local hosts and TIPs. The users of these local hosts and TIPs have a right to the same service as is given to users whose local IMPs do not have a heavy load of transit traffic. If the heavy load of transit traffic at these nodes is allowed to get so much buffer space that the throughput obtainable by the local users is degraded, then users at these nodes are at a disadvantage with respect to users at other nodes. This is hardly fair. If the transit load at some node is "too heavy," then all users which are sending traffic through that node should be forced to reduce their input rate, not just the users who happen to be locally attached to that node. Of course, this effect cannot be achieved merely by buffer management. It requires a more general congestion control scheme. Our present point though is that since a heavy transit load should not be permitted by itself (in the absence of instructions from a congestion control scheme) to degrade the throughput of local users, a non-sharable pool of buffers should be dedicated to the function of buffering local input while awaiting end-end acknowledgments. Of course, as long as the transit traffic at some node must compete with the input

traffic at that node for some resource (even if only the processor), there will always be a certain amount of "unfair" interference. A good buffer management scheme can limit, but not eliminate, the effect.

It is important to note that this point can be obscured by certain assumptions of homogeneity which it is often convenient to make when analyzing or simulating a buffer management system. When trying to perform such analysis, it is often convenient to create a network model in which the ratio of transit traffic to input traffic is the same at all nodes. Once one has made that assumption, it is clear that the question of fairness will not arise, since all nodes will be equally loaded, and input at all nodes will be equally constrained. Therefore, if one has made that assumption, it may seem reasonable to design a buffer management scheme which allows transit traffic to lock out locally input traffic entirely. Assumptions of homogeneity beg the question of fairness, and in doing so lead to congestion control or buffer management schemes which are seriously deficient.

1.4 Buffer Management with a Shortage of Buffers (ARPANET)

We have so far been discussing the issues that arise in the design of a buffer management scheme for a node which has ample buffer space. We have argued that good buffer management is

important for good network performance, no matter how many buffers exist in a node. Our basic approach has been to dedicate enough buffers to each function which requires them so that all such functions can be performed at full speed, with the minimum amount of interference from other functions. The assumption that there is an "ample" supply of buffer space is just the assumption that there exist enough buffers to do this. Any excess amount of buffers should be sharable among several functions, and should be used to smooth the effects of stochastic peak loads or processor latency.

We turn now to the issues that must be addressed when designing a buffer management scheme for a node which does not have ample buffer space. Our main interest will be buffer management in the 316/516 IMP, which is severely memory-limited. However, our discussion will also have application to the design of a buffer management scheme for new networks which are not expected to be memory-limited. It is often thought that networks designed with present technology will always have ample buffer space, since memory is now one of the cheapest components of a computer. This is somewhat of an oversimplification, though. However cheap memory is, it is always cheaper to have less. We would not expect nodes to be designed with arbitrarily large amounts of buffer space. Rather, the amount of memory configured into a node will generally be determined by making a sizing

decision based both on economics and on the design objectives of the node. Yet at the present state of the art, making such sizing decisions is more of an art than a science, and such decisions can easily be wrong. Furthermore, future re-configurations of the network, e.g., adding long-delay or higher speed lines, can invalidate the original sizing decisions. Yet the addressing, mapping, or bus structure of the computer may make it difficult or impossible to freely add additional memory to the initial configuration. It is never good to assume, in network design, that buffer space will always be ample throughout the life of the network. For these reasons, our discussion of buffer management in the ARPANET should have wider application.

In the ARPANET, each Honeywell 316/516 IMP has between 30 and 35 buffers, depending on the configuration of the node and the presence or absence of various optional software packages (which, when present, reside in an area of memory otherwise devoted to buffer space). This is nowhere near the amount of buffers needed to ensure that all processes requiring buffers can run at full speed. A sensible approach in such a case is to dedicate to each process enough buffers to allow the process to run at only a fraction of full speed, while making the additional buffers sharable. However, unless there are enough sharable buffers to enable some of the processes to sometimes run at full speed, the scheme will prevent any process from ever running at

full speed, even when there are a sufficient number of idle buffers. This would be a very undesirable situation. With a severely memory-limited node, as in the ARPANET, it may be necessary to dedicate to a process only the minimum number of buffers required to ensure that the process can run at all (i.e., to prevent a deadlock situation in which the process is completely locked out). This means that much of the ability of the buffer management scheme to protect one process from undue interference by another is lost. The price for retaining that ability would be to guarantee slow performance by some of the processes, even while resources (buffers) lie idle. Such a price may be too high to pay.

To put this point another way, we must worry not only about under-control of the buffer space, but also about over-control. If buffer space is under-controlled, one process can hog the buffers, preventing other processes from getting their fair share. If buffer space is over-controlled, then a process may be limited to a particular proportion of the buffer space, even if granting it a larger proportion in some particular situation may be the best strategy from the point of view of overall network performance. With ample buffer space, over-control is not generally a problem, since every process can get as many buffers as it needs. When buffer space is scarce, however, strict and inflexible limitations on the amount of buffer space that can be

under the control of a particular process may result in no process ever being able to get enough buffers to perform well. A loosening of the controls may be necessary in such cases. As we shall see, the current ARPANET buffer management scheme suffers from over-control in some instances.

In the ARPANET, the situation is even worse. There are not enough buffers available to dedicate even the minimum amount to certain processes. For example, one process which requires buffers is the process governing output to a host, of which there may be four attached to each node. An ARPANET message may be up to 8 packets long (i.e., may occupy up to 8 buffers). Before any message can be delivered to a host, all eight packets must be present, so that the message can be "reassembled." There is no point to dedicating fewer than 8 buffers to each host, since that would not guarantee that enough buffer space would always be available to deliver a message to the host. On the other hand, one cannot dedicate 8 buffers to each of four hosts, since that would leave no buffers for any other function. A similar problem arises with respect to packets which must be buffered at the source node awaiting end-end acknowledgments (RFNMs). There can be as many as 8 such packets per "connection," where two packets are considered to be on the same connection if they have the same source host, the same destination host, and the same priority. With four source hosts per node, each of which can be

communicating with an arbitrary number of destination hosts, the number of buffers required to guarantee maximum throughput is more buffers than exist in the entire node. However, it is still the case that there are too few buffers to enable a buffer management scheme to ensure fairness to both host input and host output functions. This means, of course, that improving the buffer management scheme can increase the fairness, but not optimize it.

The way the ARPANET deals with this problem is simply to lump together all host input and output functions and dedicate a single pool of buffers to the combined set of functions. This pool is known as the "Reassembly" pool, and its size varies from about 18 to 22 buffers, depending on an IMP's configuration. (The term "reassembly" is very misleading in this context, since reassembly of packets into messages is only one of many functions which must obtain buffers from the reassembly pool.) This approach recognizes that there is simply an insufficient amount of buffering to enable separate pools of buffers to be dedicated to the separate hosts, or even to enable separate pools of buffers to be dedicated separately to input and output functions, without paying the overly high price of ensuring poor performance by some processes even under conditions of low buffer utilization. The main disadvantage of the approach is that it robs the buffer management scheme of its ability to ensure

fairness among the various competing functions that are lumped together. However, that is really just the result of having an insufficient supply of buffers, and we do not see any way of improving the situation simply by altering the buffer management scheme. Attempting to maximize fairness under these conditions requires a strategy other than partitioning the buffer space. The scheme in the ARPANET, though, does make an attempt to separate host-related functions from functions related solely to the operation of the inter-IMP trunks. Failure to separate host-related functions from each other may cause different host-related functions to interfere with each other. Failure to separate host-related functions from operation of the inter-IMP trunks would enable host-related functions to interfere with the node's store-and-forward ability, which could be even worse, since that could make the network more prone to congestion. As we shall see, however, the ARPANET's buffer management scheme is not entirely successful in preventing interference between store-and-forward functions and host-related functions.

Even though fairness between host input and host output functions cannot be guaranteed in the ARPANET simply by partitioning the buffer space, there are other sorts of procedures which a buffer management scheme can bring to bear to help bring about (if not to guarantee) fairness. The present buffer management scheme makes no real attempt to "prioritize"

the input and output functions. That is, if at some given time, buffers are needed for both input and output, the buffers will be assigned in the order in which they are requested. Because of the software architecture of the IMP, this appears to give an advantage to host input. The request for a buffer to hold a packet received from a local host is made by the high-priority routine which services the host-IMP interface. The request for a buffer to hold a packet for output to a local host is made either by the TASK process or by one of the background processes, which run at lower priority levels. Furthermore, requests for output buffers, if not served the first time they are made (because of unavailability of buffers), are put on a queue which is served in round-robin fashion at the lowest priority level. Any number of requests for host input buffers can be served between the time a request for a host output buffer is first queued and the time it is finally served. This seems to violate the principle of congestion control which states that output-related functions should be favored over input-related functions. It would not seem to be a difficult matter for requests for buffer space to be prioritized or re-ordered so that buffers are never provided for input while there are outstanding requests for output buffers. (Note that this issue of re-ordering the requests would not arise if there were ample buffer space, since in that case, all functions could be guaranteed sufficient buffering, regardless of the order in which requests were made.)

This principle, however, would have to be applied with some care. In the ARPANET, a request for output buffer space may be either a request for one buffer (for single packet messages) or a request for eight buffers (for multi-packet messages). If a source node has requested a single-packet allocate for some packet from some destination node, it must buffer the packet until the output buffer space is made available. Meanwhile, other packets from the same source host may still be entering the network. On the other hand, if a source node is waiting for a multi-packet allocate, it does not buffer the multi-packet mesage while waiting. Rather, it stops all input from the source host until the output buffers are allocated. That is, if a single-packet request remains unserved, buffer space is used as the source node, while input at the source node continues unabated. If a multi-packet request remains unserved, not only is no buffer space wasted at the source node, but input from the source host is stopped. The congestion control principle that output should be favored over input is reasonable because "output" means that resources already in use will be freed, while "input" means that resources currently free will be put into use. Competition between a host input packet and an unserved single packet request is clearly competition between input and output. However, competition between host input and an unserved multi-packet request is more like competition between input at one IMP and input at another. Hence, prioritization or

re-ordering of requests for buffers need only be done in the former case. Even there, care must be taken to ensure that a large flow of single packet messages to the hosts at one IMP does not prevent those hosts from ever sending any inputs of their own into the network. While output should be favored over input, output should not be able to lock out input. After all, output at one IMP is input at another. If output is too much favored over input, the result is that input at one IMP is favored over input at another IMP. Therefore, it is possible that, in the absence of a general flow control procedure, which would explicitly match IMP-IMP flows to the amount of resources available, prioritization of buffer requests could do as much harm as good. A full investigation of the issues relevant to end-end flow control in the ARPANET is not within the scope of the present contract, however.

The 316/516 IMP does not have enough buffer space to ensure transmission over the inter-IMP trunks at the full rate of 50 kbps. Only the minimum number of buffers necessary to prevent a trunk from being locked out is dedicated to each trunk. This minimum number, of course, is one. There is also a maximum number of buffers which can ever be under the control of the combined trunk output processes. This number is either 10, 12, or 14, depending on whether the IMP has 2, 3, or 4 trunks. Furthermore, there is also a minimum number of buffers which are

available for trunk output, but unavailable for host-related functions. This number (which includes the single buffer dedicated to each output trunk) is either 6, 9, or 12, depending on whether the IMP has 2, 3, or 4 trunks. (There are certain exceptions to this rule, such as IMPs which have 16-channel satellite lines. See chapter 7 of BBN Report No. 4088 for details. There appears to be no hard and fast rationale for having chosen these particular numbers. Rather, they just "seem to work.") These buffers, except for the buffers which are dedicated to particular trunks, are not, however, dedicated to trunk output; they are also available for other functions that we will discuss shortly. The small difference between the minimum and maximum numbers of buffers available for trunk output (either 4, 3, or 2, depending on IMP configuration) form a pool of buffers which are generally sharable among all the processes in the IMP, which can get them on a first-come, first-serve basis.

There is also a maximum number of buffers which can even be under the control of the process which runs a <u>particular</u> output trunk. This number is eight (except for satellite lines, for which the number is sixteen). The number eight does not appear to have been chosen in order to meet constraints on the buffer management system. Rather, eight is the number of logical channels maintained by the IMP-IMP protocol. That is, it is the number of packets which can be in flight simultaneously on an

inter-IMP trunk. There is no inherent reason why the maximum number of packets under control of an output trunk (i.e. the number in-flight at some instant plus the number queued at that instant) should be the same as the maximum number of packets which can be in flight simultaneously on that trunk. This particular choice of number appears to have been made primarily for ease of programming.

The ARPANET IMP does contain a pool of buffers dedicated to the creation of end-end control messages. In keeping with the principle that, when buffers are in severely short supply, only a minimum number should be dedicated to any particular function, the size of this pool is one. Of course, an IMP may have more than one extant end-end control message at a time. When additional end-end control messages must be created, they are treated as host-related messages. That is, to create an end-end control message, a buffer from the pool for host-related functions must be obtained. This restriction is apparently due to the fact that after a control message is created, it is treated in some ways as if it were a packet submitted by a host. That is, after a control message is created, it is placed on a queue known as the Reply Queue. Packets are removed from the Reply Queue by a "Back Host," and submitted to the IMP as if they came from a real host. A Back Host is a software routine which runs at the background level of the IMP. Its purpose is to

submit control packets as if they were packets from a real host (though of course, they are submitted at a point which is later in the IMP's logic than the point where a real host would submit a packet). This fact about the software architecture of the IMP makes it appropriate to treat the creation of control packets in a manner analogous to host input. If the submission of control packets were handled differently from the submission of ordinary host input, then it might not be appropriate to create protocol messages on the same buffer pool as ordinary host messages, since protocol messages are handled very differently and in general have different constraints. (Of course, one could raise the further question as to whether the "back host" mechanism is appropriate for handling control packets. However, this cannot be considered here.)

We have spoken of the need for having a buffer dedicated to input from each inter-node trunk, in order to be able to process certain sorts of control messages which, although occurring relatively infrequently, need to be processed quickly, with a high degree of responsiveness (i.e., without having to wait on a queue). The IMP does indeed dedicate a buffer to each input trunk. That is, a packet which has just arrived on a certain trunk will not even be queued for the dispatcher (TASK) if that would result in there being no buffer at all available to receive the next input from the trunk. However, these dedicated buffers

are not used for processing those control packets which require high responsiveness. Not only are such buffers not queued for processing, but the packets in such buffers are never processed at all, they are simply discarded. Even if the packet is a line up/down protocol packet, which is ordinarily processed immediately by the routine that handles input from the trunks, it will not be processed if processing it would mean that there is a period of time when no buffer is available to receive the next input from that trunk. Not even the acknowledgments which may be piggybacked in the packet are processed. Rather, the packet is simply discarded, and its buffer reused for the next input. The apparent purpose of this procedure is to ensure that there is never any period of time when a packet can be lost because there is no buffer available in which to receive it. However, although this procedure does help to avoid packet loss, it does this by deliberately discarding packets. From a performance perspective, there does not seem to be much difference between losing a packet and throwing it away. In general, it is not sensible to throw one packet away so that the next will not be lost. Either the buffer dedicated to an input trunk should be used to ensure the processing of packets which need high responsiveness (such as line up/down protocol packets, routing updates, and received IMP-IMP acknowledgments), or there should not be any dedicated input buffers. Currently, the dedicated buffers are wasted. The worst thing a buffer management scheme can do is to waste buffers, particularly when buffers are a scarce resource.

The IMP does have a small pool of buffers which cannot be placed under the control of any host-related process or of any process which regulates output on the inter-IMP trunks. (The size of this pool is regulated by the parameter MINF, currently set to 3.) These buffers are available only for the processing of such high responsiveness packets as routing updates, line up/down protocol packets, and received IMP-IMP acknowledgments, and for the creation of such subnetwork control packets (not end-end control packets) as nulls, routing updates, and line up/down protocol packets. These buffers are also useful for mediating processor latency. They are not, however, dedicated to the individual input trunks. As we have pointed out previously, it is quite desirable to have such a pool of buffers; this seems a good feature of the IMP's buffer management system.

In BBN Report No. 4088 we pointed out several bugs in the IMP's buffer management procedure. One bug was the fact that the buffers which are dedicated to input from the inter-IMP trunks are completely wasted. This bug can be fixed either by refraining from dedicating buffers to trunk input, or by processing the packets in these buffers if (and only if) they require high responsiveness. This latter approach would in some sense be equivalent to increasing the value of MINF to three plus the number of trunks, except that it would also ensure some degree of fairness among the input trunks with respect to their

ability to obtain buffers from the MINF pool. As we have already discussed, the correct way to fix the bug may depend on whether the IMP is short on buffers or short on CPU cycles. Some mixture of the two approaches may be needed, since in practice the IMPs are sometimes short of buffer space and sometimes short of CPU cycles. It must also be pointed out that processing of received acknowledgments from a particular input trunk may also be important if the corresponding output trunk has most of its logical channels in use, even if there are plenty of free buffers. After all, processing of received acknowledgments not only frees buffers, but also frees logical channels, and a shortage of unused logical channels can have the same effect in degrading performance as a shortage of buffers. In order to pick the strategy which will have the best effect on network performance, we will need to design a method of determining in real time which resource is scarcest in the IMP at some particular moment.

We also pointed out several other bugs in BBN Report No. 4088. These bugs all have a common source, namely the fact that when a buffer is moved from a source process to a destination process, the buffer management scheme takes no notice of the source process. In particular, a buffer may be rejected even if it cannot be freed. This not only leads to the bugs we described in our previous report, but also to the following sort of bug.

Suppose an IMP has three trunks, and that it has a maximum of 12 buffers which can be under the control of the process which regulates output to the trunks. Suppose that there are 8 buffers queued for output to trunk 1, and 3 to trunk 2, while there is one buffer which has already been transmitted on trunk 3, but which is presently awaiting acknowledgment. Suppose also that a packet received from a local host is now ready for transmission to its destination, and that it is routed out trunk 3. The IMP will not permit this packet to be transmitted, since that would place a 13th buffer under control of the trunk output routines. Thus the buffer will be rejected, even through the trunk is idle, and the other resources needed to transmit the packet (e.g., logical channels) are freely available. Furthermore, the rejected buffer will not be freed. Refusing the buffer simply delays transmission of the packet without resulting in the freeing of any resource. Thus it has no salutary effect on network performance, and is in fact counter-productive. This is an example of over-control in the buffer management scheme; a buffer is prevented from moving, even though considerations of general network performance would dictate that it be passed to the destination process immediately. This bug, as well as others we have discussed, would be eliminated if the IMP took account of the buffer's source process as well as its destination process. Then the IMP could adopt a policy of never refusing a buffer for considerations of buffer management unless doing so would result in the buffer's being freed.

Even if the ARPANET's buffer management scheme were modified to take account of the criticisms we have been making, there would still be a major problem with it. The problem is that in the competition for buffers to be used to transmit packets to a neighboring IMP, packets input from local hosts are favored over packets arriving from neighboring IMPs, thereby violating an important principle of congestion control. Not only can host access lines be of higher speeds than inter-IMP trunks, but the 1822 protocol, which governs host-IMP access, does not allow the IMP to drop a packet it has received. The IMP-IMP protocol, on the other hand, does allow a receiving IMP to drop a packet. We have already pointed out the way in which this can cause a buffer management scheme to favor the packets from the local hosts. Since it is not feasible to modify the 1822 protocol, some other means of eliminating or at least reducing this favoritism must be developed.

One way of reducing this favoritism would be to define a pool of buffers reserved exclusively for "transit packets", i.e. packets whose origin and destination are both remote. No such buffer pool exists in the ARPANET at present. The current store-and-forward pool can be completely filled with locally originating packets. Although a locally originating packet requires a buffer from reassembly space when it first enters the IMP, it is moved into store-and-forward space as soon as it is

queued to an output trunk. Since locally originating packets cannot be discarded, and hence should never be refused by the buffer management scheme after they are originally received, this division of the buffer pool does not prevent host packets from locking out transit packets entirely. It does prevent all the buffers in the IMP from being devoted to host-related functions, which is very important if the IMP is to continue to function as a store-and-forward node even while handling a large amount of host traffic. Note, however, that a pool dedicated to transit packets would have the same effect. Furthermore, it would have the additional salutary effect of ensuring a supply of buffers for transit packets.

We recommend therefore the elimination of the store-and-forward pool, and the creation of a transit pool. The transit pool would consist of a minimum number of buffers which would be dedicated to packets with remote origins and remote destinations. Locally originating packets would never be placed in the transit pool, but would remain in the Reassembly pool (which we suggest renaming the "end-end" pool), even while queued for transmission out an inter-IMP trunk.

It is also desirable to ensure that a certain number of transit packets may always be queued simultaneously to a given output trunk. Although the presence of the transit pool prevents transit packets from being locked out entirely, it does not

prevent them from being locked out on some particular output trunk. However, since every packet queued for an output trunk must be assigned to a logical channel, this can be prevented by saving a certain number of logical channels on each trunk for transit packets only. This may require that a locally originating packet with a remote destination sometimes be refused, even though the trunk is idle and the refused buffer cannot be freed. However, the reason for refusing in this case is not buffer management, but management of logical channels. Refusing a host packet (destined to a remote destination) for reasons of logical channel management will result in keeping free a logical channel that would otherwise be occupied. So even though no buffer is freed, the packet can still be refused without violating any principles of resource management.

1.5 Detailed Specification of an Improved Buffer Management System for the ARPANET

In this section, an improved buffer management scheme is specified. Its purpose is to resolve the problems with the current ARPANET buffer management scheme that have been discussed in previous sections. Only those aspects of the buffer management scheme which will actually change are described below; aspects of the scheme which are not explicitly discussed will remain as they are now.

1) After receiving a modem input, check to see if another buffer can be put up for the next input. If not, then:

a) If the packet just received contains acknowledgments, process them. This processing is done at modem level. Then give the buffer back to modem input.

b) If the packet just received is a line up/down protocol packet, process it. This processing is done at modem level. Then give the buffer back to modem input.

c) If the packet just received is a routing update, perform the modem-level processing and queue it for the routing process at TASK level. The next time a buffer is freed, give it to modem input.

d) If the buffer contains anything else, just discard it, and re-use the buffer for the next input.

This scheme is an improvement over the present one, in that it does not throw away a packet without even looking at it. However, it does prevent all buffers from going on the TASK queue, which could cause failure to process line up/down packets or routing updates. Strictly speaking, we should not process the acknowledgments if the TASK queue is very long, since in that case we want to save processing time. However, in that case

there probably are not many acknowledgments to process anyway (since in a node with few buffers, if many packets are on the TASK queue, there cannot be many packets in flight), and it might take just as long to decide not to process the acks as it would to go ahead and to process them.

2) We should save n logical channels on each output trunk for transit packets (packets with remote origins AND remote destinations). That is, locally originating packets would be allowed to use only 8-n (or on satellite lines, 16-n) channels on each line. The value of n should be an easily settable parameter, so that experiments can be performed to determine its best value. Hopefully, it will be possible to find a value that will not place a restriction on local throughput. (E.g., it is possible that a local host can obtain its maximum throughput by filling only, say, 6 logical channels, so that when it fills all 8, it does not obtain any additional throughput.) If such a value cannot be found, then a restriction on local throughput may be the price to be paid for additional fairness and congestion avoidance.

3) The transit and end-end buffer pools will replace the store-forward and reassembly pools, respectively. Decisions as to whether TASK can accept a packet will be based on the following quantities:

m -- the number of modems

B -- the number of buffers in the node

TMIN -- the minimum number of buffers in the transit pool

TMAX -- the maximum number of buffers in the transit pool

MAXE -- B - TMIN, the maximum number of buffers in the end-end pool

We do not propose at present to change the internal structure of the end-end (reassembly) pool; everything there shall stay the same. When TASK gets a packet which is already counted in the end-end pool, and which needs to be routed to a remote destination, the only check which TASK makes is the logical channel test (subject to point 2 above). The packet will remain in the end-end pool. When the packet is flushed, of course, the end-end count must be decremented.

When TASK gets a transit packet to go out a particular modem, it makes the following tests:

a) Are there any transit packets already in logical channel slots for that modem? If not, the packet is accepted. The count of transit packets is incremented. Note that this scheme ensures that there are enough buffers to send at least one transit

packet out each line. However, all buffers occupied by transit packets are counted as transit buffers. This eliminates the peculiarity of the present scheme wherein the first buffer on each modem is not counted. (See chapter 7 of BBN Report No. 4088 for details of this peculiarity.) As a result, the counts will not have to be adjusted when buffers are moved to the re-route queue.

b) If there is at least one transit packet already occupying a logical channel on that modem, then the packet is accepted only if accepting it would not increase the transit count above TMAX-q, where q is the number of trunks for which no logical channel is occupied by a transit packet.

c) Of course, both the above tests are subject to the ordinary logical channel test; if there are no channels available, then no packet can be accepted.

Initially we will set TMAX to 6 + 2m in IMPs which have no 16-channel line, and to 15 + 2m in IMPs which do have a 16-channel line. This corresponds to the current setting of SFMAX (defined in BBN Report No. 4088). Currently, SFMAX is the maximum number of transit packets which can be in a node simultaneously. In the new scheme, the maximum number would be

TMAX, i.e., the same. It is more difficult to come up with an optimal setting for TMIN, since any non-zero value will decrease the maximum number of locally originating packets which can be simultaneously buffered in an IMP. (The new scheme will allow only MAXE such buffers; the old scheme does NOT have this restriction.) The initial setting of TMIN will be 2m, but this will have to be an easily modifiable parameter. Optimal values of these parameters will have to be determined empirically.

2. LOGICAL ADDRESSING

2.1 Introduction

In the current ARPANET, in order for a user to transmit data to a particular host, he must know the physical address of the host. That is, he must know which node the host is connected to, and he must know which port on that node is used to connect that host. Furthermore, this is the only means a user has of identifying a host. In many respects, the physical address of a host computer can be compared to a person's telephone number. The problems inherent to physical addressing in a computer network are similar to those we would experience in ordinary interpersonal communication if a person's telephone number were the only means we had of identifying him. Dialing a particular telephone number allows us to establish a communications channel to a particular location. This works well as long as the person with whom we wish to communicate remains at that particular location. When the person changes his location, though, the phone number becomes virtually useless, and the physical address of a host computer becomes equally useless if the computer's location within the network changes. In the context of interpersonal communication, this gives rise to the calling of a "wrong number". In the context of computer networking, this can give rise to the more serious phenomenon of mis-delivery of data. Furthermore, when a computer changes location within a network,

it is quite difficult to carry out a procedure which reliably informs all users of its new physical address. There are difficulties in identifying all users, difficulties in contacting them once they are identified, difficulties in making sure that the information receives the proper level of attention once contact is made, and if all these difficulties are resolved, it is still difficult to make the necessary changes to computer software so that the new physical address is actually put into use.

There is another sort of problem would occur in interpersonal communication if our only means of identifying a person were by his phone number. It is very common for several people to share a phone number. If we identified people only by phone numbers, we could not distinguish among several people at the same location. This problem arises in computer networking if several computers share the same port. There are, in fact, several reasons why it may be desirable to allow several computers to share the same port. One reson is simply the need to get by with a less than optimal amount of equipment, either due to economics or to shortage. If some administration has two computers, each of which needs to be on the network only part of the day, but which do not need to be on the network at the same time, sharing a single port may be the best solution. The increasing cost-effectiveness of such devices as port expanders

and local networks may also make it more and more desirable to have several computers sharing the same port. A related problem arises if one thinks of a network of computers as consisting of "logical hosts," rather than physical hosts. Whereas a physical host would be a particular piece of hardware, a logical host would be the instantiation of a particular function. Thus a physical host which supported (for example) time-sharing during the day and batch processing at night could be regarded as two logical hosts which share the port on a time-division basis. A related application is the situation in which the functionality of two (small) physical hosts is combined into one (larger) physical host, which then can be thought of as consisting of two logical hosts. It could be very useful to have the flexibility to move logical hosts freely around the network, perhaps changing the correspondence between particular logical and physical hosts, without having to inform all users of the new physical addresses.

Of course, the reasons these problems in computer networking are more serious than the analogous problems in interpersonal communication is that telephone numbers are not the only, or even the primary, means we have of identifying other people. We can identify other people by name, and this greatly facilitates our ability to get in touch with people even as they change location. It also enables us to specify the individual we wish to talk to, in the situation where several people share a telephone. Similar

advantages would accrue if we could identify computers by name rather than by physical address. In order to get our data to the appropriate computer, its physical address would still have to be determined. But the user should be able to tell the network the name of the appropriate computer (perhaps a logical rather than a physical host), and let the network itself map the name to its physical address. For speed and reliability, the mapping function should be accomplished automatically (i.e., by software) with minimal need for human intervention. Schemes to accomplish this are known as logical addressing schemes, and the name of a computer is usually referred to as its "logical address" (though in fact, logical addresses are not addresses at all, since they, like names, are independent of location).

A good logical addressing scheme should allow more flexibility than may be already apparent. We have spoken of the need to be able to identify a computer in a way which is independent of location, and of the need to be able to map several distinct names onto a single physical address. There is also a need to be able to map a single name onto several physical addresses. To carry out the analogy with interpersonal communication, this would correspond to the case where a single person has several telephones, with different phone numbers, possibly at different locations. This adds reliability to the communications process, since if the person cannot be reached at

one phone number, perhaps he can be reached at another. If he can be reached at each of the numbers, he now can handle several conversations simultaneously, i.e., he has increased throughput. In computer networking, this sort of application is known as "multi-homing." In multi-homing, a single computer connects to the network through several ports, usually (though not necessarily) at several different network nodes. This allows the computer to remain on the network even though one of its access lines, ports, or home nodes fails, thereby increasing reliability. In the case where it is more economical (or otherwise more practical) to obtain several low-speed access lines than to obtain a single high speed line, multi-homing can also allow a given host computer to obtain higher throughput at less cost.

Another sort of application which requires a single name to map into several physical addresses can be compared to a business which has several branch offices, each with a different phone number, but whose customers do not care which branch office they reach. This can be useful in certain sorts of internetting applications. Suppose, for example, that an ARPANET user wants to send a packet to SATNET, but that there are several equally good gateways between the two networks. It may be convenient for the user to simply specify a name like "Gateway-to-SATNET" and let the network choose which of the several gateways (i.e.,

physical addresses) to use. A related sort of possible application has to do with distributed processing and resource sharing. If some particular resource is available at any of several locations around the network, it may be desirable to allow the user to specify the resource by name, and allow the network to map that name onto some particular physical address according to criteria that the user need not be aware of. (Such a service was formerly offered by the ARPANET TIPs. By typing @N, a user would be connected to a "Resource-Sharing Executive" on one of several network TENEX systems. The user, however, would have no way of knowing which system he was actually on.)

2.2 Translating Logical to Physical Addresses

2.2.1 Translation Locus

It should be obvious that any implementation of logical addressing would require the network to maintain a translation table. The user would specify a logical address, and the network would use this logical address as an index into the translation table in order to obtain the physical address (or list of physical addresses) to which it corresponds. The network would use the physical address internally to determine the routing of user messages. The need to maintain a translation table gives rise to a multitude of design issues. The first question that needs to be answered is, where should the translation table be

located? Should every node maintain a copy of the whole translation table, or should there be just a few copies of the table scattered around the network in strategic locations? If there are only a few copies scattered around the network, then nodes which do not contain the tables would have to query the nodes that do in order to perform the translation function. This is less efficient (both in terms of overhead and response time) than placing the table in every node. Considerations of reliability and survivability also favor placing the table in every node. This eliminates the possibility of finding that, due to network partition, all copies of the translation table are inaccessible. It eliminates the need to have some nodes serving as hot or cold standby for the nodes which do have the tables. This is an important advantage, since the protocols needed to implement "standby" tend to be slow and cumbersome, or else unreliable. Furthermore, if all nodes maintain identical copies of the translation table, there is no need to go through any special initialization procedure for creating the table when a node first comes up. Typically, a node which is just coming up has been reloaded from a neighboring node. If all nodes have an identical copy of the table, a node coming up can simply have its table reloaded from its neighbor, i.e., can copy its neighbor's table. (Under certain unusual conditions this may give rise to a race condition, but as we shall see later, it is a race that can be easily remedied and one that will not have any bad effect other than to slow the reload process.)

There is, however, a possible disadvantage to having the tables in every node. That is simply the need to have enough memory in every node to hold the table. In certain networks, particularly commercial ones, the network nodes may be of widely varying sizes and capabilities, and the smaller nodes just may not be able to hold the complete translation table. Such networks would necessarily have a hierarchical structure (probably with some form of hierarchical routing), and a node would not be able to hold a translation table unless it were at or above a certain level in the hierarchy. However, this situation does not apply either to AUTODIN II or to the ARPANET, the two networks with which we are presently concerned. We will discuss later some of the issues having to do with table size, and we shall see that the translation table can be coded in two or three words per logical address. This should not offer any problem in the AUTODIN II environment. In the ARPANET, there is more of a problem. Currently the IMPs are restricted to 16K of memory, and there is not enough room in any IMP for the translation tables, or even for the code needed to perform the translation. When the 32K C30 IMPs become available, however, there will be sufficient room to have a complete table in each one. (These claims are based on the assumption that there will be fewer than 500 logical addresses in each network. If this assumption is wrong, the claims may need to be re-evaluated. Note that the size of the table will be determined by the number

of logical addresses, which may be larger than the number of hosts.) As long as some 16K IMPs remain, however, there will be some IMPs that cannot hold the translation tables. That does not necessarily violate our dictum that every node contain a copy of the translation table, though. Strictly speaking, only those nodes which will ever need to translate a logical address to a physical address will need to have a copy of the translation table. Whether that is all nodes or only a subset of the nodes depends on the translation methodology that we adopt. We have a choice between requiring translation to be done only at the source node, or requiring it to be done also at tandem nodes. In the former case, when the user presents some data to the network and specifies its destination with a logical address, the source node looks in the translation table, gets the physical address, places the physical address in the packet header, and sends the packet on its way. Tandem nodes do not look at the destination logical address at all, but only at the physical address. In the other case, each tandem node looks at the logical address, does its own translation to physical address, and routes the packet on that basis. The packet header would not even have to contain the physical address. If we do source translation only, the only nodes which need to contain translation tables are those that connect to hosts which use logical addressing. This is attractive for the ARPANET, since we could declare that any host which wants to use logical addressing would have to access the

network through a 32K C30 IMP; hosts accessing the network through 16K IMPs would then be prohibited from using logical addressing. Therefore, we must look at the advantages of source node vs. tandem node translation.

2.2.2 Translation Methodology

Clearly, in the case where a logical address maps to a unique physical address, source node translation is superior to tandem node translation. As long as there is only one possible physical address for that logical address, all nodes will produce exactly the same mapping. There is thus no advantage to performing the mapping several times, and the scheme which does it only once is more efficient. There is, however, one exception to this. When the translation tables need to be updated, we cannot expect all copies to be updated simultaneously. There will necessarily be some short interval of time when not all of the copies of the table around the network are identical, and during this interval, tandem node translation may yield different results than source node translation. It will certainly be necessary to design some mechanism to deal with this problem, and we shall propose one shortly. Tandem node translation, however, is not the right solution to this problem. During the transient period, some copies of the table will be right (up to date) and some wrong (out of date). But the copies at the tandem nodes will be no more likely to be right than the copy at the source

node, so tandem node translation would be as likely to amplify the problem as to reduce it. The solution to this problem lies elsewhere.

In the case where a logical address maps to several physical addresses (multi-homing), tandem node translation might well give different results than source node translation. However, we must now distinguish between virtual circuit and datagram traffic. If virtual circuit traffic is logically addressed, all translation must be performed at the source node. In fact, the translation must be performed only once, at connection set-up time. This is the only way to ensure that all traffic on a given circuit is sent to the same physical address, which in turn is the only way to provide the sequencing and duplicate detection that is the raison d'etre of virtual circuit traffic. (Additional issues having to do with logically addressed virtual circuit traffic will be discussed later.) Thus the only possible advantage of tandem node translation would have to do with datagram traffic which is destined to a multi-homed logical address. To understand the differences, though, between source node and tandem node translation, we must first discuss the criteria which a node uses to pick one physical address out of the several that are available. (Even though a host is multi-homed, we would want to send each packet for it to only one of its physical addresses; some criterion for choosing the proper one in each particular

case must therefore be available.) Several possible criteria come readily to mind:

a) When there are several physical addresses corresponding to a given logical address, it may be desirable to send packets to the physical address which is closest to the source node, according to some metric of distance. For example, if we are interested in minimizing delay, we may want to choose the physical address to which the delay from the source is least. If SPF routing is used, this information is readily available. If we are interested in minimizing the case of network resources by a particular packet (i.e., in maximizing throughput while still using orly a single path), we may want to choose the physical address which is the least number of hops from the source. (For these purposes, ties can be broken arbitrarily.) Again, this information can be made readily available by the SPF routing algorithm (although it is not readily available from the ARPANET's particular implementation of that algorithm, since a table of hop-counts is not saved). If either of these criteria is used, tandem node translation can result in better route selection for the logically addressed datagram. If delay changes or topology changes take place while a packet is in transit, it may happen that the "closest" physical address to some tandem node is different from the physical address that was closest to the source node when the packet first entered the network. It is

easy to prove that, if SPF routing is used, this cannot result in looping, except as a transient phenomenon while a routing update is traversing the network. That is no particular disadvantage, since a packet may be subject to that sort of transient looping even when its destination physical address does not change. However, it is not clear that tandem node translation provides much of an advantage either, especially when one takes into account the additional overhead of doing the re-translation at each tandem node. Doing translation at tandem nodes will necessarily increase the nodal delay and decrease the nodal throughput. These negative effects may outweigh the positive effects of improved route selection for those relatively rare cases in which delay or topology changes significantly while a packet is in transit. We must remember that although real improvements in route selection would only occur rarely (since delay and topology changes are very infrequent when compared with average network transit times), re-translation would have to be done for every logically addressed datagram. Unfortunately, all these effects are extremely difficult to quantify with any degree of confidence. Our (somewhat intuitive) conclusion is that, under the selection criterion of choosing the closest physical address, tandem node translation offers at best a small improvement over source node translation, and at worst a severe degradation.

b) It is possible that some multi-homed hosts will want to establish an inherent ordering to their ports. That is, they may prefer to receive all their traffic on port A, unless that port is inaccessible to the source of the traffic, in which case they prefer to receive all traffic on port B, unless that port is inaccessible to the source of the traffic, in which case they prefer to receive all traffic on port C, etc. This sort of strategy may be appropriate if certain of the host access lines are charged according to a volume-based tariff, while others are not. It may also be appropriate if certain of the access lines can be used more efficiently (i.e., can be serviced with less host CPU bandwidth) than others. (An example might be a host which can access the ARPANET through an 1822 line and a VDH line. It might be desirable to avoid the VDH line, unless absolutely necessary, since VDH lines tend to be used less efficiently.) In either case, the idea would be to reduce cost by favoring certain ports over others, using the more expensive ports only when needed for purposes of reliability or availability. Thus we may want the logical addressing scheme to support an inherent ordering among the several physical addresses which correspond to a given logical address. With this scheme, there is no advantage to doing tandem node translation. There will be only one ordering for the set of physical addresses corresponding to a logical address, so tandem nodes should always pick the same physical address as the source node picked, and re-translation

would simply be a waste of resources. There are only two exceptions to this. The first exception would arise in the situation where a particular physical address becomes inaccessible as a packet routed to that address is traversing the network. However, since this can happen no matter what criteria are used for choosing among the physical addresses, we put it off for later consideration. The second exception would arise if the translation tables were being updated while a packet is in transit. Clearly, some procedure to deal with this case must be devised, since the update which is taking place may invalidate the translation which was done at the source. Tandem node translation is not the proper solution, however, since there is in general no reason to believe that the tandem node is more up-to-date than the source node. We will return to this issue when we discuss the table updating procedures.

c) Certain multi-homed hosts may have a preference for receiving certain kinds of traffic over particular ports. Thus a dual-homed host may consider one of its ports more suitable for receiving batch traffic, and another more suitable for receiving interactive traffic (perhaps the first port offers a higher speed but a longer delay than the second). However, if one of the ports is inaccessible from a particular source node, that node would send all its traffic (both kinds) to the port which remains accessible. With this criterion, we see once again that tandem

node translation offers no benefits, since, barring the case of a port becoming inaccessible while a packet is in transi . all tandem nodes would select the same physical address as the source node.

d) Some multi-homed hosts may wish to try to keep their several access lines as equally loaded as possible. One possible way to do this would be to establish an inherent ordering to the ports (as in b, above), but to make the ordering different for different source nodes (or hosts). Clearly, this scheme requires source node translation; tandem node translation would only serve to defeat it. Another possible way to achieve some sort of load leveling would be for each source node to send traffic to the various physical addresses on a round-robin basis. This would be a very crude form of control, but might work reasonably well for particular traffic patterns. This scheme also requires source node translation. In fact, tandem node translation could actually cause packets to loop endlessly.

We see then that none of the suggested selection criteria give any very large advantage to tandem node translation, and some of the criteria are actually in conflict with re-translation at tandem nodes. It thus seems acceptable for the ARPANET to implement logical addressing only in the 32K nodes, with the restriction that hosts which use logical addressing must access the network at such nodes.

2.3 Organizing the Translation Tables

Another issue having to do with the translation tables is the way in which the tables should be organized. Clearly we do not want the entries in the table to be in random order, necessitating a lengthy linear search each time a translation must be done. Basically, there are two possible ways to order the table. We can sort the table by logical address, and do a binary search of the table whenever we need to do a logical-to-physical address translation. This is a rather efficient form of searching, but it causes inefficiencies when table entries have to be inserted or deleted, since that would require a potentially time-consuming expansion or compression of the table. There are, however, ways of reducing the overhead involved in insertions/deletions. Deletions could be made "logically", i.e., by marking an entry deleted, rather than physically compressing the table. New entries could be inserted into an overflow area, which itself would be searched linearly whenever a particular logical address could not be found in the main table. Actual compression/expansion of the main table would be done only when the overflow area filled. Note, however, that this sort of strategy would necessarily complicate the search algorithm, and this might actually do more harm than good, especially if insertions and deletions are rare events. We shall see later, when we discuss the conditions under which insertions

and deletions are required, that these are indeed rare events. We conclude tentatively that, if a sorted table with binary searching is used, the use of an overflow area is probably not necessary.

The other possibility for organizing the table is to use hashing. A good hashing algorithm (i.e., one which minimizes collisions) provides very efficient insertions and very efficient searches. Deletions are not quite so efficient, but are still more efficient, in general, than the table compression required if binary searching is used. However, hashing has certain inherent problems which may make it less suitable. Choosing a hashing algorithm which both minimizes collisions and is computationally efficient is not a simple matter. One must be sure that the time needed to perform the hashing is really less than the average time needed to find an entry in a sorted table by means of binary searching; otherwise, the efficiency is lost. Furthermore, the number of collisions generated by a particular hashing algorithm will depend on exactly which set of logical addresses are in use. The set of logical addresses in use during a network's lifetime will be a slowly changing set, and a hashing algorithm which is excellent at one time may give poor performance at another. Hashing algorithms are also subject to undetected programming bugs in a way in which binary search algorithms are not. A bug which is inserted into a hashing

algorithm, which, for example, causes all entries to hash into the same bucket, might go undetected for years, although it would cause a significant performance degradation by reducing the efficiency of the hashing technique to that of linear searching. A bug in the binary search algorithm, however, would be more likely to come to someone's attention. Its probable result would not be performance degradation, but rather, failure to find certain entries. This would cause inability to deliver traffic to certain logical addresses, and this would certainly come to the users' attention very quickly. These qualitative reasons would seem to indicate that binary searching is preferable to hashing. It would also be useful to do some quantitative analysis; that may be done at a later stage of our research.

Someone may wonder why we have not considered a much simpler method of organizing the tables. Logical addresses, after all, are just numbers (or at least are representable as numbers). If the set of logical addresses in use at any given time is a contiguous set of numbers, then the addresses can be used as indexes directly into a translation table, with no need either for hashing or for sorting. The problem, of course, lies in the requirement that the set of logical addresses form a contiguous set of numbers. Assigning numbers to hosts contiguously may not be a problem in itself, but it does cause a problem as soon as some host is removed from the network. Its number (or numbers,

if it has several logical addresses) cannot be left unused, or the size of the translation table would be determined not by the number of logical addresses currently in use in the network, but rather by the number of logical addresses that have ever been in use in the network, a number which may be much larger, and which in fact has no bound. Yet it is not acceptable simply to reassign the same logical addresses as hosts enter or leave the network. We all have some experience with moving to a new location, getting a new telephone number, and finding ourselves frequently getting calls intended for the person who previously had that number. Such calls may persist for years, especially if the number previously belonged to a business. Receiving phone calls or mail intended for someone else who happens to have the same name as we do is also a familiar occurrence. We would expect analogous problems if logical addresses are reassigned (at least, if they are reassigned without some very long waiting period), especially if the logical address previously belonged to a large service host. When a user tries to address a host which is no longer on the network, he should receive some indication of that fact; he should not have his data mis-delivered to some other host which has been assigned the same name. Thus it is preferable to have a logical addressing scheme which does not depend on the logical addresses forming a contiguous set of numbers.

Whatever means of organizing the table is chosen, it may still be useful to maintain a smaller table for use as a "cache." The cache would contain the n most recently used logical addresses (where n is some small number), together with a pointer to the absolute location of that logical address entry in the main translation table. When it is necessary to do translation, the cache would be searched before the main table. The assumption is that once one packet for a particular logical address is received from a source host, many more will follow. Thus it pays to optimize the search for that particular logical address. Choosing the optimum size for the cache, and the means of searching it (linear or binary) are issues left for later resolution. These issues must be dealt with carefully, however; one would not want to find that searching the cache takes as long or longer than searching the main table. It is worth emphasizing that the cache should contain a pointer into the main translation table, rather than a copy of the list of physical addresses associated with a particular logical address. For multi-homed logical addresses, this is more efficient, since it involves less copying. Also, if there are variables associated with the physical addreses, this enables unique copies of the variables to be kept in the main table. (Suppose, for example, that packets are to be sent on a round-robin basis to the several physical addresses corresponding to a multi-homed logical address. This requires a variable to be associated with each physical address,

indicating whether that physical address was the last one to be sent data. This variable must be kept in the main table, not the cache, since one cannot rely on a particular logical address always being present in the cache.) This means, of course, that the cache must be cleared whenever there are insertions or deletions into the main translation table, but that should not be very expensive as long as such insertions and deletions are relatively rare.

2.4 Initializing the Translation Tables

We turn now to the issue of how the translation table entries are to be set up in the first place. That is, what procedure is to be used for establishing that a particular logical address is to map to a particular set of physical addresses. One possibility, of course, is to have all the mappings set up by the Network Control Center (NCC). This is quite reasonable in certain cases. If some user wants his computer to be addressable by some new logical address (i.e., by a logical address not previously in use), it makes sense to have him contact the NCC directly. If the user has proper authorization, the NCC can then take action to set up the new entry in all the translation tables. A similar procedure would also be appropriate if some logical address is to be totally removed from the translation tables (i.e., that logical address will no longer be in use in that network). This procedure would

also be appropriate when a particular computer is moved from one location to another, necessitating a change in its logical-to-physical mapping, or if the functionality of two computers is combined into one, so that two logical addresses which formerly mapped to distinct physical addresses now map to the same physical address. What all these cases have in common is that they are relatively infrequent (i.e., occurring on the order of days, rather than on the order of minutes), and they require considerable advance planning. The first of these characteristics ensures that NCC personnel will not be swamped with translation table changes. The second of these characteristics makes it feasible to coordinate such changes in advance with the NCC. Unfortunately, not all translation table changes have these characteristics. For example, we have suggested that a good logical addressing scheme should facilitate port-sharing. That is, some user might want to unplug one of his computers from the network and use that port for another computer. He should be able to do this without much advance planning, and without having to explicitly coordinate with the NCC. As soon as the change is made, users who are logically addressing the first computer should be told that it is no longer on the network; only the logical address of the second computer should map to this port. If this change in the mapping does not take place immediately, the result can be mis-delivery of data, as packets which are logically addressed to the first computer

get mis-delivered to the second. A similar situation arises if some computer consists of several "logical hosts." Logical hosts may come and go quite frequently, with no advance planning at all. The logical addressing system should be able to adapt immediately to such changes, without any need for human intervention. A related situation arises in the case of logical addresses which are multi-homed. We have already discussed various possible criteria for choosing among the several physical addresses associated with a single multi-homed logical address. But before applying these criteria, any physical addresses which are "inaccessible" from the source node must be excluded. If some host has two access lines into the network, and one of them is inaccessible from a particular source node, then all traffic from that source node should be directed to the other access line. Indeed, this is one of the most important purposes of multi-homing. This implies that a source node must have some way of knowing that a certain physical address is not currently accessible. There are basically two classes of reasons why a given physical address might be inaccessible from a source node. The first is that there is no path from the source node to the destination node (either because the network is partitioned, or because the destination node is down). This information is readily available from the routing tables, and need not be kept in the translation tables. It is simple enough to check the routing tables when choosing one from a set of physical

addresses. The other reason why a physical address might be inaccessible is that the port itself, or the access line from the port to the host, has failed. Functionally, this is the equivalent of unplugging the host from the port. It may happen quite frequently, however, and certainly with no advance planning. As long as the node itself is up, the routing algorithm will give no indication that the port is inaccessible; this information must somehow get into the translation tables. Clearly, we do not want to depend on human intervention to ensure that this sort of change gets made in the translation tables. What is needed here is a quick and reliable means of making changes in the translation tables, not the cumbersome and unreliable method of contacting the NCC. The same problem arises when the inaccessible port becomes accessible again. One wants to be able to begin using this port again as soon as possible, without having to wait until NCC personnel have time to make the appropriate changes in the translation tables. So although certain sorts of changes to the translation tables can be made by the NCC, many sorts of changes will occur suddenly and unexpectedly, and need to become effective immediately. So the procedure of having all translation table changes made by the NCC is not satisfactory.

There is another sort of problem with having translation table changes made by the NCC. The problem is that carelessness,

either by the NCC or by host site personnel, can result in mis-delivery of data if changes are made by the NCC. Suppose, for example, that a network controller makes a typographical error, associating a logical address with an incorrect physical address. If there is no further check on the validity of that mapping, one computer may receive data intended for another. A good logical addressing scheme should prevent this sort of simple typographical error from resulting in mis-delivery. The same situation can occur if one computer is carelessly plugged into the wrong port. In this case, networks which use only physical addressing might also mis-deliver data. However, with physical addressing, one must expect mis-delivery if some computer is plugged into the wrong port (i.e., given the wrong physical address) due to carelessness. With logical addressing, this is not inevitable, and a good scheme should give better protection against carelessness.

Another possibility for setting up the translation table entries is to have each host, as it comes up on the network, tell the network which logical addresses it wants to be addressed by over each of its (physical) ports. This would require augmentation of the host access protocol to include a "Logical Address Declaration" (LAD) message. A given host could put as many logical addresses as it wanted in each LAD message. Multi-homed hosts would send the same LAD message over each of

their ports. The logical addresses specified in the LAD message received over a given port would all be mapped to that particular physical address. Hosts would be allowed to change their logical addresses at any time by sending a LAD message to the network. Since a host may wish to add or delete logical addresses for itself at any time, there would have to be two options for the LAD message -- "add" and "delete." Whenever a particular port goes down (either because the port itself fails to function properly, or because the access line between the network and the host fails, or because the host itself crashes), all mappings of logical addresses to that port would be cancelled. When the host can once again communicate with the network through that port, it would have to redeclare its logical addresses with a LAD message before it could receive any logically addressed traffic.

Allowing each host to set up its own logical-to-physical address mappings in this manner has several advantages over having all the mappings set up by the NCC. This procedure allows sudden and unplanned mapping changes to take effect immediately, with no need for advance planning and coordination with the NCC. Since the mappings are cancelled immediately when a port goes down, this procedure helps to ensure that, if one of a multi-homed host's ports is down, all data which is logically addressed to that host will go to the other ports. If one host is unplugged from a given port, and another plugged in its place,

the procedure ensures that the mapping for the first host is cancelled, while the mapping for the second host becomes effective. When a host goes down, there is no assumption that the same host will return in the same location. Hence carelessness on the part of site personnel or NCC personnel cannot result in mis-delivery of data; data which is logically addressed to a certain host could only be delivered to a host which has declared itself to have that logical address.

There are, however, two quite serious problems with this procedure. The first problem is that of spoofing. That is, this procedure offers no protection against the situation where one host declares itself to be addressable by a logical address which is supposed to be the logical address of a different host. Thus the procedure allows one host to "steal" traffic intended for another, simply by declaring itself to have the same logical address as the other. This sort of spoofing might be done by a malicious user, who is really trying to steal someone else's data, or it might happen accidentally, as a result of programmer or operator error. In either case, we would like to have some procedure which is less prone to spoofing. The other serious problem with this procedure is that it can easily cause the translation tables to overflow in size. If every host can specify an uncontrolled and unlimited number of logical addresses for itself, there is no bound on the size of the translation

tables. Since only a finite amount of memory will be available for the translation tables, it is clearly not acceptable to allow each host to specify an arbitrary number of logical addresses for itself.

2.5 Updating the Translation Tables

We have examined two different procedures for setting up the logical-to-physical address mappings, and have found that they both have problems. Many of these problems can be resolved, however, by a combination of the two procedures. Let us define two characteristics of a logical-to-physical address mapping, which we will call "authorized" and "effective." A mapping from a particular logical address to a particular physical address is "authorized" if a host which connects to the network at that physical address is allowed to use that logical address. Authorizations would change very infrequently, and only after considerable advance planning. Hence it is appropriate for authorizations to be determined (i.e., added and deleted) by the NCC. A mapping from a particular logical address to a particular physical address would be said to be "effective" from the perspective of a given source node if (1) that mapping is authorized, (2) that physical address is accessible from that source node, and (3) the host at that physical address has, by means of a LAD message, declared itself to have that logical address. When a port goes down, all mappings to it will become

ineffective, until they are made effective again by means of a LAD message. Logically addressed traffic will not be delivered to a particular physical address unless the mapping between that logical address and that physical address is effective. Changes in the effectiveness of a mapping will occur automatically, in real-time, with no need for intervention by NCC personnel. This facilitates multi-homing, since if there are two authorized mappings of a logical address to a physical address, and only one is effective, that one can be chosen all the time until the second becomes effective also. It facilitates sharing of ports (either by physical or by logical hosts), since each host has control over the effectiveness (though not the authorization) of the mappings that affect it. Carelessness by NCC personnel can cause the wrong mappings to become authorized, but it is rather unlikely that an incorrectly authorized mapping could become effective -- that would require carefully planned malicious intent. Therefore, such carelessness might prevent delivery of data to some host, but would not cause mis-delivery of data. Carelessness by site personnel, such as plugging a host into the wrong port, would not cause mis-delivery of data, since the mapping of that host's logical address to that particular port would not be authorized. The possibility of spoofing is greatly reduced; since host A cannot pretend to be host B unless it is at a port which is already authorized for host B. The size of the translation table cannot increase without bound, since that is

determined by the number of authorized mappings, and cannot be increased by LAD messages. This means, of course, that the network access protocol must be further modified so that it can provide positive and negative acknowledgments for the LAD messages. For each logical address that a host specifies for itself in a LAD message, the network must return either a positive or a negative acknowledgment. The positive acknowledgment would indicate that the mapping is authorized and has become effective. The negative acknowledgment would indicate that the mapping is not authorized.

It must be emphasized that the suggested procedures are not intended to provide security in any very strict sense. For networks in which security is a very important issue (e.g., AUTODIN II), further study of these issues should be carried out by security experts.

It should also be emphasized that these procedures will allow the logical addressing scheme to continue to function normally even if the NCC facilities are down. It does require centralized intervention to add or delete authorizations, and this could not be done if the NCC were down. For a fixed set of authorized mappings, however, no centralized intervention is required to determine the effectiveness of the mappings. That is, the real-time functionality and responsiveness of the logical addressing scheme does not depend in any way on the proper functioning of the NCC.

We have argued that the authorization of a mapping should be determined by the NCC, and the effectiveness of a mapping should be determined by the network node which contains the physical address (port) to which the mapping is made, in cooperation with the host that is connected to that port. We have also argued that a full translation table (i.e., a table containing all the effective mappings) should be stored at each network node (or more precisely, at each network node which serves as an access point for a host which can be either a source or a destination of logically addressed traffic). However, we have not yet discussed the algorithm by which the effectiveness or ineffectiveness of a particular logical-to-physical address mapping is communicated to all network nodes. We turn now to this issue. We will discuss two very different methods for building up the translation tables at all nodes.

The first method is based upon an extension of the SPF routing algorithm, wherein each logical address is treated like a stub node. In this method, each node is initialized with a partial translation table. This table contains a list of all the logical addresses which are authorized to map to that node, (i.e., all the logical addresses which correspond to ports at that node). Each of these logical addresses is associated with a particular port or ports at that node. At initialization time, each of these logical addresses is treated just as if it is a

neighboring node which is down, and the node sends an update (similar to a routing update) to all other nodes, indicating that all authorized mappings to itself are ineffective. When a host comes up over a particular port, it declares its logical address(es) by means of one or more LAD messages. The node then checks its table of authorized mappings, and acknowledges to the host (either positively or negatively) each logical address mentioned in the LAD message. Whenever a logical address is positively acknowledged, it becomes effective, and the node must broadcast an update to all other nodes declaring that mapping to be effective. Whenever a host declares (via a LAD message) that it no longer wants to be addressable by a particular logical address, an update must be generated declaring that mapping to be ineffective. Whenever a port goes down, all logical addresses mapping to it become ineffective, and an update indicating this must be broadcast. If the protocol used to disseminate these updates is the same as the protocol used in the ARPANET to disseminate the updates of the SPF routing algorithm, then all nodes will be able to build dynamically an up-to-date table of effective mappings, just as the routing updates enable them to build an up-to-date topology table. (The procedure used to build the topology tables is described in [1]. The updating protocol is described in [2] and [3].) In effect, this procedure extends the routing algorithm to treat the hosts (or more precisely, the mappings of logical addresses to physical addresses) as stub

nodes, and the ports as lines, except that there is no delay associated with a port but only an up/down status.

This procedure is attractive from a conceptual point of view, but it is not really cost-effective. That is, it seems to be too expensive to be practical. One reason is that it is hard to place a bound on the size of the updates. The updating protocol of the ARPANET routing algorithm is quite efficient, because the updates are so small. The maximum update size is only 216 bits (from a node with 5 neighbors) The logical addressing updates might be much longer, since there is no limit on the number of logical addresses that may map to a given node. The updates would also have to be sent periodically, even when there in no change in state. These features are necessary to ensure reliability in the face of such events as partitions, node crashes, updates received out of order, etc. With no restriction on the number of logical addresses which can map to a given node (and it seems unwise to build in such a restriction), there is no restriction on update size, and hence no bound on the bandwidth needed for updating, or on the extra delay which may be imposed on data packets due to the need to transmit the updates. Another disadvantage of the protocol is that it requires the use of a broadcasting protocol, which would have to be implemented in all network nodes, not just the ones which need logical addressing. This could make it completely unfeasible for the ARPANET.

The updating protocol which was designed for the SPF routing algorithm was designed to get the updates to all nodes very quickly, and with 100% reliability, even in the face of various types of network failures. This extreme speed and reliability is necessary for routing updates, since rapid and reliable updating of the routing tables is necessary to ensure the integrity of the network. Routing failures, after all, can make the network completely unusable, and can be very difficult to recover from, since most recovery techniques depend on the NCC's ability to communicate with the nodes, which in turn depends upon the integrity of the routing algorithm. Fortunately, the protocol used for disseminating the logical addressing updates does not need all the functionality of the updating protocol used for routing, since the integrity of the logical addressing scheme is not quite as critical as the integrity of the routing algorithm. This enables us to use a simpler and less expensive method of maintaining the translation tables, which we will now discuss.

In this second method, each node is initialized with a translation table containing all the authorized mappings. This table would have an entry for every logical address that can be used in the network. Each logical address would be associated in the table with all the physical addresses to which it has an authorized mapping. Associated with each of these physical addresses would be a Boolean variable indicating whether that

particular logical-to-physical address mapping is effective or ineffective. At initialization time a node would mark all mappings to itself as ineffective, and all mappings to other nodes as effective. Whenever a host declares itself, via a LAD message, to have a certain logical address, the node looks in the translation table to see if that mapping is authorized. (This is just an ordinary table look-up, indexed off the logical address.) If not, a negative acknowledgment is sent to the host. If the mapping is authorized, a positive acknowldgement is sent, and the entry in the translation table is marked effective. Whenever a port goes down, the node marks all mappings to that port as ineffective. Of course, this also requires a "reverse" search of the table (i.e., a search based on a physical, rather than logical address). To make this more efficient, when the initial reverse search is done at initialization time, the node can save a list of pointers into the translation table. Each pointer would correspond to a physical address entry for that node. If a separate table of pointers is kept for each port, the node will be able to find in a very efficient manner entries which map to a particular port. Using this methodology, each node's translation table will correctly indicate, for each of the logical addresses that map to it, whether or not that mapping is effective. (Of course, these pointers would have to be adjusted whenever table insertions or deletions are made.)

When a source host sends a logically addressed datagram packet into the network, the source node will search the translation table for the correct mapping. If that logical address cannot be found, i.e., its use is not authorized, an error message indicating this fact should be returned to the host, and the packet discarded. If that logical address is found, but all the corresponding physical addresses are either marked ineffective, or else are unreachable (according to the routing algorithm), then the packet should be discarded, and the host informed of that fact. If some of the physical addresses are both reachable (according to routing) and marked effective, then one should be chosen, according to some set of criteria (perhaps one of those which we discussed above). The chosen physical address should be placed in the packet header, along with the logical address. The packet should then be forwarded to its destination; in doing the forwarding, tandem nodes will look only at the physical address. According to the procedure described in the previous paragraph, all mappings to remote ports will be initially marked effective. To see how such mappings can get marked ineffective, we must see what happens when a logically addressed packet reaches its destination node.

When a logically addressed datagram packet reaches its destination node, the node looks up that logical address in its translation table. It is possible, of course, that that logical

address will not be found at all, or that it will be found, but that there will be no authorized mapping to this particular destination node. This would indicate some sort of disagreement between the translation tables at the source and destination nodes. There are three possible causes of this disagreement: (1) NCC error in setting up the translation tables, (2) deletion of the authorization for that particular logical address while the packet was in transit, or (3) a race condition, whereby a translation table update authorizing the new logical address is taking place, but the update has not reached that destination node yet. In any case, the data packet should be discarded without delivery, and an error message should be sent to the NCC indicating receipt of a packet with an unauthorized logical address. This will alert NCC personnel to a possible error. If the authorization for that logical address was deleted while the packet was in transit, however, then the NCC need not take any action; having the destination node simply discard the packet is the correct procedure. If, on the other hand, that logical-to-physical mapping is really authorized, but the update making the authorization has not yet reached the destination node, then we want to take the same action as we would take for a packet delivered according to an authorized but ineffective mapping. This action shall be described in the next paragraph.

Suppose that, upon looking up the logical address in the translation table, the destination node does find an authorized mapping to itself, but that mapping is marked ineffective. Then there are two actions to take. The first action is to try to re-address and then re-send the message. Of course, this can only be done if the destination logical address is multi-homed, and at least one of the corresponding physical addresses is effective. If this is not the case, the packet must be discarded. The second action is to send a special message back to the source node of that datagram packet. We will call this message a "DNA message" (for "Destination Not Accessible"). The DNA message will specify that the particular logical-to-physical address mapping used for that packet is not an effective mapping. The DNA message should also be sent in response to datagrams which appear to have unauthorized mappings (see previous paragraph). For reliability, each logically addressed datagram must carry the physical address of its source node (though not of its source host), so that the DNA message can be physically addressed to the source node. It is not enough for the packet simply to carry the logical address of its source host, for two reasons. The first reason is that if the source host is multi-homed, the destination node will not know which source node the packet came from, and hence will not know where to send the DNA message. The second reason has to do with the fact that one situation in which DNA messages may have to be sent is the

situation in which the translation table at the destination node has been set up erroneously. In this case, we do not want to have to rely on the integrity of the translation table to ensure proper delivery of the DNA message.

When a source node receives a DNA message indicating that a certain logical-to-physical address mapping is ineffective, it must find the proper entry in its translation table, and mark that mapping as ineffective. Henceforth, incoming packets with that particular logical address will not be sent to that particular physical address. If the logical address is multi-homed, packets will be sent to one or more of the other physical addresses, unless all the mappings for that logical address are ineffective. If this is the case, packets for that logical address will be discarded by the source node, which should also return some sort of negative acknowledgment to the source host. We see then that the DNA messages provide a feedback mechanism which enables a source node to tell when a mapping to a remote port is ineffective. The source node has no way to tell whether this is the case, until it sends a packet to that port. After sending the packet, it will be explicitly told by the DNA message if the mapping is ineffective. If it receives no DNA message, it assumes that the mapping is effective. This may mean, of course, that some logically addressed packets are sent to a wrong physical address. However, if there are other

possible physical addresses corresponding to that logical address, and the original destination node has one of those other mappings marked as effective, the packet will be re-addressed and re-delivered, so there is no data loss. Note that there are two possible reasons why a given logical-to-physical address mapping might be ineffective: (1) the physical port might not be operational, or (2) the host at that physical address might not have declared itself addressable with that logical address. If desired, the DNA message can indicate which of these two reasons is applicable in the particular case in hand. This information can be stored in the source node's translation table, and passed on to source hosts which try to use a logical address for which all the mappings are ineffective.

This procedure enables all nodes to find out when a particular authorized mapping is ineffective. We also need a procedure to enable the nodes to find out when an ineffective mapping becomes effective again (i.e., a port comes back up, or a new LAD message is received at some remote site). A simple but effective method is the following. At periodic intervals (say, every 5 or 10 minutes) each node will go through its translation table and mark all the entries which map to remote ports to be effective. (Entries which map to local ports will be marked effective or ineffective according to procedures already discussed. The current procedure will not apply to such

entries.) This enables mappings to be used again shortly after they become effective. Of course, this scheme will result in some packets being sent to the wrong physical address. When it happens, however, a DNA message will be elicited, causing that mapping to be marked ineffective again in that source node. Furthermore, this scheme does not cause any unnecessary data loss, since packets sent to the wrong physical address will be re-addressed and re-delivered, if possible.

Although this method requires all nodes to periodically mark all mappings to remote ports effective, it is important to understand that it does not require any time-synchronization among the various nodes. Also, there is no reason why all the mappings have to be marked effective at the same time. For example, if the translation table contains 600 mappings, rather than marking all of them effective every 10 minutes, it may be more efficient to mark one mapping effective each second, thereby cycling through the table every ten minutes (though if this method is used, it must take account of table compressions and expansions which may occur as the NCC adds or deletes authorizations).

There is also an issue as to the exact methodology to be used to send the DNA messages. The simplest method is for a destination node to send a DNA message to the source node of each packet which arrives as the result of an ineffective mapping. If

this method is used, there is no need to use a reliable transport protocol in sending the DNA messages. If, for some reason, a DNA message fails to get through to the source node, more packets will arrive at the wrong destination node, causing more DNA messages to be sent, until one of them finally gets through. This method, however, might generate a virtually unbounded number of DNA messages, particularly in pure datagram networks with no flow or congestion control. This in turn might contribute to network congestion. In order to gain better control of the throughput due to DNA messages, one could implement a scheme which ensures that only one DNA per ineffective mapping per source node can be sent within a certain time interval. This scheme would have a significant cost in table space, however. Also, it would require some sort of reliable transport protocol (e.g., positive acknowledgments from the source node when it receives the DNA message) to protect against the case where a DNA message is lost in transit. This issue would have to be carefully considered before any implementation is done.

The procedure to follow with virtual circuit traffic is very similar. In the ARPANET, a single virtual circuit or "connection" is individuated by the source and destination physical addresses. The user takes no part in setting up a connection; whenever a user sends a packet to the network which is not a datagram, the network checks to see if a connection from

the user's physical address to the destination physical address that he specified is already in existence. If not, the IMPs automatically run a protocol to set up such a connection. With logical addressing, we would want to redefine the notion of a connection so that connections are individuated by source and destination logical address, rather than physical address. However, translation would be done only at connection setup time. Thereafter, all virtual circuit packets received by a given source node with the same source logical address and destination logical address would be sent on the same connection. If a destination node receives a connection setup message for a logical address whose mapping is ineffective, it will refuse the connection, just as it would refuse a setup message for a physical connection to a dead port. When the source node receives the refusal message, it will mark that particular logical-to-physical mapping as ineffective. If the destination logical address is multi-homed, the source node can attempt to set up the connection again, but with a different physical destination address. If a mapping becomes ineffective after a connection has already been set up, the destination node will take action to reset the connection, also informing the source node that the mapping is now ineffective.

Note that logically addressed virtual circuit packets need not carry in their headers the logical addresses of either the

source or destination hosts, since that information can be stored in the connections tables at the source and destination nodes. Of course, all packets sent on a particular logical connection will go to the same physical destination port. However, if the destination node or port goes down, and the destination host is multi-homed, the above procedure automatically ensures that a new connection will be opened to one of the ports which is not down. Since the ARPANET connection protocol is transparent to the user, the user need never know that this has happened.

It is interesting to compare this procedure (based on DNA messages) with the previously discussed procedures (based on an updating protocol similar to that used for the SPF routing algorithm). The latter procedure would ensure that all nodes always agree (except during some very short transient period) on precisely which mappings are effective and which are not. Mappings would be marked effective (or ineffective) almost as soon as they become so. There would be no need for the source nodes to probe the destination nodes by sending data packets to possibly incorrect physical addresses. The procedure we are recommending does not have these features. In the recommended procedure, different nodes' translation tables would not necessarily be in agreement all the time as to which mappings are or are not effective, and probing is necessary. This is an acceptable situation though, since the sort of universal and

immediate agreement which is necessary to ensure the proper functioning of a routing algorithm just is not needed to ensure the proper functioning of the logical addressing scheme. However, the lack of universal agreement does require that translation be done at source nodes rather than tandem nodes, since the DNA-based procedure, while designed to keep the tables at source nodes up-to-date, will not necessarily have the same effect at tandem nodes. (That is, DNA messages are sent to the source nodes, not to tandem nodes.)

There is only one situation in which re-translation should be done at the tandem nodes. Suppose a logically addressed packet arrives at a tandem node, and that node, after checking its routing table, sees that the physical destination address of that packet is unreachable. If the packet is a virtual circuit packet, or the tandem node does not implement logical addressing (i.e., does not contain a translation table), the packet must simply be discarded. But if the packet is a datagram, and the tandem node does have a translation table, it should re-translate the destination logical address, and re-address the packet. This procedure can help to prevent unnecessary data loss. Note that this tandem node translation would happen only rarely, and only in situations in which it could not serve to defeat the criteria according to which the source node translation was done.

It should be noted that, with the recommended procedure (unlike the alternative), the size of the translation table at each node is a function only of the number of authorizations. That is, only changes in the authorizations require insertions or deletions to the table. This justifies our previous claim that insertions and deletions are relatively rare events.

It should also be pointed out that nothing in this procedure prevents a computer from being multi-homed to a single node.

2.6 Operational and Implementation Considerations

This procedure requires each network node to maintain a full table of authorized mappings. There are operational advantages to requiring all nodes to have precisely the same translation table; this simplifies the process whereby one node can be reloaded from another in case of failure, and reduces the amount of site-dependent information that must be maintained in the nodes. (In general, the more site-dependent information there is, the larger the Mean Time to Repair will be.) We have not spoken explicitly of the way in which NCC personnel add or delete authorizations. This will require some protocol between the nodes and the NCC. This protocol would be similar in some ways to the protocol used to broadcast software patches or packages to the nodes. However, since we want to be able to make incremental changes to the tables (rather than broadcasting an entire new

table each time a change must be made), the node will have to contain routines to add to or delete from the tables. The node may have to inhibit interrupts while modifying its table, so that no translations are done while the table is in a state of flux. Also, no reloads may be done from a node whose table is in a state of flux. These last two restrictions are needed to prevent race conditions; these restrictions are easily implemented.

When the NCC makes a change to the table of authorizations, it will want to receive some sort of positive feedback, indicating that the change has indeed been made. One method of doing this is to associate a sequence number with every "add" or "delete" command. Each node could periodically report to the NCC the sequence number of the last command that it fully executed, and an entry could appear in the log whenever the sequence number is other than expected. If the nodes refuse to execute commands which are received out of sequence, this would enable the NCC to determine whether each node has received the correct sequence of commands.

If memory considerations make it impossible for each node to contain a table of all authorized mappings, it is possible to get by with a shorter table. Strictly speaking, each node's table need contain only the logical addresses which map to that node itself, plus those logical addresses which the node's own local hosts are allowed to use. While this smaller table may take less

memory, it would, however, increase the operational difficulties of table maintenance. We have not yet said anything explicit about the format of a logical address. We recommend use of a 16-bit field for coding the logical addresses. This should be enough to prevent bit-coding limitations from placing any restriction on network growth. The only other information needed in the translation table is (1) destination node physical address (8 bits should suffice), (2) one bit for the effective/ineffective variable, (3) enough bits to code the port numbers, and (4) enough bits to code any variables needed to implement the selection criteria used for multi-homed hosts. This should not take more than two or three 16-bit words per entry. If space is a problem, it is possible to shorten the tables somewhat by deleting the port numbers. Strictly speaking, port numbers are only needed by the destination nodes, and hence need not appear in each node's copy of the translation table. However, eliminating the port numbers from the common table increases the amount of site-specific information in the tables, which is a disadvantage in itself.

It goes without saying that the use of logical addressing has implications for the network access protocol. We have already discussed one aspect of the network access protocol, viz., the need for the host to send LAD messages to the source node, and the need for positive and negative acknowledgments to

be returned to the host. A LAD message from a particular port contains a list of logical addresses, with an indication for each one as to whether the host wants the mapping of that logical address to that port to be effective or not. When a host declares a mapping to be ineffective, the source node must always return a positive acknowledgment, and must mark the mapping ineffective in its translation table. However, if the mapping is not authorized (i.e., not in the translation table), the source node should also return a warning to the source host, since host error is likely. When a host declares a mapping to be effective, the node will return either a positive or negative acknowledgment, depending upon whether the mapping is authorized or not. When a host declares an authorized mapping to be effective, the node must mark it so. The host should be allowed to send LAD messages to the node at any time, and they should take effect immediately.

2.7 Network Access Protocol Implications

The use of a logical addressing scheme also has implications on the part of the network access protocol that is used for ordinary data transport. When a source host passes a message to its source node, the message leader must indicate whether logical or physical addressing is desired (assuming the network allows both). If logical addressing is desired, the destination logical address must be indicated. The source node must be able to

discard that message (with an appropriate negative acknowledgment) if there is no effective mapping for that logical address. The source host must also be able to indicate its own logical address, if it wants to make this known to the destination host. (Since the source host may have several logical addresses, it must explicitly choose one to be carried to the destination host.) Again, the source node must be able to negatively acknowledge and then discard the message if the mapping from that logical address to the source host's physical address is not effective. Alternatively, one may want to return this sort of negative acknowledgment only if the source logical address mapping is unauthorized, and allow the message to be sent if the mapping is authorized but ineffective. If this is done, a particular "logical host" may be allowed to send data, but not to receive it.

When a logically addressed message is passed from the destination node to the destination host, the message header must contain the destination logical address (since the destination host may have more than one logical address), and the source logical address, if any. This implies, of course, that the source and destination logical addresses of datagram packets must be carried across the network in the packet header. (For virtual circuit packets, the logical addresses can be kept in the connection blocks in the source and destination nodes.) The

internal packet header must also carry the physical destination node number (for addressing at tandem nodes), and the physical source node number (so that DNA messages can be returned without having to rely on the integrity of the translation tables). However, these physical node numbers need not be passed to the destination host. Note that there is no need for the internal packet header to carry source or destination port numbers, since these are usually determined by the combination of physical node number and logical address. In the case where a host is multi-homed to a single node, the port numbers are not so determined, but the destination node can make a choice of ports "at the last minute," either by choosing according to one of the criteria already discussed, or by choosing the port with the shortest queue.

These considerations constitute a functional specification for a modified network access protocol that would allow logical addressing. Detailed specification of the modified protocols (SIP for AUTODIN II and 1822 for ARPANET) may depend heavily on implementation considerations, and will be put off until such time as implementation is considered.

REFERENCES

[1] E.C. Rosen, J.G. Herman, I. Richer, J.M. McQuillan, ARPANET Routing Algorithm Improvements -- Third Semiannual Technical Report, BBN Report No. 4088, April 1979, chapter 2.

[2] J.M. McQuillan, I. Richer, E.C. Rosen, D.P. Bertsekas, ARPANET Routing Algorithm Improvements -- Second Semiannual Report, BBN Report No. 3940, October 1978, chapter 4.

[3] E.C. Rosen, "The Updating Protocol of ARPANET's New Routing Algorithm," Computer Networks, February 1980, Volume 4, p. 11.

3. THE APPLICABILITY OF SPF ROUTING TO AUTODIN II

Determining the applicabilty of the ARPANET's routing algorithm (SPF) to AUTODIN II raises a number of issues, among them:

1) Can any single-path, delay-oriented routing algorithm meet the performance requirements of AUTODIN II?

2) Will SPF be as stable in the AUTODIN II environment as in the ARPANET, or will the higher connectivity and shorter average path length cause it to perform less well?

3) Will SPF use an excessive amount of trunk and/or nodal bandwidth in the AUTODIN II environment?

4) What impact will the multi-computer architecture of AUTODIN II have on SPF?

5) What impact will the presence of parallel trunks between pair of nodes have on SPF?

The first two issues will not be discussed in this chapter. They are best approached through simulation and will be considered in the second year of the contract when our network simulator is available. The last three issues, however, do not require simulation, and will be dealt with here.

We begin with a discussion of the amount of bandwidth needed for the SPF algorithm. In the ARPANET, the amount of table space (in 16-bit words) needed to hold the topological data base and the update retransmission timers used by the algorithm is given by the formula 4N + 2L + 0.125CN, where N is the number of nodes in the network, L is the number of (bidirectional) lines, and C is the average connectivity of (i.e., average number of lines per node in) the network. In the ARPANET, therefore, if 80 nodes and 100 lines are allowed, and the average connectivity is 2.5, the amount of memory needed for tables is 545 words. This same formula should be applicable to AUTODIN II. If a routing update packet generated by a particular node contains V bits of overhead and one word for each trunk emanating from that node, and if a given node can generate an update at most once every M seconds, and if there are N nodes with average connectivity C, the maximum amount of trunk bandwidth (in bits per second) needed to carry the routing updates is given by the formula N(V + 16C)/M. (There are N/M updates per second, and the average update size is V + 16C bits.) Note that because of the way the updating protocol works, every update traverses every network line, so an identical amount of bandwidth is needed on each trunk. In the ARPANET, V = 136 bits, C = 2.5 lines per node, M = 10 seconds, and N = 64 nodes, so the maximum bandwidth used is 1126.4 bps, or 2.25% of a 50 kbps line. Note though that this is the <u>maximum</u> bandwidth, not the <u>average</u> bandwidth. Although the maximum rate

at which an IMP can generate updates is once every 10 seconds, the average rate (as measured empirically) is about once every 40 seconds, which means that the average bandwidth is 281.6 bps or 0.56% of a 50 kbps line. This formula should also be applicable to AUTODIN II. In BBN Report No. 3803 (ARPANET Routing Algorithm Improvements, First Semiannual Technical Report), section 6.3, it is demonstrated that, as a general rule of thumb, the average amount of processing needed to respond to the delay update for a given line is 1.1H/C ms., where H is the average path length of the network and C is the average connectivity. Since each update contains an entry for C lines, on the average, the average amount of computation per update received and accepted is 1.1H ms. in the ARPANET. (Note that this does not include updates which are not accepted, because they are duplicates or are out-of-order. Such updates are discarded and the computation involved should be negligible.) While the figures of 1.1H ms. would not be expected to hold exactly in AUTODIN II, the proportionality to H should still hold. Since the AUTODIN II nodes are 10 years more modern than the ARPANET nodes, a given amount of computation should take much less time. On the other hand, the figure of 1.1H ms. applies only when the delays on all trunks are about equal. When congestion is present, the computation would be expected to take somewhat longer (see BBN Report No. 3803, pg. 130). The proportionality to H, however, is still a good rule of thumb.

Before these formulas can be applied to AUTODIN II, certain design decisions must be made. In AUTODIN II, each Packet Switching Node (PSN) can contain several independent computers. These computers are fully connected by means of a high speed bus (PCL), so that each can communicate directly with all the others. Some of the computers, the SCM's, will perform switching functions (i.e., will terminate internode trunks). Others, the TAC's, will perform terminal concentration functions. Each host or TAC will interface directly to a particular SCM. An important decision to be made is whether the basic unit of routing should be the PSN or the SCM. In AUTODIN II there will be only eight PSN's, but as many as twenty SCM's. (This latter figure is taken from Appendix K of Western Union's AUTODIN II Design (Technical) Specification.) If routing is done on a PSN basis, then, in effect, routing has to deal with only 8 nodes. The routing algorithm would not be aware of the internal structure of each PSN. If a PSN is composed of SCM's A and B, and host H accesses the network via SCM A, the routing algorithm would route traffic for H indiscriminately to either A or B; routing would have no way to distinguish between these two SCM's. This implies that the SCM's within a PSN must run some internal routing protocol, so that traffic which arrives at the "wrong" SCM can be shuttled over the PCL to the correct SCM. Clearly, more optimal routing is possible if routing is done on an SCM basis, but in this case the routing algorithm must deal with 20 nodes instead of 8.

Since the amount of bandwidth needed for the SPF algorithm is sensitive to the number of nodes, it is important to decide whether routing should be done on an SCM basis or a PSN basis.

Another design decision which must be made before our formulas can be directly applied to AUTODIN II has to do with the presence of parallel lines. A given pair of PSN's may be connected by several trunks running in parallel. Whenever several trunks running in parallel have comparable delay characteristics, best performance (i.e., least delay and greatest throughput) is obtained by having them serve a single queue. (We will consider the case of heterogeneous parallel trunks later in this section.) If the parallel trunks are to be run on a "single queue, multiple server" basis, the routing algorithm need not distinguish among them. Rather, routing should treat the set of parallel trunks as a single trunk. This means that if routing is done on a PSN basis, the network will have 8 nodes and 16 trunks, with an average connectivity of 4 and an average path length of 1.43. (See Figure 3-1.)

If routing is done on an SCM basis, however, determining the average connectivity and average path length is somewhat more complicated. When routing is done on a PSN basis, the routing algorithm need not take account at all of the PCL's which fully interconnect the SCM's within a PSN. In order to do routing on an SCM basis, however, the routing algorithm must model each PCL

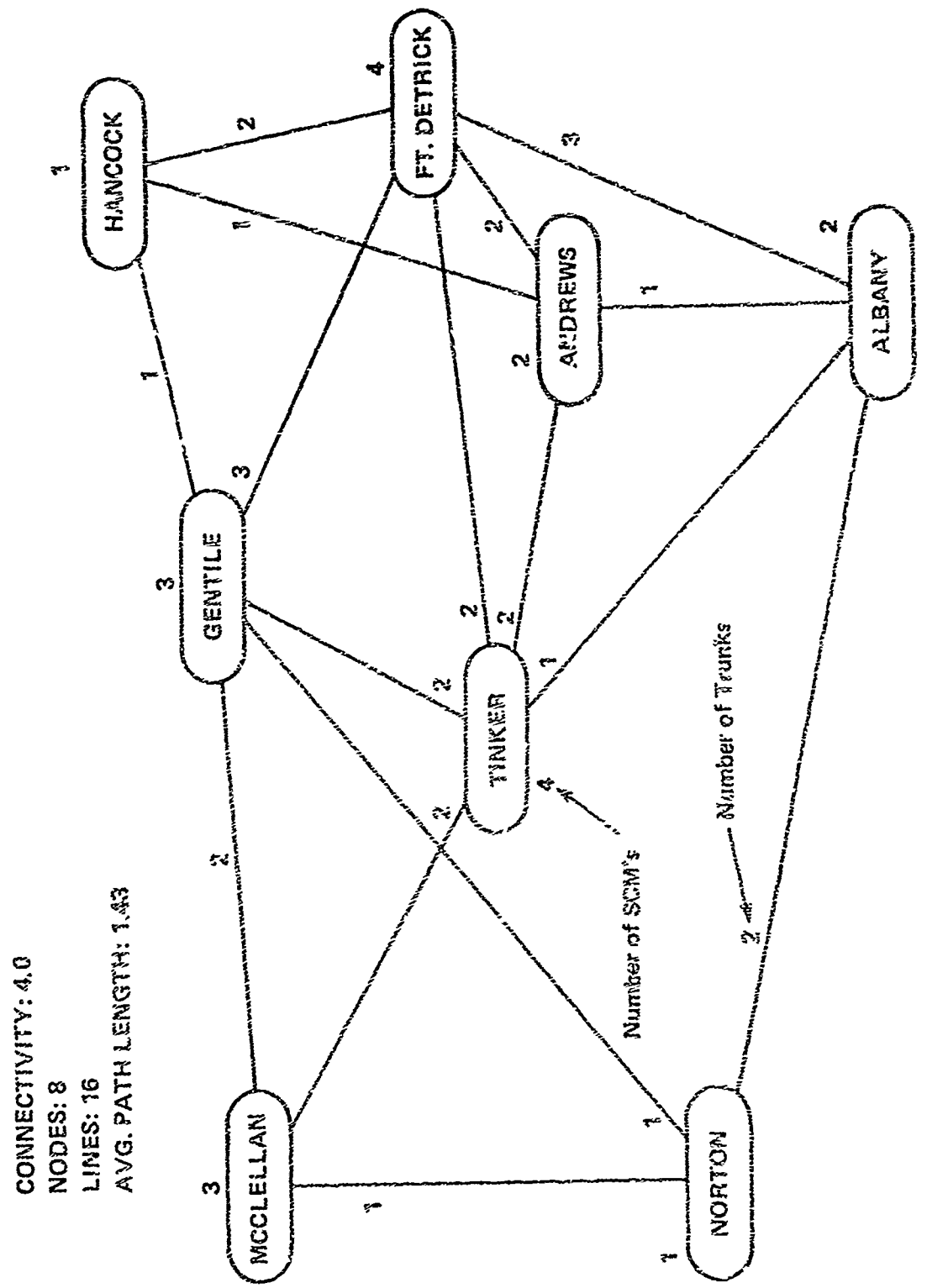
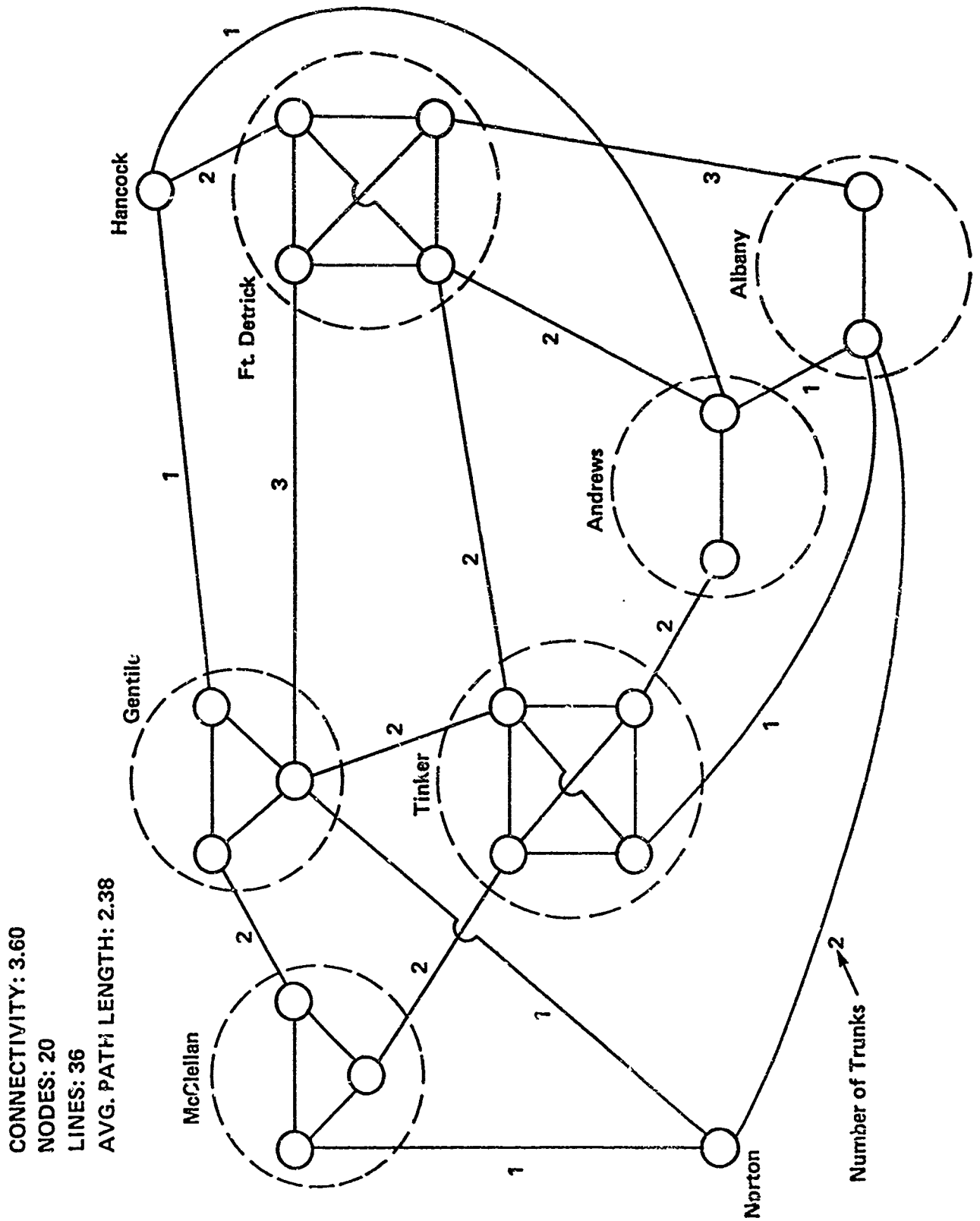

HANCOCK
FT. DETRICK
ANDREWS
ALBANY
GENTILE
TINKER
MCCLELLAN
NORTON
CONNECTIVITY: 4.0
NODES: 8
LINES: 16
AVG. PATH LENGTH: 1.43
Number of SCM's
Number of Trunks

Figure 3-1

as a set of trunks. For example, if there are 4 SCM's in some PSN, then the routing algorithm must treat the PCL in that PSN as if it were 6 bidirectional trunks, one between each pair of SCM's. Another complication has to do with the fact that when several trunks run in parallel between a pair of PSN's, it does not follow that they run in parallel between a pair of SCM's. Let P1 be a PSN containing SCM's A1 and B1. Let P2 be PSN containing SCM's A2 and B2. If P1 and P2 are connected by a pair of parallel trunks, it may mean that A1 and A2 are connected by that pair of trunks, or it may mean that A1 and A2 are connected by a single trunk, as are B1 and B2. The former situation tends to result in a lower connectivity but a higher average path length than the latter, and these factors affect the amount of bandwidth needed to perform SPF routing. Unfortunately, the trunking of AUTODIN II at the SCM level has not yet been determined. Therefore, we will carry out the computation for SCM-based routing in two ways. First, we will consider a network where the trunks are arranged so as to maximize the parallelism. (See Figure 3-2.) That is, in the network, whenever several trunks run in parallel between a pair of PSN's, we will assume that they also run in parallel between a pair of SCM's. This network has 20 nodes, 36 lines, an average connectivity of 3.60, and an average path length of 2.38. Second, we will consider a network where no two trunks run between the same pair of SCM's. (See Figure 3-3.) This network has 20 nodes, 48 lines, an


Hancock
Ft. Detrick
Albany
Andrews
Gentile
Tinker
McClellan
Norton
Number of Trunks
CONNECTIVITY: 3.60
NODES: 20
LINES: 36
AVG. PATH LENGTH: 2.38

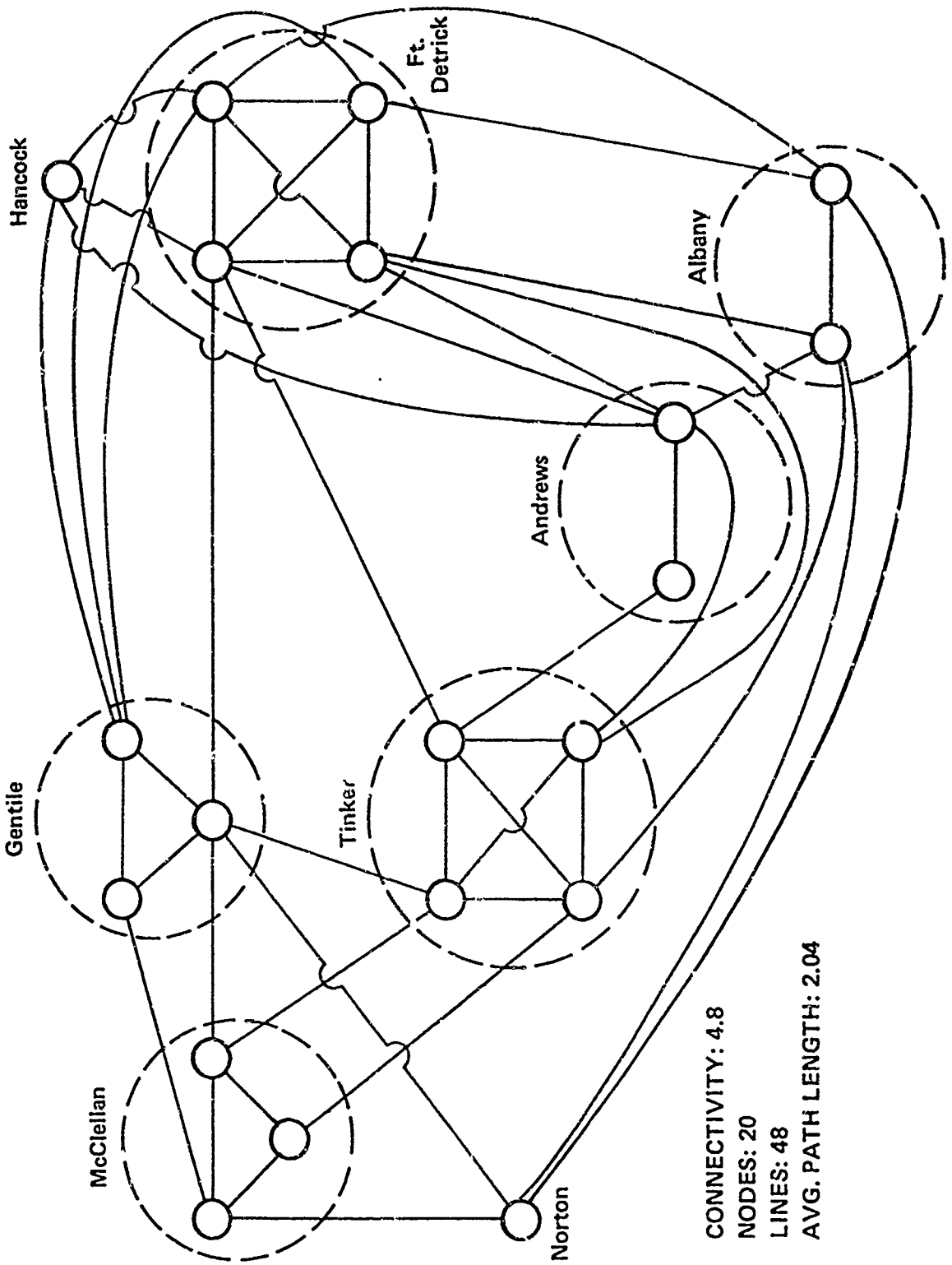

Hancock
Ft. Detrick
Albany
Andrews
Gentile
Tinker
McClellan
Norton
CONNECTIVITY: 4.8
NODES: 20
LINES: 48
AVG. PATH LENGTH: 2.04

Figure 3-3

average connectivity of 4.8, and an average path length of 2.04.

We can now compute the bandwidth requirements for each of the three possible configurations of AUTODIN II:

1) Table Space: $4N + 2L + 0.125CN$ words

   a) PSN-based routing: N = 8, L = 16, C = 4.0

      Total memory used for tables = 68 words

   b) SCM routing, maximal parallelism: N = 20, L = 36, C = 3.60

      Total memory used for tables = 161 words

   c) SCM routing, no parallelism: N = 20, L = 48, C = 4.8

      Total memory used for tables = 188 words

2) Trunk bandwidth: $N(V + 16C)/M$ bps. We assume that as in the ARPANET, V = 136 bits and M = 10 seconds.

   a) PSN routing: N = 8, C = 4.0

      Total trunk bandwidth = 160 bps (0.29% of a 56 kbps trunk)

   b) SCM routing, maximal parallelism: N = 20, C = 3.60

      Total trunk bandwidth = 387 bps. (0.69% of a 56 kbps trunk)

   c) SCM routing, no parallelism: N = 20, C = 4.8

Total trunk bandwidth = 426 bps (0.76% of a 56 kbps trunk)

3) Processing bandwidth: 1.1HN/M ms./sec.

a) PSN routing: H = 1.43, N = 8, M = 10 secs.

Processing bandwidth = 1.26 ms/sec. or 0.13%

b) SCM routing, maximal parallelism: H = 2.38, N = 20, M = 10

Processing bandwidth = 5.24 ms/sec., or 0.52%

c) SCM routing, no parallelism: H = 2.04, N = 20, M = 10

Processing bandwidth = 4.49 ms/sec., or 0.45%

We see then that the bandwidth requirements of SPF routing in the AUTODIN II environment are quite modest, even if routing is done on an SCM basis. Of course, there are other possible topological layouts of AUTODIN II, for which the figures will be slightly different. Even if our figures are off by as much as an order of magnitude, however, SCM-based SPF routing still seems acceptable. (It should also be pointed out that PSN-based routing requires a second routing algorithm to be run internally in the PSN's, and this will also use some nodal bandwidth. When this is taken into account, it is entirely possible that PSN-based routing will be more expensive, in terms of nodal bandwidth, than SCM-based routing.)

Since the choice between SCM-based SPF routing and PSN-based SPF routing is not determined by bandwidth considerations, the choice between these two routing methodologies can be based on considerations of optimality. That is, which approach will result in better network performance? With either approach, it will sometimes be necessary for packets to be shuttled within a PSN from one SCM to another over the PCL. With SCM-based routing, the routing algorithm can take account of the cost of the shuttling, treating it as an extra hop, no different in principle from a hop between PSN's over an internode trunk. With PSN-based routing, however, this shuttling is made transparent to the routing algorithm. The use of PSN-based routing, therefore, is essentially equivalent to assuming that the shuttling has zero cost. At first glance, this assumption might seem appropriate. The bandwidth of the PCL is about 8 megabits per second, and the PCL itself is only a few feet long, so neither the transmission delay nor the propagation delay imposed by the PCL seem significant. This reasoning, however, is somewhat simplistic. According to AUTODIN II Western Union Technical Note 77-03, the polling discipline of the PCL is such that the queuing delay seen by packets which must traverse a particular PCL is directly proportional to the number of processors connected to that PCL. Furthermore, that note claims that as the number of processors in the PSN gets to around 10, the queuing delays are roughly of the same order as the queuing delays over ordinary internode trunks.

This means that it is important for the routing algorithm to take account of the PCL, and to treat shuttling within a PSN as an extra hop. Even if the delay imposed by the PCL were not significant, however, it would still be important for routing to determine exactly which SCM's a packet must go through. Each SCM that a packet passes through adds a certain amount of processing delay. This can become significant under heavy load. For best performance, the routing algorithm should have the option of routing packets around SCM's which are especially heavily loaded. This ability is lost if routing is done on a PSN basis. It is also the case that, within a PSN, some SCM's may have fewer available buffers than others. If some SCM's are congested while others are not, routing should try to avoid the use of the congested ones. This can only be accomplished if routing is done in an SCM basis. In fact, if routing is done on a PSN basis, the routing algorithm has no way to control the amount of shuttling of packets within a PSN. Since this shuttling does have a delay cost, especially when the SCM's within a PSN are not uniformly loaded (and this appears to be a very likely case), PSN-based routing is clearly inferior. PSN-based routing is also inferior with respect to throughput. The maximum effective throughput of a PSN is inversely related to the amount of shuttling that must be done from one SCM to another. If the routing algorithm can have no effect upon the amount of shuttling (as in PSN-based routing), we would expect a lower maximum throughput.

In fact, it may actually be difficult or even impossible to do any sort of accurate routing on a PSN basis. The reason is that it is difficult to perform an accurate delay measurement if the routing algorithm cannot take account of individual SCM's. For the reasons noted above, the delay seen by a packet which must leave a PSN by a certain trunk may vary greatly, depending on whether the packet enters the PSN at the SCM connected to that trunk, or whether it enters at a different SCM. Yet PSN-based routing requires that only one value of delay be assigned to the trunk. Presumably, this would have to be some sort of weighted average of the delay experienced by packets arriving at the various SCM's. However, since any given packet arrives at some particular SCM, the weighted average may not be a good predictor of the delay for any packet. Therefore, PSN-based routing necessarily distorts the delay measurements, thereby causing further sub-optimalities in the routing.

Another criterion which favors SCM-based routing over PSN-based routing is robustness. It is possible that failures of the PCL, or the components connecting the individual processors to the PCL, will cause a PSN to become partitioned, in the sense that not all of the SCM's within the PSN will be able to communicate with each other. If PSN-based routing is used, this situation must result in at least some of the SCM's being declared down, so that none of their trunks will be available for

use. With SCM-based routing, on the other hand, all the trunks and processors can stay on-line, and only the failed components themselves will be unavailable. To see this, consider the following example. Suppose a PSN is composed of two SCM's, A and B. SCM A has two internode trunks, L1 and L2, and SCM B also has two internode trunks, L3, and L4. If PSN-based routing is used, the PSN appears as a single node with four lines. That is, the PSN appears to have four immediate neighbors. Now consider what would happen if SCM's A and B could no longer communicate with each other over the PCL. Traffic arriving over L1 or L2 could no longer be routed out L3 or L4, and traffic arriving over L3 or L4 could not be routed out L1 or L2. The routing algorithm, however, would have no way to detect (or even represent) this fact if PSN-based routing were used. Even though all four lines would still be up, it would be wrong to represent the PSN as a single node with four immediate neighbors, since packets arriving over the trunk from one neighbor could be sent directly to only one of the other three. Yet the routing algorithm would not be able to tell which one of the three that is. It would also not be correct to represent the PSN as having only two immediate neighbors. The way in which a packet entered the PSN would determine the way in which it could exit the PSN, and this situation could not be captured by representing the PSN as a single node with a fixed set of neighbors. The only way for PSN-based routing to handle this situation is either to treat L1

and L2 as down, or to treat L3 and L4 as down. This is equivalent to bringing down one of the two SCM's. In general, if PSN-based routing is used, and a PSN becomes partitioned, all but one of the SCM's must be brought down. On the other hand, if SCM-based routing is used, all SCM's and lines may remain up. We conclude that in certain failure conditions, SCM-based routing makes more network bandwidth available than does PSN-based routing, and hence is more robust.

Another advantage of SCM-based routing is that it allows more optimal assignment of network trunking than does PSN-based routing. AUTODIN II trunking has been chosen to optimize network performance for a particular external traffic requirement. Traffic from a particular external source enters the network at a particular SCM; traffic to a particular external destination leaves the network at a particular SCM. In order to optimize the trunking, therefore, it is necessary to ensure that particular trunks are terminated at particular SCM's. If the trunking is assigned to terminate only at particular PSN's, with no regard for the SCM's which actually terminate the trunks, then the trunking cannot possibly yield optimal performance. Note, however, that if PSN-based routing is used, the optimal assignment of trunks to SCM's will be <u>defeated</u> by the routing algorithm. Since the routing algorithm will not be able to distinguish trunks by the SCM's which terminate them, it will

never be able to generate the traffic flows needed to best utilize the trunking. In fact, if several lines are running "in parallel" between a pair of PSN's, but the lines have different SCM terminal points, the routing algorithm will have no control at all over which packets traverse which of the trunks. Essentially, this means that the choice among those trunks is made at random. This sort of random choice would be expected to produce an excessive amount of shuttling of packets across the PCL, thereby increasing delays, reducing throughput, and generally obviating the trunk assignment and sizing decisions.

The decision to perform routing on an SCM basis raises the issues of how the intranode PCL is to be treated by the updating protocol and by the delay measurement routines. In general, if there are M SCM's in a PSN, each SCM should treat the PCL as if it were M-1 trunks, one to each of the other SCM's. Whenever an SCM receives a routing update which it is supposed to forward to its neighbors, it should send M copies over the PCL, one to each of the other M SCM's. (Of course, it will also send one copy out each of the real internode trunks.) Presumably, an SCM receiving a routing update over the PCL will know who it is from, and can send the acknowledgment (echo) back to the correct SCM. Note, however, that the updating protocol causes every update to be transmitted over every network line. If each of M SCM's sends M-1 copies of each update over the PCL, the PCL will have to

carry M(M-1) copies of each update. We have already computed that the maximum average bandwidth needed by routing updates in the ordinary internode trunks, where each update is transmitted only once, is 426 bps. The bandwidth on the PCL, therefore, if we assume 4 SCM's in the PSN, is 4*3*426 or 5112 bps. This is 0.06% of the bandwidth of the PCL, which seems to be an acceptable figure. If it is possible for a single packet on the PCL to be received by several SCM's (i.e., for packets on the PCL to be multiply-addressed), then only M copies of each update need be sent on the PCL, resulting in a significant reduction in bandwidth. Of course, these considerations are based on the assumption that the same protocol would be used for transmitting routing updates over the PCL as is used for transmitting them over the ordinary trunks. It is possible that a different sort of protocol, which is more efficient, might be devised. Devising such a protocol would require a more detailed knowledge of the PCL hardware and software than is publicly available at present. In the absence of such knowledge, there seems to be no objection to using the same protocol as is used on the trunks.

We presume that all the SCM's within a PSN will participate in a protocol which enables each SCM to know which other SCM's are present in the node. If we regard the PCL as a set of trunks which fully interconnect the SCM's, this protocol will be the equivalent of a line up/down protocol. If an SCM goes down, the

other SCM's in the PSN will presumably detect that fact, and will act as if their trunk to the failed SCM has gone down. That is, they will generate routing updates indicating infinite delay on their line to the failed SCM.

Measuring the delay of packets which are transmitted over the PCL is no different than measuring the delay of packets transmitted over a trunk, except that transmission and propagation delays can probably be neglected in the case of the PCL. There is, however, the following issue. Suppose SCM A communicates with SCM's B and C over a PCL. Presumably, there will be a single queue in A which contains packets destined for B and C. Since there is only a single queue for the PCL, it may seem as if the average delay on the "trunk" from A to B should always be identical to the average delay on the trunk from A to C. However, this is not the case. It must be remembered that an SCM may refuse any packet if it lacks the resources (e.g., buffers) needed to process that packet. This applies to packets received over the PCL as well as to packets received over the ordinary trunks. When packets arriving over a trunk are refused, they will be retransmitted shortly by the transmitting SCM; and presumably, this will also be true of packets arriving over the PCL. The delay of a packet, as measured by the delay measurement routines of the routing algorithm, is the length of the time interval that begins when the packet enters the SCM and ends when

it has been transmitted for the final time to its next SCM. If SCM B is shorter on buffers than SCM C, so that packets which A sends to B have to be retransmitted several times, while packets sent from A to C need only be transmitted once, then the average delay on the "trunk" from A to B should be much larger than the average delay on the "trunk" from A to C. That is, the delay must be measured and reported separately for each "trunk."

The presence of the intranode PCL also raises an issue with respect to the quantization of the delay measurement. If delays are to be represented in a finite number of bits, there must be some unit of quantization. The choice of this unit depends both on the number of bits available for representing the delay and on the dynamic range of the delay values. If the range of values of delay for the PCL is very different from the range of values of delay for the ordinary trunks, it may be difficult to choose a quantization unit which is appropriate for both cases. A related issue has to do with the change-of-delay thresholds that are used to determine when a measured change of delay is significant enough to warrant creation of a routing update. It is possible that different thresholds should be used for the PCL than for the ordinary trunks. It is also possible that the length of the delay measurement interval should be different for the PCL than for the ordinary trunks. While a determination of the applicability of SPF routing to AUTODIN II does not depend on a

resolution of these issues, these issues would have to be carefully and empirically studied before SPF routing could be implemented in AUTODIN II.

We now discuss some of the issues which are raised by the presence of parallel trunks, i.e., trunks which connect the same pair of SCM's. It is not yet clear whether AUTODIN II will have such trunks, but AUTODIN II's routing algorithm ought to be able to handle them. We will assume for the moment that any parallel trunks are "homogeneous," in the sense of having very similar transmission and propagation delays. In this case, the routing algorithm ought not to distinguish the components of the composite trunk, but rather should treat it as a single trunk, with a single value of delay associated with it. (This is a consequence of the fact that the best delay and throughput values are obtained from a composite trunk by having the components serve a single common queue.) Since each component trunk will have its own independent error characteristics, the line up/down protocol must be run independently over each component. The composite trunk will be considered to be up whenever any of its components are up; it will be considered down whenever all of its components are down. The delay measurement can be done just as if there were only a single trunk. The average delay per packet for the composite trunk will then be the weighted average over the component trunks.

It is possible that the error rate on one of the component trunks is so great that better performance of the composite trunk is obtained if that component is not used at all. In this case, the line up/down protocol should bring that component down; there is no need for the routing algorithm to take account of that explicitly. If the error rate on one of the components is not bad enough to bring it down, but is still significantly greater than the error rates of the other components, the purely local procedure of having all components serve a single queue will ensure that more packets are transmitted by the higher quality trunks than by the lower quality trunks, as long as there is a limit on the number of unacknowledged packets which may be in flight at once over a single component. (Even though routing will not distinguish the components, each component needs to run its own link protocol, ADCCP, independently of the other components.) It is conceivable that the error rate on one of the components will be so bad that that component ought not to be used unless all other components have their maximum number of packets in flight. Even in this case, however, routing ought not to distinguish the components of the composite trunk. The decision as to which component to use can be made much more accurately and effectively by the transmitting node, as it removes packets from the queue, than by the routing algorithm.

Running the updating protocol over the composite trunks offers no particular problem. Routing updates and acknowledgments can be sent over whichever component is available first; an update and its acknowledgment need not travel on the same trunk.

If heterogeneous parallel lines are used, there are several ways to handle them. The way which is actually chosen will depend on the reason why the lines were installed in parallel. Suppose, for example, that two SCM's are connected by both a terrestrial trunk and a satellite link. It may be that the satellite link is intended for use only as a backup when the terrestrial link is down. In this case, the line up/down procedures should ensure that only one of the two trunks is up at any one time. The delay measurement routines and the updating protocol will operate on whatever trunk happens to be up at a given time; the routing algorithm need not be explicitly aware of the fact that there are actually two trunks, rather than one.

Another way of using a satellite link and a terrestrial link in parallel is to send traffic over the satellite link only if the terrestrial link is so heavily loaded that the average packet delay over the terrestrial link is about equal to the satellite propagation delay. The decision as to whether to send a particular packet over the satellite link or the terrestrial link could be made locally by the transmitting node. Again, the

routing algorithm would not have to distinguish the trunks. The delay reported for the composite trunk would just be the weighted average of the delays for the two component trunks. Of course, routing updates should be transmitted over the terrestrial trunk.

Another possible way to use a satellite link and a terrestrial link in parallel would be to use the satellite link for those traffic flows requiring high throughput, while using the terrestrial link for those traffic flows requiring low delay. This scheme would require each node to run the SPF algorithm twice, using a different topological data base each time. One of the data bases would contain only those lines which can be used for interactive traffic; the other would contain only those lines which can be used for bulk traffic. If a terrestrial line and a satellite line are running in parallel, the first data base would contain an entry only for the terrestrial line, and the second data base only for the satellite line. As a result, the SPF algorithm would create two routing trees, one for each kind of traffic. Since the two trees might not be consistent, each intermediate node would have to determine whether a given packet is bulk or interactive (the packets are already marked to indicate this) and forward the packet along the corresponding routing tree. A satellite line and its parallel terrestrial line would serve different queues in this case. Note that, although bulk traffic will not travel on links which are intended for

interactive traffic only, the SPF algorithm will still try to minimize the delay (rather than maximize the throughput) of the bulk traffic. That is, the routing algorithm will still tend to avoid the satellite lines because of their long propagation delays. If this is not appropriate, the delay measurement routines can assign the satellite lines a minimal value of propagation delay, instead of their true value. This would cause more of the bulk traffic to be routed over the satellite lines. In fact, if the satellite lines are of higher bandwidth than the terrestrial lines, they will produce shorter transmission and queuing delays than the terrestrial lines, so ignoring their propagation delays will tend to make them look better than the terrestrial lines, causing more of the bulk traffic to be routed to the satellite lines.

With this scheme, when a node generates a routing update, it would contain information on all its lines, including both the lines to be used for interactive traffic and the lines to be used for bulk traffic. However, the entry for each line in the update would have to indicate whether the line is to be used only for bulk traffic, only for interactive traffic, or for both kinds of traffic. In this way, each node receiving the update would know which of its topological data bases requires updating.

4. A MULTIPLE PATH ROUTING ALGORITHM BASED ON SPF

4.1 The Objective Function -- Maximizing Throughput

The purpose of a routing algorithm for a computer network is to route the traffic which is in the network at a given time on the set of paths which are most "suitable," given the state of the network at that time. A set of paths is considered most suitable if the use of that set of paths results in the optimization of the value of some "objective function." The objective function which is to be optimized must be chosen by the network designers, and may be different for different network applications. The proper choice of an objective function is part of the design of the network, and depends on the goals and requirements of the design. For example, a commonly cited objective function is average packet delay, and discussion of routing algorithms often proceeds on the assumption that a routing algorithm should choose the set of paths which result in the smallest average packet delay. The objective function which the ARPANET's new routing algorithm attempts to minimize is something we may call "individual packet delay." That is, the ARPANET's routing algorithm attempts to seek the path of least delay for each individual packet. (Minimizing the average packet delay may result in some packets being given very long delays so that others can have very short delays. Minimizing the individual packet delays may result in a higher average delay,

but will prevent the routing algorithm from trading off some very short delays for some very long delays, and hence is more appropriate for the ARPANET environment.) Most discussions of routing algorithms (particularly distributed adaptive routing algorithms) have tended to focus on one of these two objective functions.

Another sort of objective function which is of importance is throughput. The work reported herein is an attempt to develop a routing algorithm that will maximize the amount of throughput that a network can carry. This means that the routing algorithm should distribute the flows in such a way as to minimize the number of hops each flow must take in proceeding from the source node to the destination node. The ability of the network to carry traffic is constrained by the amount of resources, such as link bandwidth, buffer space, and CPU cycles, that the network has available. As more of these resources are used by any particular source-destination flow, fewer are left for use by other flows. Yet every time a packet passes through another link or node, it uses resources in that link or node, so a packet which travels 10 hops uses 10 times the resources of a packet which travels one hop. It follows that the amount of throughput that can be carried by the network is maximized when the number of packet-hops in the network is minimized. (The number of packet-hops is just the sum, over all packets in the network, of

the number of hops each packet must travel to get from its source node to its destination node.)

It is sometimes thought that throughput can be maximized by a routing algorithm that tries to route each packet on the path which has the maximum unused capacity. After all, to minimize network delay, one sends traffic on the path which shows the least delay. Analogously, one might reason, to maximize network throughput, one should send traffic on the path which has the maximum capacity. Such reasoning, however, is entirely specious. Other things being equal, throughput is maximized by routing packets on the min-hop paths. This can be seen by the following example. Suppose there are two paths from a source S to a destination D. One path consists of a single hop, on which there is an unused capacity of 50 units. The second path consists of 10 hops, each of which has an unused capacity of 100 units (resulting in an unused capacity of 100 units for the entire path). Suppose also that we are trying to pass 40 units of flow from S to D. If sent on the min-hop path, the flow uses only 40 units of network capacity. If sent on the 10-hop path, it uses 40 units of network capacity at each hop, for a total of 400 units. Since the network has finite capacity, using 400 units of capacity to handle a flow which could be handled by using only 40 units of capacity will reduce the total amount of throughput the network can carry.

Another way to see this point is the following. If the 10-hop path is used to carry a single flow, its total effective capacity is only 100 units. If, on the other hand, it is used to carry 10 1-hop flows, its total effective capacity is 1000 units. Given a choice of using a fixed amount of resources to carry either 100 units of flow or 1000 units of flow, we should choose the latter if our goal is to maximize throughput. Again, this implies that maximizing throughput is the same as minimizing the number of packet-hops.

However, this does not imply that simple min-hop routing is sufficient to maximize the throughput that the network can carry. Some particular source-destination flow may require more capacity than can be found on the min-hop path, or some combination of flows for which the min-hop paths are not disjoint may require more capacity than can be found on the common link or node of their min-hop paths. So in general, maximizing throughput requires the simultaneous use of multiple paths between each source-destination pair. The considerations involved in choosing the proper set of multiple paths are discussed in the next section.

4.2 Choosing a Set of Multiple Paths to Maximize Throughput

Packets from a particular source-destination traffic flow ought to be routed over several paths simultaneously only if there is insufficient capacity on the shortest (min-hop) path to carry all of the flow. In order to determine a method of choosing these additional paths, we must first determine what characteristics we would like the paths to have. For the sake of concreteness, let us assume that we need two paths to carry all the traffic of some particular source-destination flow. We know that the first path will be a min-hop path. What characteristics should the second path have? Since we are trying to minimize the number of packet hops, one might think at first that the second path should just be the second-shortest path (where the length of a path is just the number of hops in it). However, this is not the case, in general. It is possible that the shortest path and the second-shortest path will have some links in common. In particular, the traffic which is being routed over the shortest path may be causing some link to be fully utilized. If this fully utilized link is also a link of the second-shortest path, then the second-shortest path provides no additional capacity. That is, the use of both paths simultaneously will provide no more capacity than will the use of the first path alone.

The problem in this scenario is that the two paths have a common bottleneck (i.e., a node or link with no excess capacity).

If we want to choose a second path which provides additional capacity for the flow, then we must ensure that the two paths are "bottleneck-disjoint," i.e., that they have no common bottleneck. Since we are also interested in minimizing the number of packet-hops, we should choose as the second path the shortest path which is bottleneck-disjoint with the first path. We may generalize this as follows. A traffic flow from source S to destination D should be routed over the set of paths $p_1,\ldots,p_k$ such that:

a) p_1 is the shortest path which is not fully utilized by other flows

b) for all i, $1<i\leq k$, p_{i+1} is the shortest path which is bottleneck-disjoint from all the p_j, $j\leq i$

c) k is the smallest number of paths which together provide enough capacity to handle the flow.

Choosing the paths according to these constraints will tend to minimize the number of packet-hops while providing as many paths as are needed to handle the offered traffic, i.e., to maximize throughput.

The SPF algorithm, in order to produce the shortest-path tree of a network, requires as input a representation of the network, specifying for each link (where a link is considered as

a unidirectional entity) the nodes which are the endpoints of the link and the "length" of the link (which will be a constant 1 if the appropriate notion of path length is number of hops). A natural way to find the shortest path which is bottleneck-disjoint from a given path is the following. Find the bottlenecks of the first path, and "erase" them from the representation of the network. Then run the SPF algorithm on the new reduced representation of the network. The result will be a shortest-path tree containing the shortest paths which are bottleneck-disjoint from the given path.

This is the basis for our proposed SPF multi-path routing algorithm. Each link will be assigned a fixed length of 1. If p_i is a path between source S and destination D, the next path p_{i+1} will be chosen by "erasing" from the representation of the network the links which are the bottlenecks of p_i, and then running the SPF algorithm on this reduced network.

The algorithm we propose will be a distributed one (as in the ARPANET), in that every node will compute a complete set of paths from itself to all other nodes. This means that every node will have to know which links in the network are bottlenecks. A link may be defined as a bottleneck if its utilization exceeds a certain threshold. The utilization of a link can be directly measured only by the node from which the link emanates. So each node will have to measure the utilization of all its outgoing

links and broadcast this information to all the other nodes. That is, the routing updates of this multiple path SPF-based algorithm will report link utilizations, rather than (as in the ARPANET) link delays. Note that the values reported in the routing updates are not the link lengths which are used for computing the shortest-path tree; those lengths will be fixed at 1. Rather, the values reported in the updates are used to determine which nodes are bottlenecks, so that paths which are bottleneck disjoint can be chosen.

In order to know when a set of k paths jointly have enough capacity to handle a certain flow, it is necessary to know how much capacity each path is providing. Again, this requires that each node know the utilization of each link in the network. With this information, the SPF algorithm is easily modified so that it produces, for each path to a destination, the capacity which that path provides to that destination.

It is worth noting that, in general, the set of multiple paths from a given source to a given destination need not be fully disjoint. This can be seen in Figure 4-1. If only the shortest path from S to D is used (SEFD), then only 10 units of traffic can be sent. If the non-disjoint paths SEFD and SEGFD are used, 50 units can be sent. But if the fully disjoint paths SEFD and SHIJD are used, only 15 units can be sent. This example just re-emphasizes the fact that what we are interested in is bottleneck-disjointness, not full disjointness.

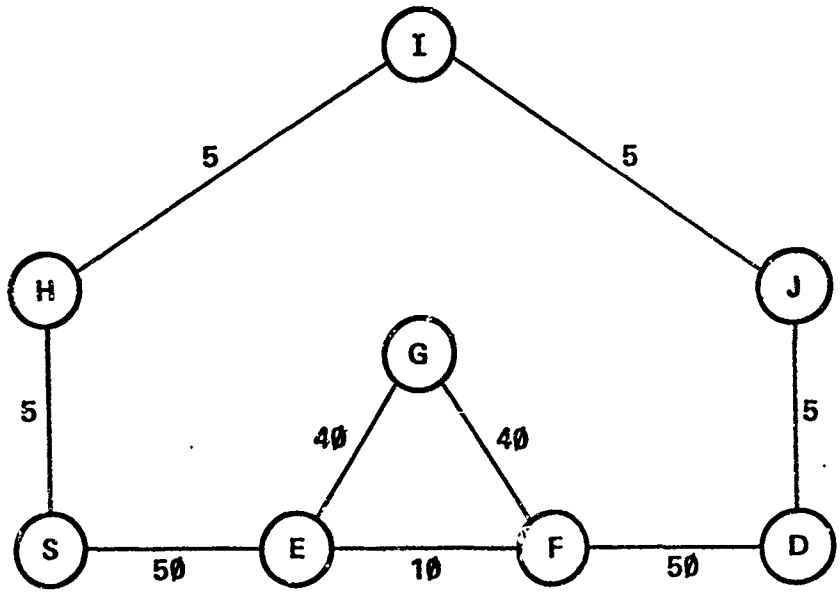


Figure 4-1

4.3 On the Notion of a Bottleneck

We have not yet discussed the notion of a "bottleneck" in any detail. Clearly, this notion must be precisely defined before it can be used as part of a multi-path algorithm, and the way in which it is defined will have an important effect on the behavior of the routing algorithm.

Note that there will not be a k^{th} path from S to D ($p_k(S,D)$) unless the previous path $p_{k-1}(S,D)$ has a bottleneck link. For if $p_{k-1}(S,D)$ has no bottlenecks, then "the shortest path which is bottleneck disjoint from $p_{k-1}(S,D)$" will be $p_{k-1}(S,D)$ itself. Furthermore, since we want to minimize the number of packet-hops,

we do not want to use a k^{th} path unless the first k-1 paths do not provide sufficient capacity to handle the offered flow from S to D. It follows that we do not want to begin using a k^{th} path until the first k-1 paths are "fully loaded." This suggests that we define a bottleneck as a link which is fully loaded. Of course, we have some leeway in deciding just what we mean by "fully loaded." A link is fully loaded if its utilization is as high as we would ever like it to be. But this does not necessarily mean that it is 100% utilized. We may decide, for example, that to avoid the long queuing delays that may be associated with links that are 100% utilized, we want to utilize the links only to the 80% level. Then a link which is 80% utilized would be defined to be a bottleneck. This threshold value of utilization might also be different for different links, depending perhaps on the speed of the link or the reliability of the link, or any other factor deemed relevant. We will not further discuss the issue of choosing the proper threshold value.

Clearly, a link which is a bottleneck at some time may not be a bottleneck at some later time. One purpose of having an adaptive routing algorithm is to be able to tell dynamically, in real time, whether a given link is a bottleneck or not. Suppose path $p_1(S,D)$ includes link l_1, which is the only bottleneck on the path. Then $p_2(S,D)$ will be the shortest path from S to D which does not include link l_1. Now suppose that at some later

time, link l_2, which is on both $p_1(S,D)$ and $p_2(S,D)$, becomes a bottleneck. Then a new $p_2(S,D)$ must be defined which will be the shortest path from S to D which does not include either l_1 or l_2. However, this is not precisely what we desire. It may be that, in order to fully utilize l_2, it is necessary for it to carry traffic on <u>both</u> $p_1(S,D)$ and $p_2(S,D)$. Yet as soon as it is fully utilized, $p_2(S,D)$ is redefined so that it no longer includes l_2. This will result in under-utilization of l_2, since traffic routed on $p_2(S,D)$ is no longer carried on l_2. As a result, l_2 will cease to be a bottleneck, and this will cause $p_2(S,D)$ to be re-defined so that it once again includes l_2. Hence the routing algorithm, as we have described it so far, is potentially unstable, oscillating between the needed two paths (in this example) and the insufficient single path.

The problem is that l_2 is fully utilized by the sum of the traffic which is traveling on p_1 and p_2, but is only partially utilized by the traffic which is traveling on p_1 alone. We would like to be able to use l_2 therefore as part of p_2. If, however, we need a third path, we do want to be sure that l_2 is not part of the third path. This suggests that the notion of a bottleneck must be relativized to a particular set of paths. The following definition seems to have the right properties.

Let packet p be a <u>type i packet</u> if and only if it is traveling on a path $p_j(S,D)$, where S is its source node, D

is its destination node, and $j \leq i$. Note that by this definition, all type 1 packets are also type 2 packets are also type 3 packets, etc.

Let a link l be a type i bottleneck if and only if it is fully utilized, and it carries some type i traffic.

Now, instead of defining path $p_{i+1}(S,D)$ as the shortest path from S to D which is bottleneck-disjoint from $p_i(S,D)$, we may define it simply as the shortest path from S to D which contains no type i bottlenecks. (Of course the shortest path containing no type i bottlenecks is not necessarily the same as the shortest path which is type-i-bottleneck-disjoint with p_i, since the latter path may contain any number of type i bottlenecks, as long as they are not in p_i. However, there is no reason to consider such paths, since they are already fully utilized). In the example above, l_2 would be a type 2 bottleneck, but not a type 1 bottleneck. Hence l_2 would be included in p_1 and p_2, but not in p_3, which is the desired result.

Note, however, that the routing process which runs in each source node does not need to know the utilization of each network link for each traffic type. It needs to know only two things. First, it must know the value of i (if any) for which the link is a type i bottleneck. Call this value of i the "bottleneck type" of the link. Second, it must know the total utilization of the

link. It needs to know the bottleneck type of the link in order to properly define the sequence of paths $p_1, p_2, \ldots, p_k$. It needs to know the total utilization of the link in order to know how much additional traffic can be carried by the link. But there is no reason at all for every node to know the utilization of every link with respect to each traffic type. Of course, we cannot deduce simply from the utilization of a link the amount of additional traffic which can be sent on the link; one also needs to know the total capacity of the link. Furthermore, we cannot assume that the capacity of a link will be a constant. The reason is that we may have multi-circuit or parallel links. We want to be able to treat a set of parallel links as a single link whose capacity is the sum of its components. However, since some of the component links may be up while others are down, the capacity of the composite link can change in real time. We will deal explicitly with this problem in the section on measurement and updating.

4.4 The Sequence of Sub-networks and the Forwarding Problem

Based on the considerations raised in the previous two sections, we suggest the following procedure for choosing a set of paths. Let G_1 be a representation of the network topology in which a link is assigned the length 1 if and only if it is operational, and is assigned infinite length if and only if it is down. For all $i>1$, let G_{i+1} be a representation of the network topology which assigns infinite length to all links which are either down or which are type i bottlenecks, and length 1 to all other links. This defines a sequence of sub-networks $G_1, G_2, \ldots, G_k$, each of which has fewer links of finite length than the previous one. Now we can run the SPF algorithm separately on each sub-network, resulting in a separate shortest path tree for each sub-network. The path from node S to node D on the tree of G_1 will be the shortest (i.e., min-hop) path from S to D. The path from S to D on the tree of G_{i+1} will be the shortest path from S to D which does not contain any bottlenecks of type i. Hence, this procedure will generate precisely the set of paths that we are interested in.

Note that the sequence of sub-networks is completely defined by the up/down status and bottleneck type of each link. This information is determined by the node from which the link emanates, and is disseminated to all other nodes in the form of a routing update. Assuming a quick and reliable updating process

(such as the updating protocol of the ARPANET's routing algorithm), all nodes will be in agreement as to the up/down status and bottleneck type of each link. Therefore, each node will define the exact same sequence of sub-networks G_1 through G_k. Now suppose that some source node S has a packet to send to some destination node D, and it has decided to send that packet on the shortest path which has no type i-1 bottlenecks, i.e., on the shortest-path tree of sub-network G_i. (We will discuss in a later section the criteria that a node will use in making such a decision.) All it need do is put the destination node number D and the sub-network number i in the packet header, and forward the packet to its next hop. Each intermediate node will forward the packet to the next node on its own shortest-path tree in sub-network G_i. Since G_i is the same in every node, and each node routes the packet on the shortest-path tree in G_i, we get consistent and loop-free routing (except, of course, for small, unimportant transients). It is not necessary, with this scheme, to have the source node pre-specify the entire path of the packet.

Recall that it is necessary for each node to determine the "highest" traffic type of all the traffic carried by each of its links. (If $j > i$, we may say that type j traffic is of a "higher" type than type i traffic.) This requires that there be some way to determine the type of each packet. Since each packet

will contain in its header the number i of the sub-network in which it is being routed, determining the type of a packet is quite trivial. If some particular packet is being routed in sub-network G_i, and $j \geq i$, the packet is of type j. (Note that a packet is defined to be of type i if it is traveling on the shortest-path tree of sub-network G_i, not if it is traveling on the i^{th} distinct path from its source to its destination. As we shall discuss later, these two notions are not the same.)

It is possible that, due to a routing update which is processed while a packet is in transit, a packet which is supposed to be routed in G_i will arrive at an intermediate node I only to discover that there is no longer any path from I to the destination D within G_i. This situation will occur whenever there is no path from I to D which does not contain bottlenecks of type i-1. If I and D are disconnected in G_1, then either D is down or the network is partitioned. In this case, there is no way to deliver the packet, and it should simply be discarded. If I and D are connected in some sub-network G_j, where $j<i$, then a path really does exist. The intermediate node I will have to choose some such sub-network G_j to forward the packet on and it will have to alter the packet's header appropriately. Considerations relevant to the proper choice of sub-network for re-routed packets will be discussed in section 4.6.

We cannot say a priori how many sub-networks will need to be represented in each node in order to obtain a number of distinct paths which is sufficiently large to maximize network throughput. Yet this is a very important question, since the number of sub-networks, as we shall see, is a large determinant of both the amount of memory and the amount of processing needed to implement this routing scheme. Unfortunately, we know of no way to answer this question short of doing a simulation. Of course, the number of sub-networks can be arbitrarily limited, with a resultant sub-optimality. Then the important question would be, just how sub-optimal is throughput if only x sub-networks are used, for some particular value of x? This question, however, is as difficult to answer as the other.

4.5 Measurement of Link Utilization

One might think that link utilization could be measured simply by counting the number of bits sent over the link during a given interval, but the situation is not so simple. We are not interested in the utilization per se, but rather in the amount of additional traffic that could be sent over the link. This depends not only on the utilization of the link, but also on the utilizations of various other nodal resources. For example, suppose that there is a period of time during which no traffic is transmitted on a particular link. Suppose also that during this period, there are no free buffers in the node. Clearly, if a packet which must be sent over the link were to arrive at the node during this period, it could not be sent, since there would be no buffer for it. If we are concerned with the ability of the link to carry additional traffic during this period, then we must regard the link as fully loaded during the period (since it can carry no additional traffic), not as idle. Similar effects can occur as a result of certain link level protocols. For example, some versions of HDLC allow only eight packets to be in flight at once on a link, so that after the eighth packet is sent, the link is "blocked" (i.e., no more packets can be sent) until an acknowledgment for the first packet is received. If the acknowledgment is slow in arriving, then there may be a period of time when the link, although not in use, cannot be used to carry

any additional traffic. Again, if we are concerned with the ability of the link to carry additional traffic, then we must regard the link as fully loaded during this period, not as idle. Another example of the same phenomenon might occur when the CPU itself is heavily utilized, as indicated by a long queue of packets waiting for the processor. If a packet arriving at the node will have a long wait before it can be transmitted over the link, even though the link is not actually in use during that period, then the link should be regarded as fully loaded.

The point we are making is the following. The multi-path routing algorithm that we are proposing will require each source node to know how much additional traffic can be sent on any given path. This requires that each node compute, for each of its outgoing links, the amount of additional traffic that can be sent on that link. Then this information must be sent to all other nodes in a routing update. However, there are many factors that control the amount of additional traffic that can be sent over any particular link. The actual utilization of the link is only one of these factors. Other such factors include the utilization of the CPU, the buffer space, and the available queuing slots. Still other factors have to do with whether the link is blocked by some protocol. To take all this into account, we propose the following series of definitions:

Let us say that a link l is idle at some particular instant if and only if all the following conditions hold:

a) No packet is in transmission on the link at that instant.

b) The link is not blocked by any protocol at that instant.

c) If no packet at all is in transmission on the link at that instant, then sufficient resources are available such that were a packet to arrive, it could be transmitted on the link. Such resources may include (but are not restricted to) buffer space, queue slots, and CPU cycles.

Let IDLE(l,t) be the amount of time during the interval t during which link l was idle.

Let TIME(t) be the length of interval t.

Let K(l) be the total capacity of link l.

Then we may define U(l,t), the utilization of link l during interval t, as follows:

$$U(l,t) = 1 - (IDLE(l,t)/TIME(t))$$

We may now define the residual capacity of link l during interval t, RESCAF(l,t), to be:

$$RESCAP(l,t) = K(l)*(1-U(l,t))$$

The bottleneck type of link l is infinite if the link is not a bottleneck. If it is a bottleneck, the bottleneck type is the smallest value i such that if any type j traffic was carried by link l, $i \geq j$.

The amount of additional traffic that could have been sent over the link l during the interval t is just RESCAP(l,t).

It is assumed here that each node will know the total capacity of each of its links. (If this is not known a priori, it can easily be determined on-line simply by transmitting a packet of known length and computing the time between the beginning of transmission and the arrival of the interrupt marking the completion of transmission.) In the case of parallel, or multi-circuit links, the measurement procedure is a bit more complicated. From the perspective of routing, a set of parallel circuits (circuits connecting the same pair of nodes) can be considered as a single circuit whose capacity is equal to the sum of the capacities of the individual circuits. Hence we would like to report a single value of RESCAP which gives the residual capacity for the entire set of parallel links. The definitions given here are directly applicable only to single circuits, but

can easily be extended to links which consist of multiple parallel circuits. Suppose link l consists of j parallel circuits $c_1,\ldots,c_j$. Then the utilization of each circuit can be computed separately according to the above definitions. The utilization of the entire composite link can then be computed according to the following formula:

$$U(l,t)=(U(c_1,t)*K(c_1)+\ldots+U(c_j,t)*K(c_j))/(K(c_1)+\ldots+K(c_j))$$

The remaining definitions go through as is.

There is, however, an additional complication due to the fact that one or more of the parallel circuits may be down during one part of the measurement interval and up during another part. If a circuit which is down during some period of time is considered to be idle during that period, the utilization of that circuit may be computed according to the above definitions. Suppose link l consists of circuits $c_1,\ldots,c_j$, where $c_1,\ldots,c_k$ are up at the end of the measurement interval, but $c_{k+1},\ldots,c_j$ are down at the end of the measurement interval. Then the utilization of the composite link l can be computed as follows:

$$U(l,t)=(U(c_1,t)*K(c_1)+\ldots+U(c_j,t)*K(c_j))/(K(c_1)+\ldots+K(c_k))$$

Roughly speaking, this yields the ratio between the amount of traffic sent over the composite link during the interval t and the total capacity of the link at the end of that interval.

Lines come up and go down relatively infrequently. Furthermore, once a link goes down (or comes up), we would expect it to stay down (or up) for longer than the length of a measurement interval. Therefore, we would expect the total capacity of the link during the next measurement period to be the same as its total capacity at the end of the current measurement period. The proposed definition of utilization takes this into account. Of course, this may result in a negative value of RESCAP for the multi-circuit link. A negative value of the reduced capacity indicates that the link was oversubscribed. It is desirable to be able to report negative values of RESCAP. We will discuss later the way in which such values will be used by the routing algorithm.

In fact, we would like to be able to report negative values of residual capacity for oversubscribed links even when those links do not consist of multiple parallel circuits. In this respect, our definition of RESCAP is defective, since it will never result in a value of less than zero (except in the multi-circuit case). While we believe that an algorithm can be developed to quantify the extent to which a link is oversubscribed, such an algorithm would be extremely dependent on the particular characteristics of the network node. In the ARPANET, we can get a measure of the oversubscription of a link by measuring the rate at which the node refuses packets destined

for that link because of a shortage of resources. But this procedure works only because of the particular effect that oversubscription has within the ARPANET IMPs. A different procedure might be needed in, e.g., the AUTODIN II PSN's; this would have to be determined by further study.

We note that there are certain packets which are not routed on the tree of any sub-network G_i. Among these packets are the routing updates themselves, the HELLO's and I-HEARD-YOU's used to test the quality of the link, and non-piggybacked link level acknowledgments. Such packets should be considered to be of type 0. Then the above definitions will hold as is.

The measurement procedure we have proposed is simple in principle, and could be implemented by setting timers in appropriate places within the code. Of course, the exact places where the timers must be set cannot be determined a priori. Neither can the length of the measurement interval t. We suggest that the link utilization be measured in blocks of several seconds. Our model is the ARPANET delay measurement procedure, which, every 10 seconds, determines the average packet delay over the previous 10 second period. However, we cannot say what the appropriate length of the measurement period within the AUTODIN II PSN should be, nor even what the measurement period in the ARPANET should be for this new kind of measurement.

It is important, however, that the measurement interval be long enough so that if the average input traffic matrix does not change, and the routing does not change, then the average utilization should not appear to change. The measurement interval should be the shortest interval meeting these desiderata. Determining the optimum length of this interval in some particular network will require actual measurements in that network, since the optimum length depends on how the network actually behaves.

We have not said whether or not the measurement intervals at all nodes should be synchronized. It is not possible to say, a priori, whether synchronization or non-synchronization will result in better network performance. In the ARPANET routing algorithm, where we measure delay, we have discovered that non-synchronized measurement intervals actually improve the network's behavior. However, without measurement or simulation results, we cannot say whether the same result would hold in other networks, or whether it holds for capacity measurements as well as delay measurements.

There is one other measurement issue of which we must take note. This is the issue of quantization. The value of residual capacity will have to be represented in some finite number of bits. This implies that the measurements will have to be quantized to some discrete value. For a given number of bits, a

coarse quantization allows a greater dynamic range of values to be represented, but a finer quantization allows a greater degree of accuracy. Increasing the number of bits used to represent the values, however, leads to a greater cost in nodal memory (needed to store the values) and in link bandwidth (since the values need to be transmitted in routing updates). Choosing the correct number of bits and the proper unit of quantization are engineering problems that cannot be solved without intimate knowledge of the network design and implementation.

4.6 Apportioning the Flows to the Paths

We have now discussed the procedure used for selecting a set of multiple paths between a given source and destination (although a detailed specification of the algorithm will not be given until later). We have also discussed the procedure used to route a packet on a particular path. However, we have not yet discussed the procedure whereby a given source node determines which of the several available paths should be selected for routing a particular packet. That shall be the concern of the present section.

Recall that the path-selection algorithm works by defining a sequence of sub-networks $G_1, G_2, \ldots, G_n$, and creating the shortest-path tree in each sub-network. For a given source node S and destination node D, these will be some value of k, $1 \leq k \leq n$ such that there is a path between S and D in all G_i, $i \leq k$, but no path between S or D in the G_j, $j > k$. Furthermore, it will not be the case in general that a distinct path between S and D will exist in each of the G_i. For example, if the path between S and D in G_m contains no links which are type m bottlenecks, then the path between S and D in G_{m+1} will be identical to the path between S and D in G_m. So the first step is to determine which of the G_i provide distinct paths to the destination D. There are several simple ways to do this. One method is just the "brute force" method. For each $m < k$, one can explicitly trace the paths

in G_m and G_{m+1} to D at the same time, to see if any discrepancies are detected. Alternatively, one might create a bit vector representing the path to each destination in each sub-network. This vector would have a bit for every link in the network. A value of 1 for some link would indicate that that link is in the path to that destination in that sub-network. Then one could determine whether the two paths are distinct simply by taking the "Exclusive-Or" of the bit vectors. A non-zero result would indicate that the paths are distinct. As another alternative, one could make use of the following property of the path-selection algorithm. Define the minimum bottleneck type of a path to be the minimum value of the bottleneck type of each of its links. (A link which is not a bottleneck at all may be considered to have an infinite bottleneck type.) If p is a path from S to D in G_i whose minimum bottleneck type is j-1, it follows that j>i, and that the next distinct path from S to D will be found in G_j, unless, of course, there is no additional distinct path (which will be indicated by S and D being disconnected in G_j). As we shall show later, the SPF algorithm can easily be modified to associate with each destination node the minimum bottleneck type of the path to it in each sub-network. This can be used for determining the sub-network which contains the next distinct path to that destination.

Using one of these procedures, each source node S can specify a sequence of k distinct paths $p_{g_1}, p_{g_2}, \ldots, p_{g_k}$ to each destination D, where p_{g_i} is the path from S to D on the shortest-path tree of sub-network G_{g_i}. (We use the double-subscript notation to emphasize that the subscript on a path name indicates the sub-network for which that is a shortest path. The subscript does not indicate how many paths are actually being used -- that is indicated by the subscript of the subscript, where present. Thus path p_i is the shortest path in the sub-network G_i. It may not, however, be the i^{th} path in use. Where we indicate a path with double subscripts, as in p_{g_i}, it is the i^{th} path in use, and also the shortest path in sub-network G_{g_i}.) Since the routing updates indicate the amount of residual capacity in each link, the SPF algorithm can be modified (as we specify later) to associate with each destination the residual capacity of each path to it. This allows us to compute the maximum amount of traffic that S may send to D along path p_i without overloading the path (assuming, for the moment, that no other source-destination flow along that path will increase simultaneously). This is just the sum of the residual capacity on that path plus the amount of traffic that S has already been sending to D on that path. That is, if the residual capacity of the path is positive, S can increase its flow to D along that path. If the residual capacity is negative, then the path is overloaded, and S must decrease its flow to D along that path.

If the residual capacity is zero, S must not increase its flow in the path, but it need not decrease it.

We can make these ideas more precise, as follows. We suppose that S is constantly measuring its flow to D along the shortest-path tree of each subnetwork G_i. Or rather, that S is measuring its average flow over some interval t. The measurement interval must be long enough so that the measured value of average flow does not change unless the actual flow changes. (That is, the effects of stochastic variations in steady-state flows must be smoothed out. This is just the same requirement as exists concerning the interval of measurements for the link utilization measurements.) Let FLOW(S,D,i,t) be the average flow from S to D along the tree of G_i during interval t. Let MAXFLOW(S,D,i,t) be the maximum permissible flow. Let RESCAP(S,D,i) be the residual capacity of the path from S to D in G_i, based on the most recently received set of updates from the links along that path. (The procedure used by a node to decide when to send an update will be discussed in a later section.) Then we can define:

MAXFLOW(S,D,i,t+1) = MAX(0,FLOW(S,D,i,t) + RESCAP(S,D,i))

where t+1 is the measurement interval following interval t.

The above definition is not entirely satisfactory as it stands, due to the following problem. Each node in the network

will know, as a result of receiving routing updates, the residual capacity of each link. If all nodes decide, at the same time, to increase their flows so as to make use of that residual capacity, then oversubscription and consequent congestion is possible. Therefore, each source should be able to make use of only a fraction of the residual capacity at any one time. Similarly, if the residual capacity is negative, it is possible that the oversubscription is due not to a large flow from one source, but to a combination of flows from several sources. Then no one source ought to be forced to reduce its flow by the entire amount of the oversubscription. Rather, each source using the oversubscribed link ought to reduce its flow by a fraction of the oversubscription. So the proper definition of MAXFLOW is:

MAXFLOW(S,D,i,t+1) = MAX(0,FLOW(S,D,i,t) + f*RESCAP(S,D,i)

where f is the appropriate fraction. The choice of the fraction f will depend on the probability of several source nodes reacting at once. Furthermore, it is possible that the value of the fraction should change with changing conditions, or that different fractions should be used, depending on whether the residual capacity is positive or negative. It may also be useful to make the fraction be a function of the size of the flow, so that flows which are already large are allowed only small increases, while flows which are small would be allowed large increases. When RESCAP is negative, we might want to force large

flows to make large decreases, while forcing small flows to make only small decreases. This procedure will tend to equalize the sizes of the various flows, thereby bringing about fairness in the use of the network. We do not know any a priori way to choose the fraction. To make the proper choice, it will be necessary to use simulation or measurement to evaluate the effect that various choices have on the system.

Recall that the purpose of our routing algorithm is to minimize the number of packet-hops. This implies that we wish to fully load shorter paths before we begin to use longer paths. If $i>j$, then we know that the length of the path from S to D in G_i is greater than or equal to the length of the path from S to D in G_j. We would like to use as few of the available paths as necessary to handle the traffic, while dividing the traffic on those paths in proportion to the capacity available on those paths. This suggests the following means of apportioning the flow from S to D among the available paths.

Assume that we have available the sequence of k paths $p_{g_1}, p_{g_2}, \ldots, p_{g_k}$ for sending traffic from S to D and that these k paths are the smallest set of available paths which provide sufficient capacity to handle the offered flow from S to D, assuming that this does not increase during the next measurement interval. (There may, in general, be more than k paths available at this time -- we are concerned only with the fewest number of

distinct paths which can handle our flow.) Let TOTALFLOW(S,D,t) be the sum over all i of the values of FLOW(S,D,i,t). Then for all $i<g_k$ such that p_i is one of the available paths, let P(S,D,i,t+1)=MAXFLOW(S,D,i,t+1)/TOTALFLOW(S,D,t). Then let SUMP(S,D,t+1) be the sum over all i such that p_i is one of the k paths and $i<g_k$ of the values of P(S,D,i,t+1). Finally, let $P(S,D,g_k,t+1)=1-SUMP(S,D,t+1)$. We now have a value of P, which must be between 0 and 1, associated with each of the k paths. The traffic apportionment algorithm is specified as follows:

> As source node S receives traffic for destination node D during measurement interval t, it shall apportion the traffic among the paths $p_{g_1},p_{g_2},\ldots,p_{g_k}$ according to the fractions $P(S,D,g_1,t),P(S,D,g_2,t),\ldots,P(S,D,g_k,t)$. Traffic shall be apportioned in small enough units so that a true simultaneous use of multiple paths is achieved. (Obviously, if traffic were apportioned in units comparable to the average link bandwidth, then true multi-path routing would not be achieved -- only one path would be used at a time.) However, in no case shall the amount of traffic sent on a path p_{g_i} in an interval t exceed the value of $MAXFLOW(S,D,g_i,t)$. When the maximum amount of traffic during the interval t has been sent on each of the first k-1 paths, any additional traffic S receives for D during that interval will be

sent on path p_{g_k}. When the amount of traffic sent on p_{g_k} reaches MAXFLOW(S,D,g_k,t), S shall send any additional traffic on path $p_{g_{k+1}}$, if such a path exists. When the amount of traffic sent on path $p_{g_{k+1}}$ exceeds MAXFLOW(S,D,g_{k+1},t), S shall begin using path $p_{g_{k+2}}$, if such exists. When S has sent the maximum allowable amount of traffic on all available paths, it shall cease accepting traffic for D until the start of the next measurement interval.

It is easily seen that if the flow remains the same from one measurement interval to the next, the scheme will result in the full loading of the shorter paths, with the longer paths receiving only as much traffic as cannot be fit on the shorter ones. Furthermore, since received flow is to be distributed according to the proportions specified by the fractions P (until such time as an increase in flow is detected), there is a true simultaneous use of the several paths. If the flow should increase, the increase in flow will appear as a surge on the k^{th} path and succeeding paths. However, in the next measurement interval, the increased value of the flow will be automatically smoothed over the k paths as the updates report the increased flow, causing the values of P to be altered. Should the increase in flow fill the last available path (by causing one of its links to become a bottleneck), an additional path will be selected

automatically (if any such exists) as soon as the update reporting the link's new bottleneck type is received. If the flow decreases, on the other hand, the procedure will not result in the full loading of the shorter paths. The probable result is that some links which are bottlenecks may no longer be so. When the updates reporting the new bottleneck type of these links are received, the number of available paths will decrease, until the new flow is again matched with the proper number of paths.

There is one additional subtlety that must be mentioned. We have spoken of the need to allow each source node to take for itself only a fraction of the residual capacity on a link, so as to leave room for other sources who may also try to increase their flows over the same link at the same time. However, a similar problem arises within a single source node. That source node may have flows to several destinations which travel over the same link. Nothing we have said so far prevents a source from oversubscribing a link by trying to use the same residual capacity for flow to several destinations simultaneously. This can be prevented by the following procedure. A table can be maintained with an entry for each network link. At the beginning of each measurement interval, each entry in the table is initialized with the value of that link's residual capacity, multiplied by the fraction f (discussed previously). Whenever S sends enough traffic to D on a path in an interval t so that it

begins to cut into the residual capacity (i.e., when it sends more traffic on some path p_i in this interval than in the previous interval), it must subtract that amount of traffic from the table entry for each link in the path. When the value of the entry for a link becomes zero, the source acts as if the amount of traffic sent had reached MAXFLOW(S,D,g_k,t). This procedure enables the source to apportion the residual capacity on a link among several destinations on a first-come, first-serve basis.

A simpler alternative procedure would be for each source node to allocate a particular fraction of the residual capacity to which it is entitled to each of its flows, with the fraction being inversely proportional to the size of the flow. While this would be less exact, its effect might be almost the same, or even better, than the more complex procedure. This same simple procedure would also work when residual capacity is negative, except that the fraction would be directly proportional to the size of the flow.

The flow apportionment procedure discussed in this section is really a sort of global, long-term congestion control. It tries to match the amount of flow sent on each path to the capacity of the path. This may be considered to be "global" congestion control because it is effected by each source node based on knowledge of the residual capacities of all links in the network. It is "long-term" because it is based on measurements

which are averaged over periods of time which are long relative to average network transit times. It is worth emphasizing that although routing and congestion control may be considered to be separable problems, any form of multi-path routing whose purpose is to maximize throughput must be integrated with global long-term congestion control. There is, after all, no point in having a path selection procedure which produces an optimal set of paths, unless the flows are adequately matched to the path capacities. However, it must be remembered that our flow apportionment scheme will not, by itself, prevent surges if there is a sudden increase in offered traffic. Nor will it ensure that a source node does not overload a destination node by sending faster than the destination node can receive. Thus there will still be a need for short-term local congestion control procedures, as well as for end-end flow control.

It must be noted that there are certain sorts of packets for which this sort of congestion control is inappropriate. For example, to prevent deadlocks, it may be necessary to send control packets even when the control variables prohibit sending any more traffic. The same considerations arise with respect to re-routed traffic (see section 4.4). One reasonable way for a node to handle re-routed traffic is to treat it just as if it were traffic entering the network locally, distributing the traffic according to the fractions P. However, since re-routed

traffic is already in the network, it cannot be refused when values of MAXFLOW are reached. When control packets or re-routed packets must be transmitted, even though our scheme "officially" allows no more traffic to be sent, we suggest distributing this excess traffic uniformly over the available paths.

We emphasize that the measurements proposed in this section raise all the same questions with respect to quantization, synchronization, and length of the measurement interval as do the measurements proposed in section 4.5. Again, these are questions that cannot be answered a priori.

It is important to note that a given network resource may be fully utilized either by a maximum number of bits or a maximum number of packets. In particular, a link may be fully utilized by a large number of small packets, even if the total number of bits in those packets (including overhead) is less than the speed of the link, as measured in bits per second. The reason, of course, is that many of the resources which a packet needs in order to be transmitted on a particular link (e.g., buffers, queuing slots, link protocol id numbers) are assigned on a per-packet basis, and are as heavily utilized by short packets as by long ones. The flow measurements done at the source nodes, if they are to be truly useful for flow apportionment and congestion control, must reflect this fact. This can be accomplished by the following procedure. Suppose we decide to treat a 50 kbps link

as being able to handle 100 (normalized) units of flow. If such a link can handle either 50,000 bits per second or 100 packets per second (regardless of packet size) before becoming saturated, then an n-bit packet should be counted as MAX(1, n/500) units of flow.

We have not suggested any explicit algorithm for computing the fractions P or for actually using those fractions to apportion the traffic. There are many possible algorithms for achieving these functions, and it seems to us that the choice of algorithm would depend almost exclusively upon implementation considerations. We point out though that we do not assume that the apportionment algorithm will be able to split the traffic in the precise proportions given by the fractions. Any algorithm for actually computing the fractions will introduce errors of accuracy, and it seems unlikely that any algorithm for splitting the traffic in accordance with the fractions will be able to effect an exact split. Our algorithm presupposes only enough accuracy in the apportionment procedure so that the shorter paths can be fully loaded before the longer paths are used. As we shall see in the next section, when we discuss the generation of updates, we have considerable freedom in deciding when a measured change in the utilization of a link is significant enough to warrant an update. Inaccuracy in the apportionment process may introduce some fluctuations in link loadings, even under

steady-state conditions. The measurement and updating process, however, can be made insensitive to these fluctuations by proper choice of parameters. We return briefly to this issue at the end of section 4.7.

4.7 The Generation of Updates

There are five situations in which a node may send a routing update to all other nodes. These are (a) a link emanating from the node comes up, (b) a link emanating from the node goes down, (c) at the end of a measurement interval, the residual capacity of one of its outgoing links has changed significantly since the last time an update was sent, (d) the bottleneck type of one of its outgoing links has changed since the last time an update was sent, and (e) although there are no changes to report, enough time has elapsed since the last update was sent that a new one must be sent to ensure reliability. We assume that the protocol used to disseminate the updates will be the same as the updating protocol of the ARPANET's routing algorithm [1,2,4], which is the only protocol known to ensure sufficient reliability for the purposes of routing. Among the features of the protocol are the following:

a) Each update from a given node will contain an entry for each link emanating from the node.

b) Previous updates from a node are made obsolete by later updates.

c) The updates are flooded over all network links. If two nodes are connected by a set of parallel circuits, the update need be sent over only one of the circuits, and

the acknowledgment need not return over the very same circuit.

d) There will be point-to-point positive acknowledgments and retransmissions.

e) The updates will be sequenced.

f) The updates will be timed out to prevent sequence number wraparound problems.

g) Transmission of routing updates will be treated with higher priority than transmission of ordinary data or end-end control packets.

h) Routing updates shall be generated periodically even if there has been no change in the values updated.

i) When a failed link is ready to resume operation, it is held in a special waiting state for an amount of time equal to the maximum length of the period between the generation of two successive updates by a single node. During the waiting period, only routing updates may be sent over the link, and the link appears dead to the routing algorithm. The purpose of this is to ensure that a failed node does not rejoin the network until it has received routing updates from all other network nodes. This enables it to have complete, up-to-date

routing information by the time it rejoins the network. This procedure also ensures that if the network is partitioned into two or more segments, the segments are not fully rejoined until after a complete set of updates from each segment has traveled into the others.

It is known that the SPF algorithm, in conjunction with this updating protocol, is reliable, in the sense that it will always be able to deliver a packet to a destination, without any long-term looping, as long as there is a physical path from the packet's present location to the destination which has the capacity to handle the packet. We think it obvious that the multi-path algorithm we have proposed, if coupled with the ARPANET's updating protocol, will be equally reliable in the same sense. Therefore, we shall not comment any further on this aspect of the updating scheme. However, the times at which updates are generated, and the precise information carried in the updates, can have a significant effect on the stability of the routing scheme. In the remainder of this section, we discuss these issues in more detail.

When a link fails, we need to generate a routing update reporting that fact as soon as possible. If heavy flows are being directed toward a dead link, congestion will surely result unless an update causing them to be re-routed is sent as soon as possible. A node indicates that one of its links has gone down

by generating a routing update packet which contains no entry for that link. Note, however, that instability can result if the link goes up and down very rapidly for short periods of time. We assume that each pair of neighboring nodes runs a protocol between them to determine whether the link is of operational quality. This protocol must ensure that the link does not flap up and down too quickly. Furthermore, the protocol should ensure that the link is considered down only when the network can actually carry less throughput by using the link than it can by not using it. (Suppose, for example, that the link's error rate were so high that a packet would have to be re-transmitted many times, on the average, before getting over the link. If the retransmission process takes so long that the user resubmits his packet to the network, we have the result that the network may have been able to carry more throughput had the link been declared down.) Also, the determination as to whether the link is up or down must be independent of the amount of traffic flowing over the link. In some networks, a link will be declared down whenever a data packet sent over the link is not acknowledged after a fixed number of retransmissions. The assumption, apparently, is that the only reason the acknowledgment has not arrived is that the link is not of sufficient quality to carry it. However, it is usually true that a node will not acknowledge a packet unless there are sufficient nodal resources to forward the packet. As a result, a perfectly

good link may be declared down simply because there is congestion of nodal resources. But if links can go up or down simply because of the amount of traffic directed over them, instability of the routing process is sure to result.

When a link comes up, we want to generate a routing update immediately, informing all other network nodes of this fact. However, we would like to exert some control over the way in which the new link is utilized. Some source-destination pairs will be able to use the new link for their type 1 flows. Others may only be able to use it for their type 2 flows. In accordance with the principle of attempting to minimize the number of packet-hops, we would like to ensure that the new link becomes as heavily utilized as possible with type 1 flows before any type 2 flows are placed on it. Then, if there is still residual capacity in the link, we would like to ensure that it becomes as heavily utilized as possible with type 2 flows, before we allow any type 3 flows to be placed upon it, etc., etc. This can be ensured via the following procedure. Recall that the update entry for each link specifies the residual capacity for the link and the bottleneck type of the link. All nodes, upon receiving a routing update, must base their computations upon the information in that update. That is, no node may use its own local information to alter the information in a routing update. Otherwise, there is no way of ensuring consistent and loop-free

routing. However, when a node generates an update to report on the state of its outgoing links, it can put whatever values it likes into the update, knowing that all other nodes must believe it. Ordinarily, the values for residual capacity and bottleneck type which are placed in an update will be the actual values measured by procedures which we have discussed earlier. However, in certain cases, a node may want to place values in an update which differ from the measured values. One of these cases exists when a link comes up.

When a link first comes up, its residual capacity is equal to its total capacity, and it is not a bottleneck at all. However, when the network is heavily loaded, and a link comes up which, initially, is carrying no traffic at all, there is likely to be a mad rush for it as all nodes direct their traffic towards the new link. This is likely to overload the link, resulting in instability. One way of enhancing the stability of the scheme is to try to load the link with type 1 traffic before allowing any type 2 traffic on it. To bring this about, the update that first reports the link to be up should declare the link to be a type 1 bottleneck. The next update generated by the node should declare it to be a type 2 bottleneck (unless, of course, it has really become a type 1 bottleneck in the meantime). Each succeeding update should increase the reported value of bottleneck type for that link by 1, until the link actually does become a type i

bottleneck for some i, or until the updated bottleneck type exceeds the number of sub-networks. This procedure will help to control access to the link, thereby enhancing stability. We emphasize again that only the node which generates an update has any freedom as to what values of residual capacity or bottleneck type to assign to a link. Other nodes must always base their computations on the values reported in the updates.

Suppose a node detects, at the end of an interval, that the bottleneck type of a certain link has changed. This must cause an update to be generated, so that other nodes can be made aware of the change. Suppose the bottleneck type has decreased. This means that the link will be assigned infinite length in more sub-networks than previously. The limiting case of a decrease in bottleneck type is a link's going down, which causes it to be assigned infinite length in <u>every</u> sub-network, including G_1. So the same considerations apply. When a decrease in bottleneck type is detected at the end of a measurement interval, an update should be generated immediately, reporting the new bottleneck type.

However, if the bottleneck type of a link <u>increases</u>, it will be assigned infinite length in fewer sub-networks than previously. Hence, this case is analogous to a link's coming up, and the same considerations apply. No matter how large the increase in the link's bottleneck type actually is, the initial

update should report an increase of 1. Successive updates should continue to report increases of 1 in the bottleneck type until the actual bottleneck type is reached.

This procedure of only slowly and gradually reporting increases in bottleneck type may also be important to maintaining the stability of the routing scheme. The routing scheme works by assigning three values to each link: a length, a residual capacity, and a bottleneck type. The length of a link is fixed, and has no relation to the amount of traffic being sent on the link. Since there is no feedback relationship between link length and traffic load, there is no possible source of instability related to link length. There is, of course, a feedback relationship between the amount of residual capacity on a link and the amount of traffic sent over it. However, it does not seem that this feedback relationship can, by itself, lead to serious instability of the routing scheme. Changes in residual capacity will not, by themselves, lead to changes in the number of, or identity of, paths selected between some given source-destination pair. Rather, such changes will lead only to incremental changes in the amount of flow sent on the various paths. The real potential source of instability lies in the feedback relationship between the bottleneck type of a link and the amount of traffic routed over the link. It is the bottleneck types of the links which determine the number of paths that the

path selection algorithm makes available between a given source-destination pair. Other things being equal, an increase in bottleneck type of some link will tend to cause a reduction in the number of distinct available paths. Our scheme for apportioning the traffic is supposed to fully load the shorter paths before sending any traffic in the longer paths. Suppose, however, that during some interval, some source S fails (for whatever reason) to fully load its min-hop path to destination D, even though the total flow it sent to D exceeds the amount of capacity available on a single path. This failure to fully load the shorter path may result in all the links of that path becoming non-bottlenecks. If this happens, S may find that, in the next interval, it has only a single (insufficient) path to D. It will now fully load the path, thereby causing additional paths to be selected, but this may cause an oscillation between one path and several. We would not expect this sort of instability to occur if the flow apportionment algorithm is working properly. Note, however, that even if there is a problem with the flow apportionment, the procedure of only slowly and gradually reporting increases in bottleneck type will have a significant dampening effect on any oscillations that might otherwise occur.

This sort of problem can be regarded as a problem in congestion control. That is, there may be inherent stability problems in congestion control procedures which are based on flow

measurements. These problems will be discussed further in section 4.8, where the congestion control properties of the flow apportionment procedure are discussed in much greater detail.

When a node measures a significant decrease in residual capacity on one of its links, an update must be generated and broadcast so that the source of the flow will know to reduce the amount of traffic sent on the overloaded path. Since a long-term steady-state overload of a path will lead inevitably to congestion and possible message loss, information about the existence of such an overload must be made available as soon as possible after the overload is detected. On the other hand, if there has been an increase in residual capacity, we must worry about the possibility that too many sources will try to utilize this extra capacity at once, thereby causing an overload. We have attempted to minimize the effect of this phenomenon by allowing each source node to take for itself only a fraction of the residual capacity along each path. Another means of reducing the likelihood of overload is the following: whenever an increase in residual capacity is "very large," report in the routing update only a fraction of the increase. This procedure may increase the time (by one or two measurement intervals) it takes to fully load the link, but will make the possibility of overload much less likely.

It must be remembered that we want to load the links only to a certain level. If a link is loaded above that level, we want the routing algorithm to act as if the link is overloaded, even though it actually has positive residual capacity. For example, if some link is 90% loaded, but we would like to operate our links only at the 80% level, we wish the routing algorithm to react as if the link is 10% oversubscribed, that is, as if it had a residual capacity of -10%. The way to achieve this, of course, is to place a negative value of residual capacity in the update.

We have been using such terms as "very large," "significant change," etc., which must of course be quantified when the routing scheme is implemented. It is the values assigned to these parameters, along with the length of the measurement interval, that will determine the rate at which updates are generated and broadcast. The more frequently the updates are sent, the more reactive the scheme will be, and the more expensive it will be (in terms of link bandwidth devoted to carrying routing updates). Setting the parameters to make the routing scheme sufficiently accurate, sufficiently stable, sufficiently reactive, and sufficiently inexpensive involves making a set of engineering trade-offs and compromises which is specific to a particular network, and which can be made only by the system engineers of that network. We have tried to design the scheme with enough engineering "handles" so that such

trade-offs can be easily implemented, once the policy decisions are made.

It must also be understood that the settings of certain of these parameters can cause interactions with other parts of the routing scheme. Such interactions must be very carefully considered when tuning the parameters. For example, one might think that best performance could be obtained by considering every change in utilization, no matter how small, to be "significant." However, as we have pointed out in the previous section, inherent inaccuracies in the apportionment algorithm may cause small fluctuations in the utilization of a link, even under steady-state conditions. In order to avoid spurious routing changes and the resultant instabilities, the threshold of significance should be chosen large enough so that reaction to such small fluctuations does not occur. The parameters of one part of the routing scheme cannot necessarily be tuned in isolation from the other parts.

A very important consideration in the decision as to whether our proposed routing algorithm ought to be implemented is the amount of link bandwidth needed to carry the routing updates. Unfortunately, there is no way to predict this exactly, since it depends on the design of the update packets, and on the frequency with which updates would have to be sent, which in turn depends on the settings of the various paramters. In the ARPANET, each

routing update consists of 136 bits (which includes 72 bits of hardware framing characters) plus an additional 16 bits for each neighbor of the node that generated the update. This 16-bit field is needed to identify the neighbor (8 bits) and to indicate the delay to that neighbor (8 bits.) Since the average connectivity of the ARPANET is 4.5, this means that the average update size is 176 bits. During peak periods in the ARPANET, each node generates an update, on the average, approximately once every 38 seconds. With 65 nodes, this comes to about 300 bits/sec, or about 0.6% of a 50 kbps link. In AUTODIN II, for example, the average size of an update packet would be larger. One reason for this is that the average connectivity would be between 4 and 5 rather than 2.5, so each update would have to report on a larger number of neighbors. Also, rather than just giving neighbor number and delay for each neighbor, each update would have to have three fields for each neighbor, viz., neighbor number, residual capacity over the link to that neighbor, and the bottleneck type of that link. Thus instead of 16 bits per neighbor, it is likely that at least 20 bits will be needed, and possibly as many as 24. However, the fact that AUTODIN II will have a smaller number of nodes than the ARPANET may compensate somewhat for the larger size of the update packet. The most serious difficulty in estimating the amount of bandwidth needed to carry the routing updates is the fact that it is impossible to tell at present how often updates would have to be generated.

The frequency with which updates are sent is under the control of the system engineers, and can be tuned to be any desired frequency. However, the frequency of updating would be related to the performance of the routing scheme, and it would not be wise to set it arbitrarily. As long as the network is trunked with lines whose capacity is at least 50 kbps, though, the bandwidth requirements should still be modest.

4.8 More on Stability: Basing Congestion Control on Flow Measurements

In this section we discuss a problem with the flow apportionment or congestion control scheme that is integrated into the multi-path throughput oriented routing algorithm. The problem has to do with the proper way to set MAXFLOW when there is a change in the identity of a path (in a particular sub-network) to a particular destination. More generally, the problem has to do with the potential instability of any congestion control scheme which is based on measurements of link utilization.

Recall that there are two different sorts of measurements that must be made by the nodes, each requiring a "suitably long" measurement interval for the purpose of obtaining a smoothed result. Each source node must measure the amount of traffic it sends to each destination node, along the shortest-path tree in each of the sub-networks. Also, each node must measure the residual capacity of each of its outgoing links. Let us assume for the moment that all measurement intervals (i.e., for both kinds of measurements) in all nodes are synchronized (and of the same length). Furthermore, we will assume that there is a period of time, which we may call an "adjustment interval," between successive measurement intervals. That is, rather than one measurement interval beginning at the precise moment the previous

one ended, the two successive measurement intervals will be separated by an adjustment interval during which no measurements are made. The adjustment interval should be long enough for all updates generated by the various nodes at the end of a measurement interval to be received at all other nodes and for the SPF and CAA computation to be fully run on all updates. That is, the adjustment interval corresponds to the transient period during which routing changes are taking place. Note that, barring topological changes, no routing changes will take place during a measurement interval, but only during the adjustment interval. Since topological changes are relatively rare events (i.e., the average number of toplogical changes during a measurement interval is much less than one), we will ignore them for the moment. We may assume then, that at the end of a measurement interval, we can speak of both "the old path" and "the new path" for each source-destination flow in each sub-network. The old path is the one that was used during the previous measurement interval. The new path is the one in use at the very end of the adjustment interval, i.e., during the next measurement interval.

At the end of each adjustment interval, changes must be made to the values of MAXFLOW and to the apportionment fractions P. The values of P are fully determined by the values of MAXFLOW, so we need only consider the procedure used for changing the value

of MAXFLOW. Recall that each source maintains a value of MAXFLOW, as well as a value of measured FLOW (in the measurement interval immediately preceding the adjustment interval), for each path (i.e., for the path in each sub-network) to each destination. In considering whether to alter the value of MAXFLOW for a path in a given sub-network to a given destination at the end of the adjustment interval, there are five cases to consider:

1) There has been no change during the adjustment interval in the path to that destination in that sub-network (i.e., the path is exactly the same, and there has been no change in the residual capacity along that path), and the most recently measured value of FLOW is the same as the previously measured value (in the penultimate measurement interval). In this case, there is no change to the value of MAXFLOW.

2) There has been no change in path, but the amount of FLOW measured in the previous measurement interval is different from the amount measured in the interval before that. This requires a recomputation of MAXFLOW, according to the formula described in section 4.6. After all, if the residual capacity of the path has remained the same, but the FLOW has increased, then some other flow along that path must have decreased. In

effect, this is an increase in residual capacity, and should result in an increase in MAXFLOW. If the residual capacity has remained the same, but the FLOW has decreased, then some other flow must have increased, and the value of MAXFLOW should be reduced.

3) There has been no change in the identity of the path (i.e., the path contains all the same lines and nodes), but there has been a change in the residual capacity of the path. Then clearly, a new value of MAXFLOW must be computed according to the formula given in section 4.6.

4) There is a new path to that destination in that sub-network, and the path is fully link-disjoint from the old path. Thus the new value of MAXFLOW should be set to a fraction of the residual capacity of the new path. Note that the fraction of the residual capacity should not be added to the previouly measured FLOW, as in cases 2 and 3 above. The previously measured FLOW is not relevant since it was measured along a completely different path. Another way to put the point -- the previously measured flow from that source to that destination along the new path is zero.

5) There is a new path from the source to that destination in that sub-network, but it has some links in common

with the old path. In this case, computation of the new value of MAXFLOW is somewhat more complex. In the case where the path has not changed during the adjustment interval, we can define the "available capacity" for a flow to be (a portion of) the residual capacity of the path plus the throughput of the flow during the previous measurement interval. If the path has changed, however, it will contain some links which, in the previous measurement interval, carried no traffic of the flow. In this case, the available capacity of the path cannot be computed simply from the residual capacity of the path and the previous amount of flow. Rather, the available capacity depends upon whether or not the link of least residual capacity on the new path is also on the old path. If not, then the value of MAXFLOW must be set to a fraction of the path residual capacity, without adding in the throughput of the previous measurement interval. However, if the link of least residual capacity is on the old path, it does not follow that MAXFLOW can be set to a fraction of the residual capacity plus the old flow, because this value may exceed the amount of residual capacity on (at least one of) the links which were not on the old path. Hence the proper value of MAXFLOW must be computed as follows. Let L1 be the set of links which are on both the new and

the old paths. Let L2 be the set of links which are on the new path only. Let RC1 be the minimum residual capacity of all links in L1, and let RC2 be the minimum residual capacity of links in L2. Let f be the fraction used for allocating a portion of the residual capacity of a path to a particular flow. Let OLDFLOW be the throughput of the flow in question during the previous measurement interval. Then let MF1=OLDFLOW+f*RC1, and let MF2=f*RC2. Then MAXFLOW=min(MF1,MF2). This computation ensures that the setting of MAXFLOW is determined by the link of least available capacity, not the link of least residual capacity. The computation, however, is somewhat cumbersome. While computing path residual capacity can be done quite efficiently with a simple modification of the SPF algorithm, there does not seem to be any such efficient means of computing path available capacity. The reason is that "available capacity" is not solely a function of the residual capacities on the links, but rather depends also upon the source and destination nodes.

So we see that setting MAXFLOW to its precise optimum value can be rather expensive, even if we assume synchronization of all measuremet intervals. The situation is further complicated if

there are topology changes during a measurement interval (as opposed to during an adjustment interval). Topology changes may cause path changes in the middle of a measurement interval. If this happens, the notion of "old path," i.e., the path that was used during the previous measurement interval, is not well-defined. The value of the flow measurement taken during the previous interval will not correspond to any one path, so the procedure suggested above in 5) will not be applicable. (We will, of course, be able to measure the flow on the new path, but this measurement will not cover a full measurement interval, and hence may not be suitably smooth.) This problem can be alleviated to some extent by delaying "link coming up" reports until the next adjustment interval. However, reports of links going down can never be delayed, so the problem will always exist to some extent. Since link failures are relatively rare occurrences (relative, that is, to the length of a measurement interval), this problem should not be serious, as long as the values of MAXFLOW eventually converge to reasonable values. A short period of inaccurate flow control (and consequent path apportionment) is not a major problem, as long as the scheme can recover.

There are a number of technical reasons why synchronization of all measurement intervals is not desirable. With synchronization, all routing "work" (i.e., the sending of updates

and the SPF computations) is done in a short interval by all nodes. This can be expected to create a large spike in the utilization both of line and node bandwidth, which will certainly degrade network performance. Furthermore, synchronization of the measurement intervals may be undesirable from a control-theoretic point of view. This turned out to be the case for the ARPANET routing algorithm (see BBN Report No. 3940, Appendix A), where it was determined that synchronization of the measurement intervals would seriously decrease the stability of the routing algorithm. If a similar result holds for the multi-path algorithm, synchronization would not be acceptable. Yet without synchronization, the problem we are discussing would arise not only when there are topological changes, but also whenever there is any path change.

We see then, that the sort of congestion control we have integrated into the multi-path algorithm, and consequently, the sort of flow apportionment we have proposed, are useful only in conjunction with routing algorithms that have the following property: changes in the identity of the path (or paths) being used to carry traffic from a given source to a given destination are infrequent when compared to changes in the residual capacity along the same path. Since the delay-oriented routing algorithm of the ARPANET may not have this property, the simpler congestion control scheme described in chapter 6 may be more effective in

combination with delay-oriented routing. (It may seem odd that a simpler scheme can, under certain circumstances, be more effective than a more complex one. However, this is just a consequence of the fact that the more "moving parts" a mechanism has, the more likely something is to break. Unfortunately, this maxim has not been as well appreciated by software designers as by hardware designers.) On the other hand, congestion control or flow apportionment based on link utilization seems excellently suited for combination with fixed routing (either deterministic routing or virtual circuit routing). With fixed single-path routing, there is no need for flow apportionment, but there is still a need to match the throughput sent over a path to the capacity of the path, i.e., to exert congestion control. With fixed multi-path routing, our scheme can be used not only for congestion control, but also for variable flow apportionment. Suppose, for example, that some network has three fixed paths between a given pair of nodes. Presumably, the paths are optimized for a certain peak period of flow between the nodes. During non-peak periods it may be possible to obtain better performance by using fewer of the paths (i.e., by using only the shorter paths). The flow apportionment procedure we proposed for use with the multi-path adaptive algorithm should be well-suited for this purpose.

What about the adaptive multi-path throughput-oriented routing algorithm itself? Does it have the property needed for it to work well with the congestion control and flow apportionment procedures that have been integrated into it? Does it ensure that changes in the identity of a path are infrequent, relative to changes in the residual capacity of an unchanging path? It seems that it does, or at least can be made to do so, with proper engineering and tuning. If we ignore topological changes for the moment, the only events which can cause path changes are reported (in routing updates) changes in the bottleneck type of a link. It does seem that such changes can be made to be much less frequent than simple changes in the residual capacity of a link. To see this, let us look at the various scenarios under which the bottleneck type of a link can change, and see what effect such changes can have on the identities of the paths between a given pair of nodes.

1) For our first scenario of bottleneck type change, let us suppose that link l, which was previously underloaded, is now fully loaded with traffic from sub-networks $G_1,\ldots,G_i$. That is, link l has changed from not being a bottleneck at all to being a type i bottleneck. As a result, link l will be removed from all sub-networks of index i+1 or greater. However, ex hypothesi, link l has been carrying no traffic from any of those sub-networks. So in sub-networks of index greater than i, there

may be a change of identity in the path between a certain pair of nodes, but only if there is no traffic between that pair of nodes. That is, any new path which forms between nodes S and D in sub-network j, j>i, will be a path consisting only of links which previously carried no flow at all from S to D in sub-network j. Therefore, MAXFLOW can be correctly and accurately computed on such paths simply by taking a fraction of the residual capacity of the path, without having to add in any previously measured flow. In sub-networks of index less than or equal to i, however, there is no change in path identity at all, but only a change in residual capacity (from positive to non-positive). Hence, MAXFLOW can be correctly and accurately computed according to the formula given in section 4.6.

2) For our second scenario of bottleneck type change, let us suppose that link l, which was previously underloaded, is now overloaded. It is fully loaded with traffic from sub-networks $G_1,\ldots,G_i$, but also carries traffic in sub-network G_j, j>i, for an overload. Note that, even though the link is fully loaded with type i traffic, our definitions make it a type j bottleneck. The present scenario reduces, therefore, to the previous one, and MAXFLOW can be accurately computed according to the formula in section 4.6. If, on the other hand, we had decided to declare link l to be a type i bottleneck, it would be removed from a sub-network in which it is carrying traffic, namely G_j. That

would give rise to the inaccuracy in the computation of MAXFLOW that we have been discussing, and hence is best avoided. If the link were declared to be a type i bottleneck rather than a type j bottleneck, all type j flow on the link would cease. In effect, the type j flow would be bumped by the type i flow. This is not, however, desirable. Flow types are not priorities, and ought not to be confused with priorities. Declaring the link to be a type j bottleneck will allow the type j flow to remain, though of course the type i and the type j flow will be scaled down when the network nodes receive a routing update reporting that link l has negative residual capacity.

3) For our third scenario of bottleneck type change, we suppose that link l was previously fully loaded with traffic in $G_1,\ldots,G_i$, and now remains fully loaded, but carries traffic in a sub-network G_j, $j>i$. It may seem as though this scenario is impossible. After all, if the link was previously a type i bottleneck, and $j>i$, the link is not used in G_j, and hence cannot carry any type j traffic. However, there is a race condition to consider. Consider a link l which emanates from node N, and consider three successive measurement intervals. At the end of the first interval it is underloaded. At the end of the second interval, it is a type i bottleneck, and node N generates a routing update u reporting l's change in bottleneck type. Once the update u is processed by node N, link l will be removed from

G_j, after which N will not carry any type j traffic over l. (Note that even during a transient period when N has processed u but other nodes have not yet done so, there will be no type j traffic sent over l, since each node makes an independent determination as to which packets to route over which of its lines.) However, there will generally be some lag time between the moment when N generates update u and the moment when its processing of u is complete. This lag time will occur at the beginning of the third measurement interval. During the lag time, it is possible for node N to send type j packets over link l. This may result in l's becoming a type j bottleneck. Since l was previously a type i bottleneck, making it a type j bottleneck will cause it to be reinserted in sub-networks $G_{i+1},\ldots,G_j$. It should be clear that the presence of this race condition can lead to an instability, where link l is alternately removed from and then reinserted into G_j after successive measurement intervals. This instability could result in inaccurate computation of MAXFLOW in G_j, and these inaccuracies might propagate over a long period of time. Therefore, it is very important to prevent this race condition from occurring. This is not difficult to do; in fact, we can think of three different methods of eliminating this race condition:

a) Through the proper assignment of CPU priorities to the measurement, update generation, and path-computation

processes, one can make it impossible for any packet forwarding (i.e., routing table look-ups) to be done between the time a measurement interval ends and the time the node generating the update (if an update is generated) completes its processing of that update.

b) If an update is generated at the end of a measurement interval, the start of the next interval can be delayed until the node generating the update has fully processed it.

c) Once a node has generated an update declaring a link to be a type i bottleneck, its measurement process can ignore the presence of any type j packets, knowing that the presence of the packets is due to the race condition.

Which of these three methods ought to be used depends largely on implementation considerations; any would do the job adequately. If we can assume that one of these methods will be implemented, we may regard the scenario under discussion as impossible, and hence as not contributing either to instability or to the inaccurate computation of MAXFLOW.

4) In our fourth scenario of bottleneck type change, a link which was formerly fully loaded with traffic from sub-networks

$G_1,\ldots,G_j$ is still fully loaded, but only with traffic from sub-networks $G_1,\ldots G_i$, $i<j$. That is, its bottleneck type has decreased from j to i. As a result, the link will be removed from sub-networks $G_{i+1},\ldots,G_j$. However, as in scenarios 1 and 2, the link is only removed from sub-networks in which it carries no traffic. Hence no inaccuracy is introduced into the computation of MAXFLOW, and no instability can result.

5) In our fifth and final scenario of bottleneck type change, a link which was formerly a bottleneck now ceases to be one, i.e., now becomes underloaded. Suppose link l was formerly fully loaded with traffic from sub-networks $G_1,\ldots G_i$, but that it is now underloaded. If the line is restored to all the sub-networks $G_{i+1},\ldots,G_n$ at the same time, inaccurate values of MAXFLOW may be computed for all these networks. Furthermore, all these sub-networks may try to route traffic over l, resulting in a sort of mad rush for the additional capacity. This issue is discussed in section 4.7, where we propose to continue reporting the link to be a type i bottleneck for a period of time after it actually ceases to be one. This will allow the link to be re-loaded, if possible, in those sub-networks in which it is already being used. If this is done, then there will have been no path identity changes, but only path residual capacity changes. If, after a period of time, the link has not become a type i bottleneck again, it can be reported to be a type i+1

bottleneck, whether it actually is or not. If it does not really become a type i+1 bottleneck, after a certain period of time, it can be reported to be a type i+2 bottleneck, etc,... If this procedure is followed, the link is revived in only one sub-network at a time, which limits the possible effects of inaccuracies in the computation of MAXFLOW. Each time the link is revived in a sub-network, enough time is allowed before reviving it in the next sub-network to allow it to become fully loaded in the former sub-network. As a result, the number of path identity changes should be much less than the number of path residual capacity changes.

We began by noting that our proposed flow apportionment procedure is suitable only if path identity changes are relatively infrequent when compared to path residual capacity changes. We then discussed the situations in which path identity changes can occur, and showed that the situation is either very infrequent (either because it is naturally infrequent, like topology changes, or because we explictly slow the reporting of the changes, possibly making the report unnecessary), or it cannot give rise to problems of stability or problems in the computation of MAXFLOW. We may therefore conclude that the multi-path throughput oriented routing algorithm possesses sufficient stability to enable it to interact well with the flow apportionment procedure. Of course, there are additional

questions that still need to be answered. Even though path identity changes may be infrequent, they do introduce inaccuracy into the flow apportionment procedure, and it would be interesting to determine just what effects this has, and how long it takes the system to re-converge to a more optimal routing pattern. This should be investigated by simulation.

4.9 Specification of the Routing Algorithm

Suppose a routing update is received which specifies that either the residual capacity or the bottleneck type of some link has changed. Then there may be a change in the set of paths used for a particular destination, or in the fractions P used to apportion the flow. In this section we describe the way in which a given source node must react upon receiving such an update. First, we define some notation.

Let F(A,i) be the father of node A in the shortest-path tree of sub-network G_i.

Let D(A,i) be the distance to node A along the tree of G_i.

Let C(A,i) be the residual capacity of the path to node A on the tree of G_i.

Let LINK(A,B) be a boolean which is true just in case A and B are neighboring nodes. Note that LINK(A,B) implies LINK(B,A). If A and B are neighbors, then AB will denote the link which carries traffic from A to B.

Let TREE(AB,i) be a boolean which is true just in case link AB is in the tree of G_i.

Let SUB(B,N,i) be a boolean which is true just in case N is in the sub-tree of B in G_i. Note that SUB(B,B,i) is identically true.

Let L(A,B,..) be the length of link AB in G_i. If AB is down, or if it is a type i-1 bottleneck, then L(A,B,i) is infinite. Otherwise, L(A,B,i) = 1. Note that if LINK(A,B) is false, L(A,B,i) is undefined for all i.

Let RC(A,B) be the residual capacity of link AB. This is undefined if LINK(A,B) is false.

When an update is received for link AB, we must see whether L(A,B,i) has changed for any i. A change in L(A,B,i) indicates either that link AB has gone down, has come up, or that its bottleneck type has changed. Then we must run a modified incremental SPF algorithm for each sub-network G_i for which L(A,B,i) has changed. This modified SPF algorithm is quite similar to the ordinary incremental SPF algorithm that runs on the ARPANET. The only modification concerns the method of computation of the values of C(A,i). which are not used in the ARPANET.

The main data structure of the SPF algorithm is LIST. Each element of the LIST is an ordered quartet of the form <SON,FATHER,DISTANCE,CAPACITY>. Each such quartet represents a particular path to the node SON. The penultimate node on this

path is FATHER. The distance from the source node to SON is DISTANCE, and the unused capacity of that path is CAPACITY. If Q is a quartet, we use the notation SON(Q), FATHER(Q), DISTANCE(Q), and CAPACITY(Q) in the obvious way to refer to particular elements of the quartet Q.

We first define the function ADDLIST(Q) which takes as argument a quartet and performs LIST manipulation according to the following algorithm:

> If there is no quartet Q' on LIST such that SON(Q')=SON(Q), then Q is placed on LIST.
>
> If there is a quartet Q' on LIST such that SON(Q') = SON(Q), then if DISTANCE(Q') > DISTANCE(Q), Q' is removed from the LIST and Q is placed on it. Otherwise the LIST is left unchanged.

The purpose of this function is to ensure that only one path to a given node is on the LIST at any one time, and that it is the shortest path so far encountered.

Now the SPF algorithm itself:

1. If no tree exists, ADDLIST(<SOURCE,SOURCE,0,INF>), where SOURCE is the node in which the algorithm is running, and INF is a representation of infinity. Go to step 5.

2. If the change in length was to link AB, then

   a. If TREE(AB,i), then set

      DELTA = L(A,B,i) - the old value of L(A,B,i)

   b. If not TREE(AB,i), then set

      DELTA = D(A,i) + L(A,B,i) - D(B,i)

      If DELTA $\geq$ 0, the algorithm terminates.

3. For all N such that SUB(B,N,i), set D(N,i)=D(N,i)+DELTA

4. If DELTA > 0, then

   for each N such that SUB(B,N,i),

      for each K such that LINK(K,N) AND NOT SUB(B,K,i)

         if D(N,i) > D(K,i) + L(K,N,i)

            then

         ADDLIST(<N,K,D(K,i)+L(K,N,i),min(C(K,i),RC(K,N))>)

   If DELTA $\leq$ 0, then for each N such that SUB(B,N,i),

      for each K such that LINK(N,K) AND NOT SUB(B,K,i)

         if D(K,i) > D(N,i) + L(N,K,i)

            then

         ADDLIST(<K,N,D(N,i)+L(K,N,i),min(C(N,i),RC(K,N))>)

5. If the LIST is empty, the algorithm terminates. Otherwise, remove from the LIST the quartet Q with smallest DISTANCE. Place SON(Q) on the tree so that its father is FATHER(Q). (Exception: if SON(Q) = SOURCE, place SOURCE on tree as its root.)

Set D(SON(Q),i) = DISTANCE(Q)

C(SON(Q),i) = CAPACITY(Q)

5. For each neighbor N of SON(Q),

   a. If N is already on the tree, and
   D(N,i) $\leq$ D(SON(Q),i) + L(SON(Q),N,i),
   then do nothing, and continue with the next neighbor.

   b. If N is already on the tree, but
   D(N,i) > D(SON(Q),i) + L(SON(Q),N,i),
   then remove N from the tree and go to step c.

   c. ADDLIST(<N,SON(Q),D(SON(Q),i) + L(SON(Q),N,i),
   min(C(SON(Q),i), RC(SON(Q),N>)

7. Go to 5.

After running the SPF algorithm for each sub-network where it is necessary, we must check to see whether that update reported a change in the value of RC(A,B). If so, then for each sub-network G, such that TREE(AB,i) the following Capacity Adjustment Algorithm (Algorithm CAA) must be run:

If the new value of RC(A,B) is less than the old value, then for all nodes N such that SUB(B,N,i), set C(N,i)=min(C(N,i), RC(A,B))

If the new value of RC(A,B) is greater than the old value, then

1) set C(B,i) = min(C(A,i), RC(A,B))
2) Put B in a FIFO stack
3) If the stack is empty, the algorithm terminates. Otherwise, remove the first element from the stack, and call it "N"
4) For each node K such that LINK(N,K) and SUB(B,K,i),
   a) set C(K,i)= min(C(N,i), RC(N,K))
   b) Put K on the stack
5) Go to 3.

It is worth pointing out the reason why algorithm CAA is so much simpler if the residual capacity of a link decreases than if it increases. If RC(A,B) decreases, then we have two cases to consider. Either AB is now the link of least residual capacity on the path to a destination D, or not. If so, the new value of C(D,i) is equal to RC(A,B). If not, then the value of C(D,i) remains unchanged. If RC(A,B) increases, we have two different cases to consider. Either AB was not previously the link of least capacity on the path to D, or it was. In the former case, C(D,i) remains unchanged. In the latter case, some other link may now be the link of least capacity on the path to D. In order to determine the new value of C(D,i) we must determine which link

this is, which is the cause of the additional complexity of the algorithm.

Note that, in general, a single update packet will contain updates for several links, viz. all the links which emanate from the node which sent the update. Although several link updates may be blocked together for the purposes of the updating protocol, for the purposes of doing the routing computation each link update must be considered as a separate and independent update from any other link updates which may happen to be in the same packet. The SPF and CAA algorithms must both be run, if necessary, for the first link update before any processing of the second link update is done.

After the CAA and/or SPF algorithms have been run, the set of paths available to a given destination may have changed, and the maximum amount of flow which can be sent to destination D on the tree of G_i may have changed. Thus the values of MAXFLOW and the fractions P will have to be recomputed, as described in the previous section. It is possible that due to these recomputations, one will find that more flow has already been sent to a particular destination in a particular sub-network than is allowed. (In general, the arrival of updates will be asynchronous with respect to the boundaries of the measurement interval which a source node uses to measure and control its flow to the various destinations within the various sub-networks.) In

that case, one must not send any more flow on that path until the next interval begins. Conversely, the value of MAXFLOW to some destination in some sub-network may increase due to an update, even if the source node has already sent the (previous) maximum amount of flow during the current interval. In this case, the source can resume sending until the new value of MAXFLOW is exceeded. Of course, as soon as the fractions P are recomputed, the flow apportionment algorithm should begin to use the new fractions for apportioning the flow.

If there is no path to destination D in G_i, then D(D,i) will be infinite.

In general, a single routing update packet will contain entries for several links. Note that the CAA algorithm at the SPF algorithm must be run for each link individually, before proceeding to the next.

As we have discussed previously, it may be desirable to associate with each destination node the minimum bottleneck type of each path to it. This facilitates the selection of the next distinct path to the destination. Let BOT(AB) be the bottleneck type of link AB. Let MINBOT(D,i) be the minimum bottleneck type of the path to D in G_i. Now suppose an update is received which indicates that BOT(AB) has changed from i to j. Then, for each sub-network k in which L(A,B,k) is now finite, one must run an

algorithm which is obtained from algorithm CAA by substituting BOT for RC and MINBOT for C. In addition, the SPF algorithm must be modified so that the LIST elements are quintets, whose fifth element is the minimum bottleneck type of the path represented by the quintet. These values would then be manipulated in exactly the same way as the values representing residual capacity.

It may seem that there is quite a lot of computational work to do each time an update is received. However, it must be remembered that the SPF algorithm is quite efficient computationally, in that it does a lot of work only if there is a large routing change to be made, which will happen only if the link for which the update is received has a large sub-tree in the shortest-path tree. In a highly connected network, the shortest-path tree will generally be quite shallow, and the average sub-tree size will be very small (in a 40-node network with a connectivity of 4, the average sub-tree size is only about 0.5), therefore, the amount of computational work needed to process each update will also be quite small.

4.10 The Sub-Optimality of Incremental Changes

An important feature of our proposed routing algorithm is that changes in the loadings of the various links cause only incremental changes in the routing of the flows. As the network's residual capacity increases or decreases, traffic is diverted incrementally so as to make best use of the network's capacity. However, there is no wholesale global rearrangement of the flows, and this can cause sub-optimality in some situations.

This point can be illustrated by means of an example. Consider a 10-node ring network with contiguous nodes numbered consecutively. Suppose that each link has a capacity of 10 units of flow, and that node 2 is sending 10 units of flow to node 1 on the path 2-1. Now if node 3 should desire to send 10 units of flow to node 1 also, our routing scheme will force it to use the path 3-4-5-6-7-8-9-10-1 (assuming no other flows). This routing is optimal for the offered flows. Should the flow from 2 cease, our routing scheme would divert all the flow from 3 to the path 3-2-1, again achieving optimality. Suppose, however, that after this happens, the flow from 2 resumes. Our routing scheme will put it all on the path 2-3-4-5-6-7-8-9-10-1, which is extremely sub-optimal. That is, our routing scheme picks the best path for a new flow, assuming that the other flows remain fixed. It will not, however, rearrange the already existing flows. To achieve optimality, the flow from 3 must be diverted from the shorter

path to the longer, so that the flow from 2 can use the shorter path.

A possible approach to this problem might be the following. Suppose that, instead of computing RESCAP, and hence MAXFLOW, on the basis of residual capacity (RC), we compute it on the basis of another quantity, RC', where

$$RC'(A,B) = RC(A,B) - k*(HOPS(S,B) - 1)$$

k being a constant (say), S being the node doing the computation, A and B being neighboring nodes, and HOPS(S,B) being the number of hops on the path between S and B. Now, the further away a particular link is, the less capacity it appears to have. Consider how this would affect the example above. Suppose k = 5 units of flow, and that we allow a link to handle 80 flow units before becoming a bottleneck. Suppose also that at some given time, link 2-1 and link 3-2 are empty, and node 3 wishes to send 80 units of traffic to node 1. RC(3,1) will be 80, but RC'(3,1) will be only 75. Hence node 3 can send only 75 units of traffic on the path 3-2-1, and will have to send the other 5 units on the other path. Now if node 2 has 80 units of traffic to send to node 1, it will be able to send 5 units on path 2-1 (since it sees link 2-1 as having 80 units of capacity, only 75 of which are in use), and will have to send 75 units on the longer path. If link 2-1 is carrying 75 units of the flow from 3 and 5 units

of the flow from 2, node 3 will have an RC' value of -5, which will force it to reduce its traffic on the path 3-2-1 by 5 units. This will allow node 2 to send an additional 5 units of traffic. Eventually, the flow 2-1 will be able to force the flow 3-1 off the link 2-1 entirely, forcing it to take its alternate route, and thereby converging to optimal routing (eventually!).

Unfortunately, there are several serious problems with this scheme. Note that, with this scheme, node 3 will never be able to send 80 units of flow to node 1; the scheme will only allow a full 80 units of flow to be sent between nodes which are immediate neighbors. Furthermore, if link 3-4 breaks, node 3 may not be able to send any flow to node 1; node 2 can lock out the flow from node 3. Another problem is that the scheme is wasteful of space, since it can force capacity to be unused whenever there are a significant number of multi-hop flows. This is not a good property for a routing algorithm which is expected to maximize throughput.

Clearly, this scheme needs two additional features before it can become acceptable. It needs a notion of fairness, so that all flows will be allowed some capacity, even if they must travel many hops. Also, it needs some sort of triggering mechanism, so that it is not called into play unless it is actually needed. At present, these are still unsolved problems.

4.11 The Delay Issue

We have not taken account at all of the issue of network delay. Our proposed routing algorithm has as its goal the maximization of throughput, with no considerations of delay. We realize though that what is really needed is an algorithm for maximizing throughput subject to the constraint that the delay or some function of the delay be kept below a certain threshold. We expect that the delay constraint can be met simply by setting the bottleneck threshold appropriately. That is, if delays are too high, the bottleneck threshold can be lowered. This would lower the total utilization of the network, hence decreasing the delay. We expect that the threshold can be tuned to give an acceptable value of delay, so that no real-time delay measurements will be needed. If this turned out not to be the case, link delays could be measured dynamically, and the bottleneck threshold could be altered in real-time to reflect changing delays. This would affect only the update generation process, leaving all the rest of our proposed routing algorithm unaffected.

5. CONGESTION CONTROL METRICS AND THEIR APPLICATION TO RAFT

Computer networks have had the luxury of technical evolution during a decade of relatively modest user demand. This fact has allowed network engineers the luxury of concentrating their efforts on the design of the basic packet-switching service as opposed to the design of special procedures intended to assure adequate performance during protracted periods of high resource utilization. As the ratio of users to available resources grows, however, this situation can be expected to change. That is, as average resource utilization increases, the possibility arises in any network that a set of resources will become congested to the extent that network performance will be affected.

In order to clarify this point, consider the difference between a network in which the average resource utilization is 10% and a network in which it is 70%. Both of these networks have excess capacity. Despite this fact, it is highly probable that stochastic fluctuations in offered traffic will frequently produce very long queues in the network whose average utilization is 70%. These queues may result in unacceptable delays; i. the queues grow so long as to exhaust the buffer space in some nodes, packets may have to be discarded and retransmitted, thereby lengthening the observed delays and wasting resources. Throughput will, therefore, also suffer. While congestion can (and will) occur in a network where average resource utilization

is 10%, the frequency of its occurrence will be much less and the scope of the problem not nearly so severe.

The problems arising from increased network demand must be addressed in several ways. First and foremost, of course, one must have sufficient resources in the network. Secondly, network resources must be utilized intelligently. In this context, "intelligent use" is that which maximizes the probability that performance criteria are, on the average, met. Finally, there must be several procedures designed to ensure reliable service in the face of the enhanced possibility of resource congestion and consequent degraded service. That is, there must be algorithms within the subnetwork which are specifically intended either to prevent resource congestion from occurring or to respond in a salutary manner to such congestion. There has therefore been increased attention paid to the development of congestion control procedures for packet-switching networks. These procedures are designed to control the flow of traffic into the network so as to prevent or eliminate resource congestion.

Since congestion control procedures control the rate at which traffic is allowed into a network, the designers of such procedures walk a fine line. If insufficient controls are applied, then congestion may not be eliminated; if controls are applied too liberally, then too much traffic may be kept out of the network. Unfortunately, given the complex nature of computer

networks and the fallibility of engineers, not all congestion control algorithms will do what is claimed for them, and this introduces the possibility for disaster. The conclusion is that the development of congestion control procedures requires the associated development of a methodology for the evaluation of those procedures.

In the following, we will attempt to define precisely what a congestion control procedure is. We also define criteria, or metrics, against which to measure the performance of specific algorithms. Finally, we will apply these metrics to evaluate the algorithm which has been proposed to effect congestion control in AUTODIN II.

5.1 Definition of Congestion Control

While congestion control procedures are designed to manage resources during periods of high demand, not all procedures so designed are properly classified as congestion control procedures. In particular, both flow control algorithms and routing algorithms might also be designed so as to manage resources in order to produce an acceptable response during periods of heavy utilization. For this reason, it is easy to confuse the respective functions of routing, flow control and congestion control. In the case of flow control and congestion control this confusion is rampant. We propose therefore to

discuss congestion control by describing how it differs from routing and flow control.

Flow control refers to a set of algorithms which operate in the communications subnetwork in order to maximize the traffic that can be passed from a source to a destination. In this context, source and destination may refer to either a pair of hosts or a pair of exit and entry nodes. A classic situation requiring flow control is that in which a high speed host attempts to send packets to a low speed host. If the output rate of the source exceeds the ability of the destination to receive traffic, then traffic may back up into the network. Ultimately, significant amounts of traffic will be lost. The growth of queues and the need to retransmit packets will produce excessive delays. Since resources are used by packets that are eventually discarded, the total end-to-end throughput will be reduced. In order to remedy the situation, a flow control algorithm might be implemented to manage the rate at which traffic is offered by the source in such a way that throughput is maximized without exceeding the destination's capacity to receive data.

Adaptive routing algorithms typically operate in the subnetwork in order to redirect flows in such a manner that network performance, as measured by some objective function, is improved. For example, if a routing algorithm is intended to maximize throughput, then the algorithm will direct a flow along

the path that uses the smallest amount of network resources. If a single such path is not adequate, then the algorithm may split the flow among several paths. As the volume of the flow changes, then the paths may change too. In any event, a routing algorithm which maximizes throughput dynamically will allocate resources so as to allow the network to carry as much offered traffic as is possible.

Congestion control, unlike flow control, does not explicitly operate on an end-to-end basis and, unlike throughput-oriented adaptive routing, does not have the capacity to redirect specific flows in order to provide them with additional resources. Rather, congestion control refers to the collection of algorithms in a network which reduce traffic when some critical resource in the communications subnetwork is congested. Thus the function of congestion control is the inverse of the function of throughput-oriented routing. Whereas the latter may increase resources used by a given traffic flow in order to allow an increase in the flow's volume, congestion control reduces volume in order to reduce the demand of a flow for one or more resources.

It is misleading to attempt to rank routing, flow control and congestion control in order of importance or to imagine that there is a definite chronological order in which each algorithm is invoked by the network software. For example, it is sometimes

stated that flow control establishes and manages an end-to-end flow. Then, as the flow increases, routing is supposedly invoked in order to distribute the flow over the critical subnetwork resources. Finally, according to this scenario, congestion control is invoked when routing can no longer find additional resources over which to distribute the traffic. This scenario may be simple enough to indicate the functional differences among congestion control, flow control and routing. However, the true behavior of packet-switched networks is considerably more complex. Thus, the attempt to assign a chronology to routing and congestion control is a bit like assigning a chronology to the initial chicken and egg. For moderate to high utilizations both congestion control and throughput-oriented routing may operate simultaneously, the former attempting to reduce resource utilization by throttling traffic while the latter seeks additional resources for the offered flows. In general, it may be said that there are complex and subtle interactions among routing, congestion control and flow control. These interactions are represented schematically in Figure 5-1.

Because of the complex nature of these interactions, in the following material we will consider congestion control apart from its interactions with other network protocols.

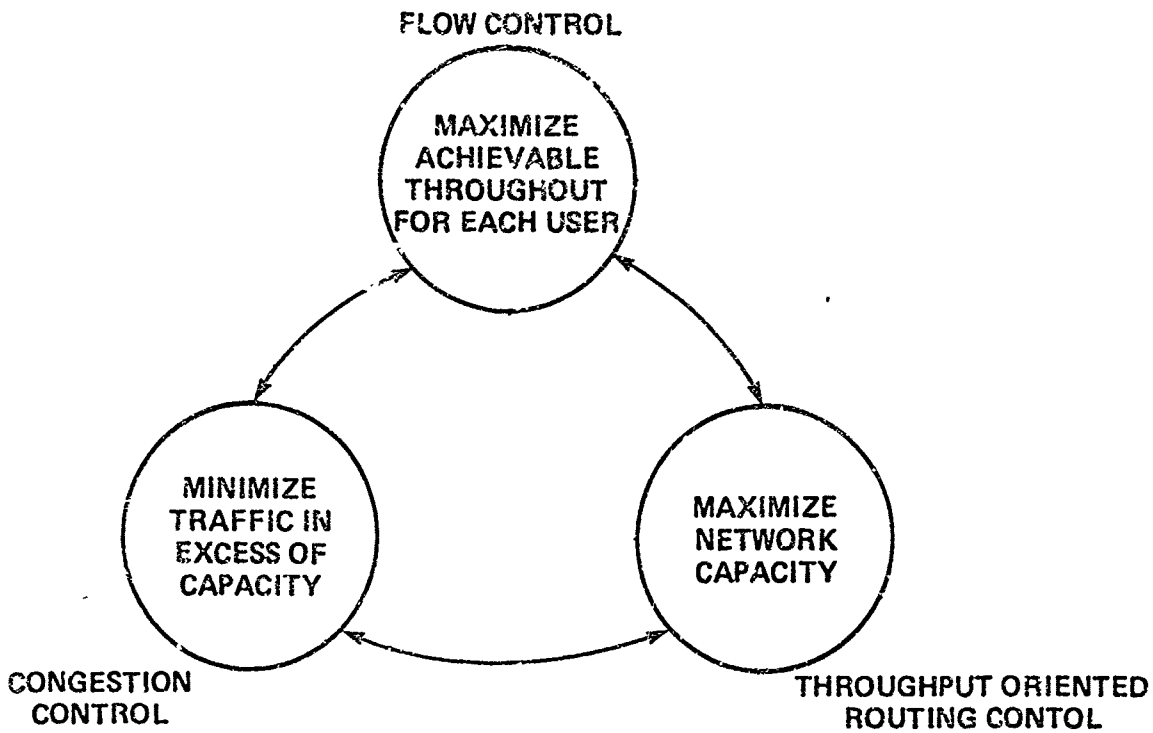


Figure 5-1 Network Control Interactions

5.2 Types of Congestion Control Algorithms

The primary conclusion of the previous section is that congestion control attempts to throttle traffic in order to prevent or alleviate resource congestion in the communications subnetwork. Within the boundaries established by this definition, there are any number of approaches by which congestion control may be effected. Despite these options, a large number of schemes have features in common. This, in turn, allows one to develop something of a taxonomy for congestion control.

One way of classifying congestion control algorithms is based upon whether they are designed to respond to congestion or whether they attempt to prevent congestion from arising in the first place. Algorithms which are based upon the former strategy we term curative. Curative algorithms are motivated by two related philosophies. First, any algorithm has an associated overhead. The algorithm introduces control traffic, consumes CPU cycles, requires memory for programs and tables. Curative algorithms seek to minimize this overhead during periods in which it is deemed not to be necessary. Additionally, since congestion control algorithms attempt to control the rate at which traffic is delivered to the network, there is always the possibility that an algorithm which anticipates congestion will throttle traffic needlessly. Curative algorithms refrain from controlling traffic

until congestion is detected. They are based upon a conservative attitude toward the application of controls and a liberal attitude toward the risk of congestion.

Algorithms which adopt a conservative attitude toward the risk of congestion by attempting to prevent its occurence, we term preventative. Such algorithms are clearly based upon the assumption that network performance is so seriously affected by congestion that it is appropriate to constantly carry whatever overhead is required in order to prevent congestion from arising. Proponents of such algorithms maintain that there is no real cost associated with overhead which is carried during periods of non-congestion since, by definition, the network has ample unused capacity during such periods.

Congestion control algorithms also may be classified according to the manner in which congestion is measured. Some algorithms, which we term implicit, do not directly attempt to measure resource utilization in order to detect congestion. Instead, they infer information about resource congestion indirectly from some global network state, such as the total number of packets in circulation. Other algorithms are based upon the view that congestion is a local phenomenon. (This does not, of course, preclude the possibility that congestion may simultaneously occur in a number of locations.) Such algorithms attempt to directly measure congestion by monitoring resource

utilization, queue lengths, etc. For this reason we call these algorithms "explicit".

It is possible to map a number of widely discussed congestion control algorithms into these categories (see Figure 5-2). For example, isarithmic algorithms place a limit on the total number of data packets which can be in the network at any given moment [1]. Simply put, these algorithms assume that congestion is a global phenomenon that can be prevented by limiting the total number of packets in circulation. The input buffer algorithm imposes a limit on the fraction of buffers in a node's buffer pool that input traffic can occupy [2]. These algorithms are preventive and explicit. The Delta-K algorithm, designed for AUTODIN II, measures queue lengths in the network and acts to throttle traffic if a queue exceeds a given length (see below for further details). Thus, Delta-K is explicit and curative. The congestion control procedure designed for Cyclades is similar to Delta-K [3]. Finally, the congestion control procedure described in Chapter 6, in which controls are, in a sense, always present but adjusted according to the measured state of each node and link, is preventive and explicit. It will be observed that we have had difficulty finding an entry in the curative/implicit box.

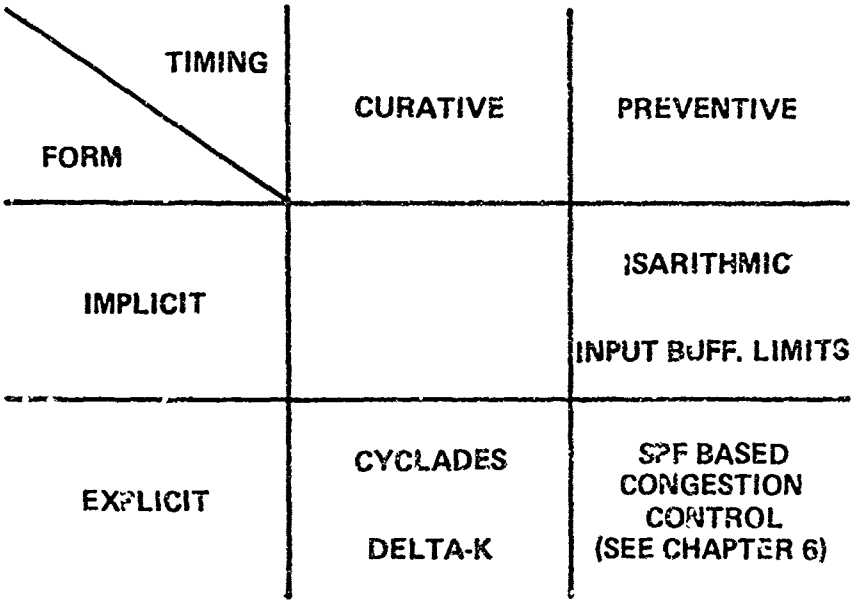


Figure 5-2 Types of Congestion Control Procedures

5.3 Congestion Control Metrics

No congestion control algorithm can be evaluated in the abstract. In determining the efficacy of an algorithm, one must clearly consider the characteristics of the specific network environment in which it resides. These characteristics include the other network procedures and protocols, the network topology and traffic profile, the specifics of the nodal software and such user-specified requirements as throughput, delay, and security. Despite the fact that congestion control algorithms reside in a variety of environments, a systematic approach to the evaluation of such algorithms is not precluded.

One can take a number of approaches to the evaluation of congestion control algorithms. Suppose one is given a particular network design and a traffic profile and is asked to examine the suitability of a particular congestion control procedure. A common approach is to simulate the network and to calculate the throughput, average packet delay, and packet loss rate for several multiples of the given traffic matrix. The numbers produced are taken to be measures of the congestion control procedure's performance. While this approach results in a nice "hard" answer, it suffers in several respects. First, one is not really given a detailed enough picture to understand one's results. If the performance of the algorithm was poor, one wants to know what about the algorithm resulted in poor performance.

Perhaps the congestion control algorithm was not at fault at all; perhaps routing and/or flow control caused the observed performance. Furthermore, one's assumptions about network design and traffic are frequently wrong. If the algorithm produced adequate performance under specific conditions, can one safely conclude that it should be implemented? Finally, throughput and delay do not completely describe all that a user may demand from congestion control; there may be other system requirements.

Thus, without discounting the value of throughput, delay, and data loss rate as measures of system performance, we also seek to develop other measures against which to evaluate congestion control procedures. It is our hope that a methodology which is based upon these performance criteria will allow one to examine proposed procedures at a detailed level, will address the "why" behind system performance, and will accommodate a less restricted definition of system performance.

While it would appear to be desirable to develop quantitative measures for system performance, most of the performance metrics which we are proposing are not inherently quantifiable. In that sense, they are to be understood as providing a methodology rather than a set of cookbook formulas for congestion control evaluation. In the following we will describe nine performance criteria. This list, although comprehensive, is by no means complete.

5.3.1 Sensitivity to Traffic Patterns

Congestion control techniques are generally developed with certain assumptions about the network traffic. Sometimes these assumptions are not explicitly stated. For example, consider a congestion control procedure which measures resource utilization at an intermediate node in order to determine whether some source node should be throttled. When congestion is detected, an update is generated and sent to a host which must subsequently reduce the rate at which it offers traffic. The update may be received seconds after the congestion was detected. For this procedure to make any sense, it must be assumed that the conditions which gave rise to the original congestion would have persisted had not some action been taken. In addition, it must be assumed that the consequences of the action can be predicted. In order for these assumptions to obtain, the network traffic must be described by a function which is reasonably well behaved. This assumption underlies many, if not all, curative procedures.

Frequently, even stronger assumptions are made by the designers of congestion control procedures. For example, it is common practice to assess the behavior of network protocols using some (analytic or simulation) model. It is very convenient for the developer of this type of model to assume that packet lengths and interarrival times are exponentially distributed. Assume that an engineer fine tunes a congestion control procedure using

such a model. Clearly, in doing so, he assumes that the exponential distributions used to model network traffic provide a reasonable approximation to that traffic.

Given the fact that a network designer always must assume something about the traffic on the network, the question emerges as to the sensitivity of the performance of the congestion control procedure to deviations from the assumed traffic profile. Can an isarithmic algorithm give acceptable performance on a heterogeneous network? Will an algorithm which gave good performance on a simulated network with exponentially distributed packet lengths give adequate performance on a real network in which packet lengths have a higher variance? Will a procedure which performs well under steady-state conditions be acceptable if the traffic profile has a marked time dependency? If a new application is added to a network, resulting in an immediate increase in traffic, and if there is a certain lag time before new resources can be added, can the algorithm manage traffic in a graceful manner?

The importance of these questions can be better understood if we consider the difference between a terminal-traffic-oriented computer network and a telephone network. The number of subscribers on telephone networks is so great that the law of large numbers produces a very regular and predictable traffic profile on all trunks. On the other hand, the number of computer

network users is orders of magnitude less. For example, the AUTODIN II TAC will allow a maximum of 256 terminal users; an ARPANET TIP will service a maximum of 63 terminal users. The number of TACs (or TIPs) is very small. Thus, for transaction-oriented computer networks one might expect flows which are highly unstable relative to telephone network traffic. (Of course, computer networks which handle a large amount of file transfer data can be expected to produce much more stable traffic.)

From the above we conclude that an appropriate metric against which to measure a congestion control algorithm is the degree to which the performance of that algorithm is affected by a changing traffic profile.

5.3.2 Effectiveness

Effectiveness measures the ability of an algorithm to achieve maximum reduction of congestion while minimizing the number of distinct flows (or users) to which controls are applied. An effective algorithm is one which seeks to reduce the rate at which traffic is allowed to enter the network only so much as to eliminate congestion.

There are a number of features which an effective congestion control technique must possess. First, the technique must be able to determine which network resource is, in fact, congested.

It must determine how much excess traffic there is and act to reduce traffic by that amount. This process is akin to a surgical procedure in which just the right amount of traffic is to be removed. A congestion control scheme that holds utilization too low may be as bad as one that allows utilization to be too high. The problem with allowing resource utilization to remain too high is that this can result in low effective throughput and high delays. But keeping utilization too low can have exactly the same effect!

Another aspect of effective congestion control is the ability to throttle only those flows which are causing the congestion. This may be a fairly complex task. Assume that a given flow A is causing a node to become congested. As a result, traffic from flows B and C which transit the congested node back up into neighboring nodes. Because of this, the neighboring nodes also congest. In the absence of the original flow A, no node would be congested. Upon observing the network, however, one finds several nodes congested with traffic flows A, B, and C. Clearly, one wants to throttle traffic A and not B and C. An effective congestion control scheme will throttle only those flows causing the congestion and not those flows which are being affected by the congestion.

Related to the requirement that an effective congestion control technique not throttle non-offending flows is the

requirement that it not excessively interfere with such flows. One way in which a congestion control technique can cause such interference is by creating excessive amounts of control traffic in the congested area of the network. An excessive amount of control traffic can cause nodes which neighbor a congested node to themselves congest. This can happen in two ways. Control traffic emanating from the congested node and forwarded to neighboring nodes can inflict an excessive processing burden on those nodes, causing them to congest. In addition, the requirement that a congested node produce control traffic might increase the utilization of the congested node so that it now has even more trouble accepting traffic from neighboring nodes. In both cases, the actions of the congestion control procedure have caused congestion to spread. In doing so, the procedure has interfered with traffic not implicated in the original congestion.

5.3.3 Fairness

Fairness measures the ability of a congestion control algorithm to apportion resources among users in such a way that all users receive equal treatment. This concept is, as stated, vague in that both "equal" and "user" are undefined. The definition of "equal" and of "user" are policy decisions which do not proceed primarily from technical considerations. The most general statement that one can make is that a "fair" set of

network procedures might guarantee users equal throughput or equal delay response.

Implementing fairness, as defined above, may be extremely difficult. In order to simplify our discussion of the problem, we assume that a congestion control procedure acts so as to allow fair access to critical resources. This means that when it is apparent that some particular resource cannot service all users who require its service, the resource is equitably apportioned among these users.

There are a number of alternatives from which a network planner might choose a definition of equitable apportionment. He might decide that resources should be divided equally among hosts. Thus, if there are ten hosts on the network, then each host would be entitled to one tenth of each critical resource. As an alternative strategy, the designer might decide that critical resources should be apportioned equally among terminal users. In this case, traffic from a host with 80 users would be guaranteed four times the resources as would traffic from a host with twenty users. As a third strategy one might choose to apportion critical resources in accordance with demand. In this case, a user (person, host, or source node) seeking to input traffic at rate 4r is apportioned four times the resources as is a user seeking to input at rate r. As another and simpler alternative, resources might be allocated evenly among source

nodes. As a variant on all of the above strategies, one might choose to apportion resources among flows (user-user, host-host, or node-node).

Assuming one has opted for one of the above alternatives, the definition of fairness is still incomplete. It remains to be decided whether or not resources are to be divided on a global or local basis. Assume, for example, that critical resources are to be divided fairly among users (however user is defined) and that a specific source node, whose resources are demanded by traffic from users A and B, congests. A local definition of fairness would require that the resources of the node in question be shared equally between users A and B. A global definition of fairness would require more information than the state of the congested node. For example, user B might have no other flows in the network other than that which is passing through the congested node, while user A might have a very large number of other flows. In this case, global fairness would require that A be throttled more than B.

Having defined just what fairness is, bringing it about may be quite a complex task. Assume that resources are to be allocated among hosts according to demand. This presumably means that each host is to be guaranteed a percentage of each critical resource in proportion to the amount of traffic it offers to the network. Unfortunately, this may be difficult to bring about.

The congestion control technique must be able to measure exactly what each user's demand for each resource is. The users and resources may be dispersed geographically. The demand from any particular user is subject to stochastic fluctuation. Even worse, the demand from some user may appear to be low as a result of some previous decision of the congestion control procedure. That is, if some user had been previously throttled, his measured input rate may appear to be low. But this result is an artifact of the congestion control procedure.

A factor which can further complicate a definition of fair resource allocation is priority. Some networks assign a priority level to each data packet. The intent of this assignment is that traffic of a given priority be given preference for resources over traffic of lower priorities. In designing a fair congestion control scheme one wants to ensure that higher priority traffic is throttled only if congestion cannot be alleviated by throttling only lower priority traffic. This may be difficult to accomplish. Assume that there are two priority levels, and suppose that some node congests. In order to accomplish our "surgical procedure" of congestion control, we presumably must separate the amount of congestion due to low priority traffic from the amount of congestion due to high priority traffic and throttle each separately.

It will be observed that there is a relationship between fairness and effectiveness. It will be recalled that one of the features of an effective algorithm is that only the offending flows be throttled. Exactly which flows are "offending" is something which clearly cannot be decided apart from the particular network's definition of fairness. For example, assume that a given node can handle 80 units of traffic and that it is offered 5 units from source A and 80 units from source B. In this case, we have a total of 5 units of offending flow. How those offending units are to be eliminated is something which can only be determined via fairness considerations. As a simple rule, one can say that effectiveness deals with the quantity of flow throttled while fairness has to do with the particular flows which are chosen for throttling.

5.3.4 Stability

Stability refers to a congestion control scheme's ability to offer steady state, smooth service to users under conditions in which the offered load to the network is smooth. This definition will be made more precise in what follows.

In general the burstiness with which packets are delivered to a destination is greater than the burstiness with which they are offered to the network. Assume there are packets which are being sent from a source A to a destination B and that these

packets are being delivered to the network at a constant rate. The variance of the interarrival times at the network is therefore zero. If one measures the variance of the interarrival times of these packets at destination B, one will probably measure a non-zero variance.

There is one primary reason for the increased variance of packet arrival times at B. Traffic belonging to the flow AB is contending for resources all along its path with traffic from many other flows. The volumes of these other flows are subject to stochastic fluctuation. Thus, all along their path, packets from AB will periodically be blocked while packets belonging to other flows are serviced. This causes some packets from AB to bunch up and increases the distance between other packets. Despite the fact that the sender has offered smooth input, the network appears to be delivering an erratic output stream.

It is not inconceivable that the performance of the congestion control algorithm will itself affect the smoothness of the service offered by the network. As an extreme case, consider a group of users who wish to offer traffic at a steady rate during a period of generally high network utilization. The congestion control algorithm, detecting congestion, acts to severely throttle traffic. The throttling is effective; the congestion is reduced. The congestion control technique now detects the absence of congestion and acts to reduce the controls

on the input traffic. This reproduces the original congested state. The process continues ad infinitum. In this case, the congestion control algorithm has responded to a situation in which user demand was constant, although heavy, and provided extremely erratic service as measured by the ability of the network to accept traffic and by the manner in which it delivers traffic. Such an algorithm would score low when evaluated for stability.

It is probably the case that no congestion control algorithm will be able to provide totally smooth service during extreme congestion. Thus, a measurement of an algorithm's stability must be done under varying network conditions. In general, a stable algorithm would induce no instability during periods of non-congestion and only a modest amount of instability during periods of congestion. During periods of extreme congestion, the technique may provide bursty service although this burstiness must be kept within bounds.

5.3.5 Responsiveness

Responsiveness measures the ability of a congestion control technique to track and control congestion. A "responsive" congestion control procedure has the ability to detect congestion early enough to contain it and the ability to detect the end of congestion early enough to prevent unnecessary reduction of user traffic.

A simple representation of the relationship between the true state of the network and the state of the network as perceived by the congestion control technique is displayed in Figure 5-3. The shaded area in the figure represents the error in the congestion control technique's view of the network state. This error is an inherent feature of any technique and is due to the fact that the procedure must first measure congestion, then distribute the measurement results, and then process those results. This entire sequence of operations takes time, and it is this response time that is shown in the shaded area in the picture.

The "ideal" congestion control procedure is clearly one in which the shaded area is minimized. Unfortunately, the minimization of the response time is not so simple a task. To begin with, the view of network congestion displayed in Figure 5-3 is an overly idealized representation. In the diagram, there is a single measure of network congestion, i.e., "traffic". But traffic, as opposed to its volume, is not a one-dimensional quantity. A given volume of traffic may or may not result in congestion depending upon how that volume is distributed. In addition, the level of congestion is, itself, not a one-dimensional concept. There is no one single measure of network congestion. In reality, congestion is a local concept and is measured by the utilization of a large number of geographically dispersed resources.

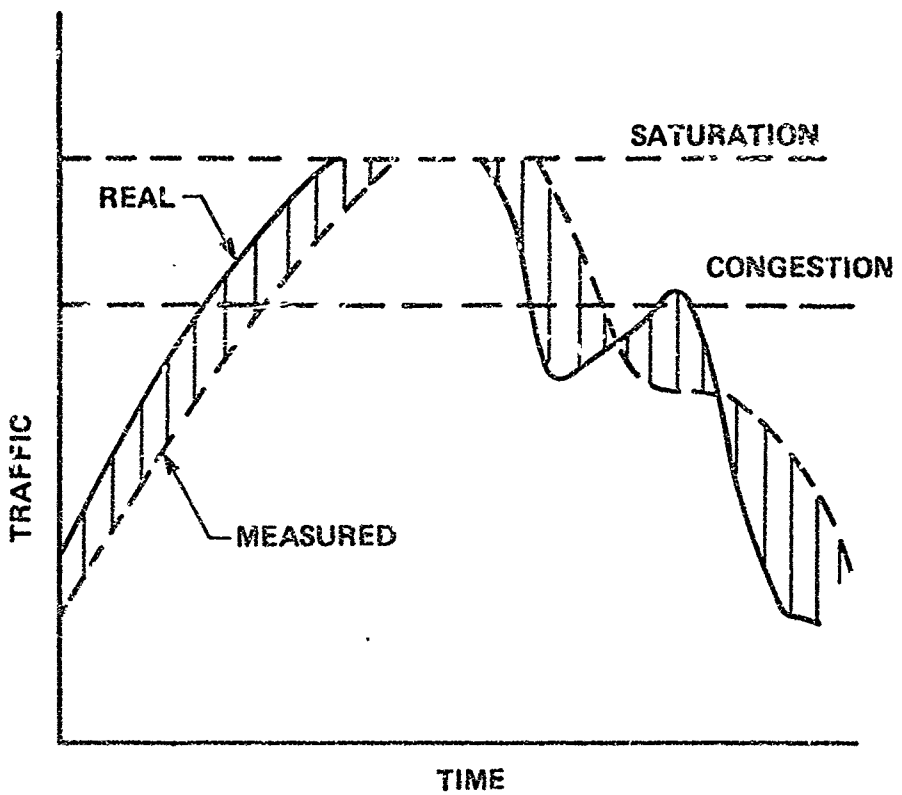


Figure 5-3 Congestion Control Responsiveness

Since different measurements of congestion are simultaneously being made in those geographically dispersed locations and since the results of these measurements reach different nodes at different times, it cannot necessarily be said that the congestion control procedure has a single, integrated consistent view of the state of the network. Node A, which detects congestion on one of its outgoing links, may send node B an update so that node B may throttle its sources. By the time node B receives the update, node A may have made another measurement and gotten some new result. Nodes A and B may never agree on what is happening at any particular moment.

If one thinks carefully about this example one can discern three components to responsiveness. Congestion must be detected quickly; news of its occurrence must be transmitted rapidly and reliably; controls must be applied promptly. Thus, responsiveness is a function of the promptness of the measurement, updating and metering functions of the congestion control procedure.

There is a complication involved in the attempt to respond rapidly to congestion. Since resource utilization is subject to continual stochastic fluctuation, there will be short periods during which resource utilization is quite high but the resource uncongested. As an extreme example, consider a network link. Assume that the link is defined to be congested if its average

utilization is 80%. Whenever a packet is in flight over the link the line utilization, averaged over the transmission time, is 100%. Yet, one certainly does not want to throttle traffic each time a packet is transmitted. On the other hand, one does not want to measure average utilization over, say, an hour, in order to see if a line is congested. One wants to respond rapidly to congestion but one wants first to be sure that congestion has occurred.

5.3.6 Overhead

Overhead is a measure of the quantity of control traffic required to operate the congestion control technique and the amount of network resources required to process and transmit this traffic. A procedure which requires a large amount of overhead runs the risk of reducing the capacity of the network to carry data. Of all the performance criteria which we are proposing, overhead is probably the most directly quantifiable.

One may divide the overhead imposed by a congestion control procedure into a number of components. First, any congestion control procedure will require processing resources, probably in every node. The amount of resources required is a function of the computational complexity of the algorithm and the frequency with which the algorithm must be invoked. A second aspect of a congestion control procedure's overhead is its requirement for

memory, once again in each node. There are two components to a procedure's memory requirements: program space and table space. The latter may impose an important restriction on how congestion control is to be implemented. Clearly if controls are to be applied separately to each user-to-user flow, more table space will be required than if controls are to be applied on a node-to-node basis.

A third component of a congestion control procedure's overhead is the bandwidth required in order to transport control packets. Both nodes and lines have a finite capacity to carry information. Frequently, the network software responsible for forwarding packets will make no distinction between packets which contain control information and packets which contain user data. Thus, control packets can be seen as contending directly with data packets for each node and each line. As the number of control packets increases, the ability of the node to process data decreases.

A fourth and extremely important component of a congestion control procedure's overhead might be called induced suboptimality. In order to understand this concept consider a network without congestion control. If one were to plot the effective throughput vs. the average resource utilization for this network, one would find that throughput peaks at some value of the utilization which is less than 100%. This phenomenon is a

consequence of the fact that, at very high average utilizations, stochastic fluctuations in queue lengths will result in the loss of data. All resources used by data which are eventually discarded must be considered to have been wasted. The capacity of a network to carry traffic is inversely related to the extent to which resources are wasted.

One way of looking at congestion control procedures is as acting to throttle traffic at exactly the point at which the short term utilization has become much greater than the average utilization. The probability distribution of queue lengths in a network without congestion control can be seen as having a long tail. With congestion control, this tail becomes much shorter. The risk one runs in operating a network at high utilization without congestion control is that queues will grow so long that data will be lost. Therefore, the major benefit of congestion control is that it allows one to operate a network with a higher average resource utilization.

In theory, one would like to be able to achieve 100% resource utilization. Unfortunately, no procedure is perfect. Thus, if one were to plot effective throughput versus average resource utilization for a network with congestion control, the curve would presumably peak at some value of the utilization which is still less than 100%. (The benefit of the congestion control procedure will be seen in the fact that average resource

utilization and maximum throughput will be higher than if there had been no congestion control.) By "induced suboptimality", we mean the difference between 100% and the utilization which yields the maximum throughput under the congestion control procedure.

An analogy for the notion of induced suboptimality may be found in the attempts of young children to carry a glass of water. If the goal is not to lose any water while transporting the glass, then the amount of water which should be transported is a function of the child's skill. An unskilled child may only able to carry a glass which is half full; a more dextrous child can carry glass in which the water level is close to the rim. It is only the rarest of human beings who will be able to carry a completely full glass of water.

The importance of overhead lies in the fact that each of the above overhead elements robs the network of resources which might have been used to process and transport user packets. A congestion control technique therefore comes with a practical pricetag. It is important, however, to distinguish between a technique's costly overhead and its free overhead. If a network is designed to operate at an overall resource utilization of 50% and the actual offered data traffic requires an average resource utilization of 40% then the congestion control technique has no cost if overhead represents less than 10% of all resources. If the overhead exceeds 10%, then additional resources will have to be purchased. In this case, overhead has cost.

5.3.7 Robustness

Robustness measures the capacity of a congestion control procedure to successfully handle a variety of network conditions. There are a number of reasons to require that a congestion control procedure be robust. One important reason is related to the fact that an actual network will differ substantially from its early design. A congestion control procedure may be developed in parallel with its network's design. This forces the procedure's designer to make assumptions about the network topology, the nodal architecture, the processing power and memory in the node, the structure of the nodal software and the other network protocols. Many of these assumptions will prove to be wrong in the initial network implementation or will become wrong as the network changes. The congestion control procedure must either provide satisfactory performance in the face of these changes or be sufficiently parametrized so that it may be fine-tuned should performance begin to suffer.

Even if a congestion control procedure's designer has made assumptions which are largely valid, the model of the network which he uses inevitably represents a simplification. The actual behavior of any network will be much more subtle and complex than the picture of that behavior is some engineer's mind; it will be more complex than any simulator. Congestion control procedures are designed by engineers and tested on simulators but they must

operate in the real world where anything can happen. This requires a sort of durability which we call robustness.

There is a special and important example of the manner in which a network's environment can differ from the assumptions made about it by the designer of a congestion control procedure. When a procedure is designed, it is frequently assumed that it is to reside in a benign environment. A benign environment is one in which all of the participants obey all of the designer's (implicit or explicit) rules. By making this assumption, however, a protocol designer introduces the risk that the efficacy of a procedure can be compromised if a user chooses to act maliciously by subverting the host interface, or if a program error produces the same effect. For example, there have been several instances of software bugs which have led to runaway hosts. It is not difficult to imagine that a congestion control procedure, some of whose control functions are situated in the host software, can be undermined by modifying that software. If the design of a congestion control procedure has not anticipated these problems, poor performance can result from their occurrence.

From the above discussion, one can deduce the features that a robust congestion control procedure must possess. Such a procedure will not require that all of its assumptions be true in order for its operation to be satisfactory. In addition, a

robust procedure will possess a sufficient number of "hocks" so that it may be modified after its installation on the network.

5.3.8 Control Feedback Coupling

A congestion control technique performs a fundamental control theoretic task. There is a measurement function (to determine when congestion exists) which sends a feedback signal to a metering function (to reduce traffic). It is essential, therefore, that the technique draw the correct inference from a measurement in order to exercise the proper sort of control. Control feedback coupling expresses the extent to which this is the case.

It is inevitable that an error will be associated with a control mechanism which operates on secondary effects in order to infer the existence or non-existence of some primary event. For example, a congestion control technique must determine whether a specific resource is congested and then act in such a way as to reduce that congestion. Consider a node which consists of an input queue, a processor and an output queue. If the length of the output queue is less than n, packets will be processed, i.e., they will be removed from the input queue and placed on the output queue. Suppose that a congestion control technique infers information about processor utilization from the size of the input queue; if the queue is long, the processor is held to be

congested. Obviously, such a technique will frequently produce erroneous results. A long queue might indicate an overutilized CPU; but then it might also indicate an overutilized output line. The moral of this example is, of course, that a procedure which attempts to determine whether a CPU is overutilized should, if possible, measure CPU utilization.

Another way in which false inferences may be drawn by some control technique occurs when the technique's measurement function couples two unrelated events. For an example (not related to congestion control) consider a procedure which determines that a network link is down if 3 successive data packets are garbled during transmission. In this case, a line, no matter how noisy, cannot be declared down unless someone attempts to send data over it. Thus the up/down status of each line has been tied to the transmission of data packets. This type of connection violates control feedback coupling since the evaluation of network conditions is subject to the vicissitudes of user traffic.

Control feedback coupling is another example of a metric which is easy to understand but difficult to implement. Given the complex interactions in a network, it is conceivable that the measurement function implemented could be based upon multiple levels of false inferences. This could cause a large error in congestion measurement. However, the source of the error may be obscure.

5.4 Application of The Congestion Control Metrics to RAFT

The application of the congestion control metrics to specific algorithms is a problem which we have only begun to attack. Nevertheless, we believe that there are a number of general observations that can be made.

The use of the term "metric" to refer to the various performance criteria that we have discussed suggests that each criterion provides a yardstick against which one can quantitatively assess a congestion control procedure. To really merit the name "metric," the quantitative scale defined by the criterion should have a universal character to it. One should be able to ask, "What is algorithm A's score on the fairness scale?" and receive a numerical answer which, when compared to algorithm B's score, has an unambiguous interpretation.

This is, on the surface, an attractive concept. We believe, however, that the attraction is restricted to the surface. In order to see this, consider how one would go about defining a fairness or stability or effectivenes metric. One might invent and model a "standard" network and set of traffic matrices. If one were measuring stability, one might then determine the variance of packet delivery times under the tested congestion control procedure. One might average these variances over all "standard" traffic matrices and take the resulting number to be

the measure of the algorithm's stability. Unfortunately this result would, in general, offer no information about the stability of the algorithm when applied to another network or traffic profile. Statements which talk about the stability or the fairness or the effectiveness of an algorithm are not subject to universally unambiguous quantification. In fact, to be completely unambiguous, such statements should refer to a specific network environment and traffic profile. Algorithms which are effective under traffic matrix A may be totally ineffective under traffic matrix B.

The problem with the quest for quantification is that it frequently seeks to substitute a simple number for a detailed description of a complex reality. Consider how one goes about analyzing the suitability of a particular congestion control algorithm for a specific network environment. One generally thinks carefully about all of the features of the algorithm and considers how these features will interact with the characteristics of the network. One then attempts to determine a range of network conditions under which the algorithm might fail and a range of conditions under which it might succeed. If the algorithm has not been rejected at this point, one would then perform experiments, first using a simulation and then using the network itself, to see whether adequate performance will occur.

Suppose that one is investigating an algorithm for fairness. At the end of one's experiments one has, in a sense, a quantitative measure of fairness to the extent that the results of one's simulation experiments are quantitative. One does not have a quantitative measure of fairness in the sense implied by the sentence, "Algorithm A has a fairness rating of 7." One instead has a measure of fairness which has meaning only when given the specific experimental arrangement and an idea of how realistic the experimental arrangement is.

Thus, our "metrics" should be understood in a methodological sense rather than in a mechanical sense. We are proposing that a congestion control algorithm which does not yield adequate performance will invariably violate one of the criteria described above. Thus in analyzing a specific procedure, one should be guided by our performance criteria. In the following, we will provide an example of this process by analyzing the congestion control procedure proposed for AUTODIN II.

5.4.1 Background

AUTODIN II is a general purpose, common user packet-switching network for the DoD community. It is an extension of the ARPANET technology and is intended to serve operational military needs into the 1990s. Before evaluating the congestion control procedure proposed for AUTODIN II, it is

necessary to understand the motivation for a number of the procedure's features.

The two primary requirements for AUTODIN II which resulted in departures from the ARPANET design were the requirement that AUTODIN II be secure and that it operate at a higher utilization than the ARPANET. In order to effect security, each switch is protected by a security kernel. Unfortunately, the burden which the security kernel imposes upon the node CPU seriously reduces the ability of the switch to process packets. Thus, there is a severe conflict between the requirements for throughput and for security. The major implication for the subnetwork protocols of this dilemma is that the protocols must not make a difficult situation worse by imposing excessive overhead. The ratio of control packets to data packets and the computational complexity of the protocols are required to be small. The protocols which were designed to meet these requirements are collectively known as the Revised Acknowledgement and Flow Technique (RAFT) [4] [5] [6].

Congestion control is implemented in RAFT (see Figure 5-4) in the following manner. Associated with every trunk output queue are two parameters K' and K, K'<K. K is the maximum allowed queue size; K' is the point at which congestion is held to begin. The purpose of the algorithm is to reduce input at the source(s) as an output trunk's queue grows from K' to K.

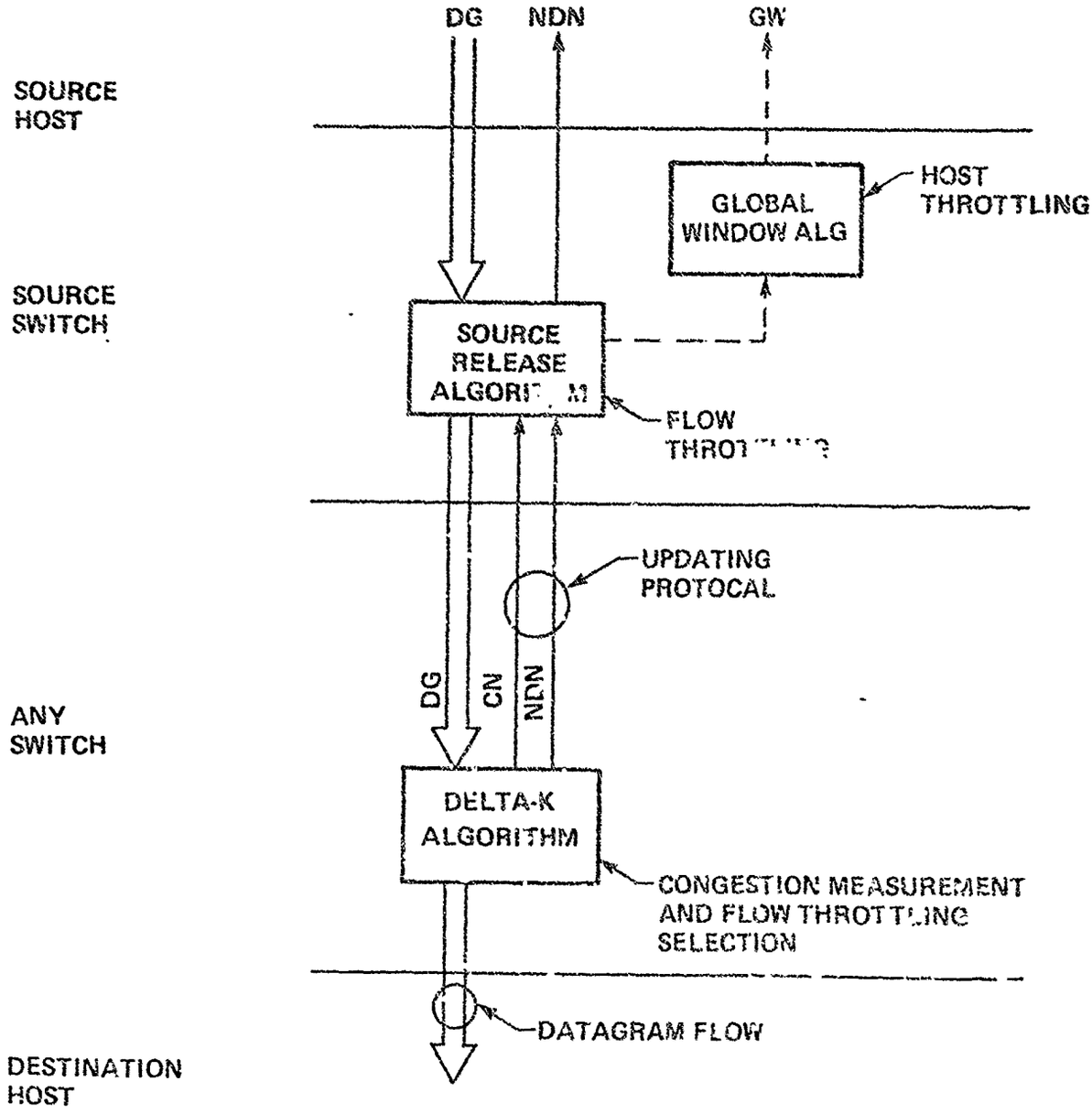


Figure 5-4 RAFT Congestion Control Algorithm

When a packet is received at a node and the routing computation places it on an output queue whose length exceeds K' but is less than K, a special "congestion notice" (CN) is sent to a source node so that traffic flow travelling over the link in question may be throttled. There are at least two procedures which have been proposed to determine which flow should be throttled and therefore which source should be notified. Assume that the difference between the queue length and the value K' is m. In the simplest scheme, the queue is scanned from the back forward and the first m different source addresses encountered receive congestion notices. In the more complex method the queue is scanned and the frequency of occurrence of each source address is determined. CNs are sent to the m most frequently occurring source addresses. In both cases, the flows (as opposed to hosts) which are throttled are chosen at random from the flows which belong to each source receiving a CN and which have packets queued on the congested link. This entire procedure is called the "Delta-K" algorithm.

Should a packet be received at a node and should the routing algorithm determine that it should be placed on an output queue whose length is K, then the packet is discarded and a "non-delivery notice" (NDN) is sent back to the source node. The NDN identifies the packet which was discarded (and is eventually passed back to the originating host which can then retransmit).

When a congestion notice or a non-delivery notice is received at the source node no additional packets from the identified flow are allowed beyond the source node for a fixed amount of time. Packets from a flow which is so throttled will be accepted by the source node until two such packets are queued on a "holding queue." If two packets are queued, then each additional packet from the flow is discarded at the source node and an NDN returned to the host. These procedures, which are implemented at the source node, are called "Source Release."

Host input is controlled via a procedure called the Global Window. Each host periodically receives from its source node a count. Each time the host submits a packet, the count is decremented. If the count has reached zero the host may submit no further packets until a new count is received from the source node. The size of this count, or window, is computed as a function of the total number of free input buffers in the source node and the number of packets in the holding queues belonging to all flows from the host whose window is being computed.

5.4.2 RAFT Congestion Control Evaluation

In the following material, we will present a critique of the RAFT congestion control procedure using the methodology described in the previous chapters. Although we are highly critical of a number of the procedure's features, it is important to note that

some of these features were necessitated by network requirements other than throughput. Thus, while we do propose various changes to the procedure, we do not guarantee that the implementation of these changes will not have a cost. That is, some modifications may result in the violation of some other performance criteria or of some specific AUTODIN II system requirement, particularly the requirement for security. While we will, at times, refer to the tradeoffs involved, these will not be discussed in depth inasmuch as our topic is neither security nor the performance of the security kernel.

5.4.2.1 No Buffer or CPU Utilization Measurement for Tandem Traffic

Problem

RAFT measures congestion at tandem switches by comparing the size of the queues for each outgoing line to the values K′ and K. Since only queues for lines are measured, this approach assumes that circuit bandwidth, as opposed to nodal memory or processing power, is the network bottleneck. This assumption occurs frequently in the literature on RAFT. For example, the analysis of RAFT performance presented in Reference [5] argues that the switch CPU can process packets at a rate which is high relative to the rate at which packets can be received by the circuits. The analysis also argues that the node contains so much buffer

space that it can be assumed that there are always a large number of free buffers.

All of these assumptions appear to have been invalidated by the subsequent development of the switch. Early tests of the AUTODIN II switch performance indicate that the CPU bandwidth for packets will be smaller than projected. (This problem is compounded by the demands which the NCC can make upon the node CPU when it requests any of a number of complex reports.) In addition, the program memory requirements are larger than had been assumed in the early design. At least some of this added memory for node storage was obtained at the expense of buffer space. To summarize, the RAFT congestion control design assumes that contention for lines will be the only source of congestion at tandem switches. Therefore, congestion is measured only at the queue for each line. Subsequent developments give reason to believe that limited buffer space and demand for CPU will also contribute to congestion.

A simple example can illustrate how RAFT's assumptions will, if false, affect network performance. Assume that the CPU in some node is a bottleneck. That is, assume that packets arrive at the node at a rate which exceeds the ability of the CPU to process packets and that the output lines are uncongested. Then the input queues will be very long and output queues will be very short. The Delta-K algorithm, which bases its estimate of

congestion upon the size of the output queues, will not generate congestion notices despite the fact that the node is congested. If the situation persists, the buffers in the node will be exhausted as more packets queue for processing. The node will no longer accept traffic from its neighbors which may, in turn, congest. Eventually, the output queues in adjacent nodes may grow and congestion will be detected.

Performance Criteria Violations

The fact that RAFT restricts its measurement of congestion to output queue lengths is one of its most serious deficiencies. The performance criteria violated are responsiveness, control feedback coupling, and robustness. Responsiveness is violated because the procedure will not rapidly respond to congestion under highly probable circumstances. In the example cited above, the output queues in the congested node will remain quite short so that the algorithm does not immediately detect the occurrence of congestion. It is only after the congestion has spread to the adjacent nodes that there is the possibility of reacting to it. Control Feedback Coupling is violated because the algorithm does not measure resource utilization in order to detect congestion. Instead, congestion is indirectly detected by measurements on queue lengths. Finally, the procedure is not robust because its performance is dependent upon the presence of inexhaustible buffer space and processing bandwidth. Neither of these

assumptions is true and, as a result, performance can be expected to suffer seriously.

Solutions

The obvious solution to the above stated problem is to monitor the demand for the CPU and for buffers. The demand for the processor can be measured by examining the size of the SCM Communications Processing Queue, which is the equivalent of the ARPANET's TASK queue. Another (and more powerful) approach to resource utilization measurement would involve the security kernel, which is the only process which knows the exact state of the buffer pool and of the CPU.

Any of these solutions will impose a greater overhead. However, it appears that the cost will be slight when compared to the obvious gains.

5.4.2.2 Delta-K Measurement Accuracy

Problem

The key measurement procedure in RAFT is the Delta-K algorithm which infers information about a circuit's utilization by measuring the number of packets queued for that circuit. A little reflection indicates that this procedure can produce highly erroneous results. As an example consider that RAFT will be likely to decide that a line is more congested if there are

ten 200-bit packets queued than if there are five 1000-bit packets queued. That is, RAFT will sometimes decide that the utilization of a line which transmits 2000 bits is greater than the utilization of a line which transmits 5000 bits, even if the lines have the same capacity.

The decision to count packets rather than bits was based upon security considerations. In order to prevent the possibility of establishing a covert storage channel, the untrusted software of RAFT cannot be privy to packet lengths. Thus, congestion control must either be implemented in the security kernel, or it must be ignorant of the lengths of packets. It was decided not to put the congestion control algorithm in the security kernel because trusted software requires formal verification. Since congestion control routines can be expected to require frequent modification, the necessity for repeated verification of this software was thought to be an excessive burden.

The fact that the congestion control procedure was only allowed to measure quantities of packets was not viewed as a serious deficiency because the original AUTODIN II specification indicated that bulk file transfers would dominate the network traffic profile. It was estimated, therefore, that 92% of all packets would be of maximum size. This being the case, it was reasonable to assume that, on average, packet count is a fairly

accurate measure of bit count (although not necessarily for all output queues).

Unfortunately, the projected traffic profile has since been modified and the expected percentage of full packets has been reduced to 66%. This has removed whatever justification there was for assuming that the number of packets queued for a line is a good measure of that line's utilization.

Performance Criteria Violations

Effectiveness is the performance criterion which is most seriously violated. Since in the best of circumstances, the Delta-K algorithm will only have an approximate sense of the state of any line, the amount of traffic which is throttled will invariably be wrong. In our discussion of effectiveness, we likened congestion control to a surgical procedure by which precisey the right amount of traffic is removed from the network. By measuring congestion as it does, Delta-K may be compared to a surgeon operating blindfolded; Delta-K never knows exactly how much traffic is present.

A second performance criterion which is violated is that of feedback coupling. The measurement of congestion is based upon a third order effect. The number of packets queued is held to imply the number of bits queued which is held to imply the state of the circuit for which those bits are queued. To know whether

a circuit is overutilized one should obtain a more accurate measure of link utilization.

A third performance criterion which can be violated is fairness. Since the source which is throttled is that with the most packets queued for a congested line, then a source which typically generates a large number of small packets will find itself throttled more frequently than a source generating full packets at a slower rate. This will happen despite the fact that the line bandwidth required by the slower source may be greater than that required by the source more frequently throttled. One would expect that sources which generate a large amount of interactive traffic will find themselves throttled by Delta-K a disproportionate amount of the time.

Finally, the algorithm as designed has proven to be virtually helpless in the face of changes in the traffic profile. Once again, the precondition which allows one to detect congestion by counting packets is that all packets be the same length. When the algorithm was designed, it was thought that most packets would be the same length. It is now believed that this will not be the case. The change in the expected probability distribution of packet lengths has severely undermined the expected performance of RAFT congestion control.

Solution

This particular problem provides a classic example of the conflict between security requirements and congestion control. There are two solutions. The first is to relax the security constraint and allow untrusted software to have access to packet lengths. The second is to assign responsibility for the measurement of congestion to the security kernel. From the standpoint of congestion control, the effect of both of these solutions is the same, i.e., to count bits rather than packets. The solutions differ in that the first solution requires a relaxation of the security constraints and the second requires an expansion of trusted software.

5.4.2.3 Lack of Smoothed Measurement

Problem

The measurement portion of the Delta-K algorithm makes its decisions based upon instantaneous measurements. Each time a packet arrives at an output queue, the length of the queue at that moment is checked to measure congestion. There is no attempt to compute an average queue length measured over some period of time. The use of instantaneous measurements is a poor procedure because the results of such measurements are subject to extreme stochastic fluctuations. There is always a finite probability that a large number of customers will be queued for

an extremely underutilized server, although this large queue will be short-lived. For example, any time a burst of more than K' packets arrives at a queue, Delta-K will throttle some host. This happens despite the fact that the queue may immediately empty out and remain empty for minutes. By relying on measurements of instantaneous queue lengths, the Delta-K algorithm may respond to "congestion" which is destined to dissipate quickly without external intervention.

Reliance upon instantaneous measurements is a feature common to many congestion control designs. The argument mustered for these designs is that congestion control is needed specifically to handle peak transients. According to this view, longer term congestion prevention is not the responsibility of congestion control but of routing and flow control. This argument can be responded to in several ways. First, even if it were true that congestion control is intended to react to short-lived peaks, this does not argue for a procedure which detects congestion from instantaneous measurements. It merely argues that the period over which observations are averaged be short. It is also incorrect that long-term congestion is the exclusive domain of routing and flow control. It may be the case that a large number of small flows have to traverse a single circuit and that no alternate paths are available. In this case, steady state congestion can arise which cannot be removed by routing since the

flows in question have but one route. Finally, if it is true that congestion control is intended to respond to short-lived peaks, then why have congestion control at all? That is, why act to eliminate something which will rapidly disappear by itself?

It should be pointed out that the absence of smoothed measurments in RAFT was, in part, motivated by security considerations. It was a security goal that routines implemented in untrusted software be memoryless so as to prevent the establishment of covert storage channels. Since averaging requires the storage of data for at least the duration of the averaging period, a congestion control procedure using smoothed measurements cannot be memoryless. Since the enlargement of trusted software to include a relatively volatile protocol is inconvenient, the use of smoothed measurements in Delta-K was precluded.

Performance Criteria Violations

The performance criterion which is most seriously violated by the use of instantaneous measurements is effectiveness. Since the algorithm reacts to short-lived transients, traffic may be unnecessarily throttled. This will result in reduced throughput. It should be pointed out that transients work in both directions. That is, it is conceivable that the output queue for a very busy link will be, for short periods, very small. In this case RAFT

may relax controls too soon, resulting in insufficient throttling, prolonged congestion and, therefore, reduced throughput.

The fact that the congestion measurement is being made only upon the arrival of a user packet results in violations of other performance criteria. Robustness is violated since the procedure does not contain sufficient parametrization. One can certainly adjust K and K'. One does not, however, have the ability to adjust the manner and frequency with which measurements of congestion are made in the same sense that smoothed measurements allow one to alter the time period over which the averaging is performed.

By measuring congestion only upon the arrival of a packet at an output queue, RAFT also violates control feedback coupling. Ideally, one should determine whether a resource is congested by continually monitoring the state of that resource. In RAFT, measurements are always conditioned upon the arrival of a data packet at the queue for an outgoing line. There is absolutely no basis for coupling these two phenomena.

Finally, stability also suffers under the Delta-K measurement procedure. Since instantaneous queue lengths are subject to constant stochastic fluctuation, the actions of RAFT congestion control, which are based upon those queue lengths, are

similarly subject to fluctuation. Users may find themselves throttled in a manner which seems arbitrarious.

Possible Solutions

The solution to the problem caused by instantaneous measurements is to change the RFNM procedure so that smoothed measurements of congestion are made. This modification requires a significant change to the present algorithm's implementation.

There are a number of ways in which smoothed measurements can be introduced. For example, the input queue length can be averaged over some small measurement period. This modification has the virtue of simplicity but retains the problems caused by detecting congestion solely by measuring the lengths of the output queues. A more complex modification of the RFNM procedure would result in measurements of CPU, buffer, and line utilizations, averaged over time. This modification might require security kernel involvement.

To avoid introducing new problems, the use of smoothed measurements requires careful tuning. Assume that there is a fixed time period over which resource utilizations are measured. Each time one of these measurement periods ends, average utilizations are computed and it is determined whether or not any resource is congested. If the measurement period is excessively long, then the procedure will not be able to react

congestion quickly enough. Thus, the performance criterion of responsiveness will be violated, the average utilization may be too high, and throughput may suffer. On the other hand, if the measurement period is excessively short, one reproduces all of the problems associated with instantaneous measurements.

Of course, the use of smoothed measurements requires a congestion control procedure with memory. If implemented in untrusted software, such a procedure violates security guidelines; if implemented in the security kernel, such a procedure cannot be easily modified

5.3.2.4 Selection of Offending Hosts

Problem

Delta-K, in its most complex design, selects the host that is to be throttled in the event of congestion in the following manner: if a packet arrives at a queue whose length exceeds K' by m, a RFNM is sent to the m sources which have the most packets in the queue. This procedure has the virtue of simplicity. Unfortunately, it also can result in a number of serious violations of performance criteria.

Begin with the number of packets from a flow A which are queued at some instant for a queue L is not necessarily a measure of the average demand for [illegible] is common to flows A

and B, each of which originates at a different source. Flow A may, on average, require 60% of L's bandwidth and Flow B 30%. Nevertheless, at any given instant, there is a finite probability that more packets from B will be queued and that Delta-K will therefore choose to throttle B.

It may be argued that A will be chosen for throttling with much higher probability than B and therefore the problem is not severe. Unfortunately, a number of RAFT features argue against this conclusion. First, recall that Delta-K counts packets and not bits when it examines queue lengths. Therefore, if B's packets are typically very short and A's packets are typically full, then B may be chosen for throttling more frequently than A despite the fact that A is using twice as much of the line's bandwidth. This was discussed in Section 5.4.2.2 above.

Even if A's packets and B's packets are of the same size, a number of not improbable scenarios can result in the frequent throttling of the non-offending flow B. Assume, for example, that traffic from A arrives in a regular manner at the queue for L so that there are typically 4 packets queued. Assume also that B sends traffic to L intermittently. Although B is dormant for long periods of time, it will periodically send packets in such a manner that, for short periods of time, it has an average of 8 packets queued. Then, for the periods during which both A and B are active, there are 12 packets queued for L. If the value of

K' is 11, then during such periods B will be throttled, despite the fact that its demands for L are much less than A's.

It may be said that B merits throttling. Congestion control is concerned with short-term congestion and, in the example cited, B is responsible for the short-term congestion. This argument is hard to justify; if B is active only for short periods, then it is probably more equitable for A to back off somewhat during those periods. Futhermore, B's bursty behavior may itself be an artifact of the RAFT procedure. Assume that B's users are constantly active, submitting traffic in such a way that the number of packets queued for L would be, on average, 2 in the absence of congestion control. Assume that, at some point, B is throttled so that traffic queues at its source host. When the congestion control timeout period ends, a flood of packets may arrive at the network, resulting in the throttling of B. This process can continue indefinitely.

It might be argued that for this unfortunate sequence of events to occur and reoccur hosts A and B must behave more or less synchronously and that such behavior is unlikely. Consider, however, that AUTODIN II hosts will, in all likelihood, be operating the same host-to-host protocols. This increases the probability that hosts will behave uniformly in response to some set of network conditions. Thus, one might expect periods of time during which host behavior is, in some sense, synchronized.

There is another not unlikely example in which the non-offending host will be throttled by RAFT. Suppose that buffer space is the critical resource, that K' = 14 and consider a node with two queues. On one queue there are 14 packets, all from Host A. On the other queue, there are 15 packets, 2 from host B and the other 13 from different hosts. One would expect a flow control procedure to throttle host A, which is using 7 times as many buffers as any other host. Under Delta-K, however, B will be throttled.

Thus far, we have only addressed the question of how Delta-K chooses which host(s) should have its (their) traffic throttled. We have not yet discussed how it is determined which flow is selected for throttling. (Two packets are considered to be part of the same flow if they have the same source and destination host addresses and the same precedence level.) The congestion notice that is sent back to a source node not only identifies a host but also a particular flow from that host. Only traffic from the identified flow is throttled. Thus, when a host is selected for throttling by Delta-K, some flow is chosen from those flows which have packets queued at the congested link. There is no attempt to select a "most offending flow" from each offending host; the only rule is that one and only one flow is selected for each host receiving a congestion notice. This flow appears to be selected more or less at random.

The effect of this procedure is to exacerbate the problems associated with the method used to select offending hosts. The net effect of the two procedures is that there is a finite, if not significant, possibility that the least offending flow from the least offending host will be selected for throttling. A specific example dramatizes this point. Suppose that a user at host A is sending a large amount of traffic to host B, that another user at host A is sending a large amount of traffic to host D and that a third user at host C is sending a small amount of traffic to host E. Assume that all of these flows and only these flows travel over link i which, at some point, has K'+2 packets on its queue. In this case, Delta-K will always throttle the flow CE, despite the fact that it is profoundly non-offending.

Performance Criteria Violations

The most obvious performance criterion which is violated in the above examples is fairness. In each example, the host or flow which is not using the largest amount of resources is chosen for throttling. This results in a situation in which some other host or flow is allowed to consume a disproportionate percentage of network resources.

Since the offending source is not necessarily throttled under Delta-K, the effectiveness of the congestion control is

consequently called into question. The analysis of Delta-K presented in Reference [5] assumes that as control traffic (CNs and NDNs) increases, it replaces offending user traffic. Since the announcement that a source is causing congestion is made prior to saturation and since that source is immediately throttled, the analysis concludes that (1) there are sufficient resources to carry the control traffic and that (2) the control traffic is effective in removing congestion. If the non-offending source is consistently chosen for throttling, both of these conclusions will be wrong. The actions of the congestion control mechanism will not cause the offending source to be throttled and the addition of control traffic exacerbates the congestion. The resulting performance can be disastrous, as congestion not only continues to grow but even accelerates as more control traffic enters the network.

Solutions

Clearly a more complex method is required to identify offending flows. There are any number of options, although each requires that more information be provided to the selection procedure than is currently assumed. Ideally, one wants to monitor the utilization of each node resource by each flow (rather than host). These measurements should be averaged over some suitable period of time. When a resource is congested, a flow is selected for throttling in a manner consistent with the network's definition of fairness.

The obvious cost of a more complex procedure is in overhead. The number of possible combinations of sources and destinations in a network is sufficiently large that the memory required to track the utilization of each resource by each flow may be prohibitive. Since the most offending flow is to be selected for throttling, some sorting or searching procedure is required which is cognizant of source and destination host addresses. This induces overhead in the form of added computational complexity. If the definition of fairness is to be relaxed, then one can reduce all of this overhead. Rather than look at source-destination host pairs, one can look at source-destination node pairs.

Parenthetically, it is worth noting that the introduction of destination addresses to the congestion detection algorithms allows for a more general type of congestion measurement. If one observes a large number of source addresses on a long queue, one can decide who should be throttled. A preponderance of some particular destination address on a long queue indicates congestion which is forward of the queue. Thus, the use of destination addresses not only allows one to throttle offending flows but also can help identify the ultimate source of the congestion.

It goes without saying that a procedure which has access to detailed information about network traffic may require the participation of the security kernel.

5.4.2.5 Reliability of CN and NDN Propagation

Problem

Communication between the Delta-K algorithm (at a store and forward node) and the Source Release and Global Window algorithms (at a source node) is accomplished with Congestion Notices (CNs) and Non-Delivery Notices (NDNs). Both CNs and NDNs are treated like datagrams; there is no special reliable transport protocol for CNs and NDNs and they are not acknowledged. The CN packet is assigned the same priority as the data packet for which it was created. An NDN is assigned a high priority. Since there is no reliable transport mechanism for CNs and NDNs, and since CNs are not necessarily sent at a high priority, the possibility arises that these control packets may be lost, or at least excessively delayed during transmission. This possibility is compounded by the fact that CNs and NDNs are generated at a node only when that node is having difficulty handling traffic.

It may be argued that, since NDNs and CNs travel toward the source, they are never placed upon the same queue whose excessive length caused their generation in the first place. That is, they are sent away from the congestion and will not be excessively

delayed. This argument is specious for a number of reasons. First, it does not follow that because a link is congested, links in the opposite congestion are uncongested. Congestion in one direction may, in fact, be directly correlated with congestion in the opposite direction, particularly during the peak hour. This will occur, for example, if there is a large volume of interactive traffic. In addition, the version of TCP that will be implemented on AUTODIN II may result in separate acknowledgments for each packet. Thus packets travelling in one direction automatically give rise to packets travelling in the opposite direction, possibly causing bidirectional congestion. Furthermore, some resource other than a line may be congested. If memory or the CPU is the critical resource in the congested node, then the argument that the CN or NDN may be lost or delayed still holds. Buffer and CPU congestion are not directional.

Finally, it cannot be assumed that a CN or an NDN will not be placed on the queue whose congestion they are reporting. Suppose that source node S is sending traffic to destination node D on a path that travels through intermediate node I. Suppose also that I's trunk in the direction of D and its trunk in the direction of S are heavily loaded. Now consider what happens if I's trunk in the direction of D goes down. The many packets queued for that trunk will need to be rerouted, perhaps being placed on the queue for the trunk in the direction of S. This

will doubtless congest the trunk, causing generation of a flurry of CNs or NDNs to be sent to S. The packets will now be placed on the same queue where the packets from S are residing.

In summary, RAFT does not include a reliable transport protocol for CNs and NDNs. This fact and the fact that CNs and NDNs are exclusively generated in congested nodes enhances the possibility that CNs and NDNs will be lost. Since the congestion control procedure in RAFT depends upon receipt of CNs and NDNs at the source node in order to perform properly, the performance of RAFT can be questioned.

Violations of Performance Criteria

The two performance criteria which are violated are effectiveness and responsiveness. Effectiveness is violated because there is the strong possibility that congestion control will break down under not improbable circumstances. Short of total collapse, the loss of a CN or NDN or an excessive delay in the delivery or a CN will result in an excessively slow response to congestion.

Solutions

There are a number of ways to approach this problem. Currently reception of a CN will result in some sources being throttled for a fixed amount of time. This time period does not

vary from CN to CN. Persistent congestion may therefore require the generation of a large number of congestion notices. If the period for which a flow will be throttled is computed at the congested node as a function of the degree of congestion, the number of CNs required may be accordingly reduced. This will tend to reduce the burden which congestion control places on the links from the congested node. However, it will increase the processing overhead required to produce a given congestion notice.

Unfortunately, reducing the number of CNs makes delivery of each CN more important. It would therefore be cost effective to introduce a reliable transmission protocol for CNs and NDNs.

5.4.2.6 Sudden Throttling at Source Switches

Problem

Upon reception of a CN or NDN, the source node ceases to release traffic from the identified flow into the network. Packets from the flow continue to be accepted by the node and are stored in a temporary holding queue until there are two packets stored. At that point, additional packets received from the host for the throttled flow are discarded. After a fixed amount of time has elapsed from the reception of the original CN or NDN, and assuming that no additional CNs or NDNs have been received, all queued packets may be released into the network. This

procedure is termed Source Release. The Source Release procedure can be thought of as assigning one of three states to each flow. For flows which are held to be uncongested, the procedure operates transparently. For flows which are congested, the algorithm holds off traffic from the network by placing packets in the holding queue. For flows which are saturated, the algorithm discards packets.

The problem with Source Release is that the sudden release of packets from the holding queue and from the host may exacerbate the original congested condition which caused the CN and NDN to be generated. Assume that a host has a large volume of traffic to send and that the host has been temporarily throttled because of the receipt of a congestion notice. One can therefore expect that the holding queue for the flow in question will immediately fill. The source node will then reject all further packets from the flow, but these packets will simply requeue at the host. When the congestion notice times out, the node will stop rejecting packets and the packets queued at the host and at the source node will stream into the network. There is no provision in RAFT to gradually increase the size of flows which have been throttled due to congestion. On the contrary, a significant backlog of packets is allowed to suddenly rush into the network. This flood may produce further congestion.

The problems caused by Source Release would be mitigated if there were some other protocol acting to reduce the volume of some large end-to-end flow. In this case, congestion control would intercede to stop the initial occurrence of congestion caused by this flow and flow control would act to prevent the reoccurrence of congestion. Unfortunately, Source Release is the RAFT end-to-end flow control procedure. Thus, if some flow in the network is excessive, the result will be an endless repetition of congestion and throttling.

Violations of Performance Criteria

Source Release is clearly unstable and ineffective. Under conditions of steady, but excessive, host input, the network offers erratic service. Packets will be accepted from the source and will arrive at the destination in bursts. Excessive flows always cause congestion or are kept from the network; they are never managed so that throughput is maximized without taxing the network.

Control feedback coupling is also violated. The source node, having received a congestion notice, will not receive any additional notification about resource congestion or non-congestion unless it transmits additional user packets which give rise to additional CNs. The technique therefore requires that a source node have host data to transmit in order for the

node to know whether or not some resource is congested. It must contribute to congestion in order to know that congestion exists.

Solutions

The seriousness of these problems may be reduced by modifying the Source Release algorithm. Unfortunately, the algorithm may be tuned only by changing the congestion timeout value and the size of the holding queue. Neither of these changes eliminates the problem of queued data rushing into the network when the congestion timeout period has passed.

A more complex modification would see the introduction of a fourth state called "metered flow", which is intermediate between the congested and uncongested states. While a flow is metered, the rate at which packets enter the network is strictly controlled. If no congestion notices are received, the controls are gradually loosened until they are non-existent.

The major potential cost associated with the metered flow approach is to responsiveness and effectiveness. If the controls are relaxed too slowly, then an excessive amount of traffic will have been withheld from the network and the algorithm will respond too slowly to the absence of congestion. If controls are applied too rapidly, then metered flow will not solve the problem for which it was invented. All of this requires careful tuning. The introduction of metered flows clearly increases the overhead imposed on source nodes.

5.4.2.7 Circumventing the Global Window

Problem

The Global Window restricts the total number of packets which can be submitted from a given host. If a host has most recently been assigned a window size of n, then once that host has submitted n packets, it cannot submit additional packets until it receives a new window size from its source node. The window size is inversely related to the number of congested flows (i.e., flows for which CNs and NDNs have been received) and is directly related to the total number of flows from that host. (The details of the host interface are given in the Segment Interface Protocol [7].)

The Global Window algorithm introduces a number of potentially serious problems. Most of these problems arise from the fact that the window restricts the total traffic from the host; each flow is not separately controlled. A malicious host can easily undermine the Global Window by establishing flows over which it intends to send no traffic. These bogus flows may be established by sending datagrams to a large number of different hosts at multiple precedence levels. Since the Global Window is computed as a function of the number of uncongested flows, the malicious host, because it has established many bogus flows, will receive a very wide window. This window will be wide even if all of the host's "real" flows are congested.

A second problem caused by the Global Window algorithm also arises from the fact that a single window controls all flows. A cooperative host receives no information about which flows are involved in congestion and therefore cannot voluntarily suppress those flows. This is a problem which is present in many similar techniques. Global information, when used to control multiplexed flows, hides details which are essential for the intelligent system-wide management of traffic.

Violations of Performance Criteria

Since a malicious user can undermine congestion control, the procedure is neither effective nor robust. Since all flows are controlled by a single window, non-offending flows may suffer. This will happen because flows which contribute to congestion result in a reduced window size. But a reduced window size affects both offending and non-offending flows. The procedure is therefore not fair.

Solutions

Windows can be implemented for each flow. (This should not introduce the additional burden of a virtual circuit interface.) Alternatively, a single global window can be maintained, but the hosts can be informed of which flows are congested. Clearly the ease with which bogus flows can be established should be reduced.

Introducing windows for each flow or informing the host about congested flows adds to the procedure's overhead. Not only might additional source node memory and processing power be required, but additional access line bandwidth might also be needed.

5.4.3 Conclusions

Our overall conclusion is that AUTODIN II will have serious performance problems with the current congestion control procedure as the network develops beyond the IOC. These problems will probably not be so noticeable under the relatively light load that can be expected during IOC, but will become worse as traffic grows.

Some of the particular problems cited have easy solutions of minimal cost. Other problems have solutions which introduce significant CPU and memory overhead, particularly at source nodes. Many problems have solutions which either require that security be relaxed or that the security kernel actively participate in congestion control. Several problems require further study via simulation and/or network measurements. We wish to stress that it would be unwise to "patch" RAFT by fixing only some of the identified problems. Improvements in AUTODIN II congestion control should be done as part of an overall evaluation. It will be most critical to keep the design changes

extremely well engineered in order to avoid the introduction of new problems.

Some of the issues identified in this critique go beyond the details of RAFT's problems and touch upon two fundamental conflicts. There is a basic conflict between the preventive and curative approaches to congestion control. Does one get better performance if one is conservative about allowing traffic into a network after congestion has been detected than if one adopts a less conservative stance and is willing to respond to, rather than anticipate, congestion? RAFT was designed as a curative procedure with a liberal attitude toward the risk of congestion; our proposed modifications would tend to move it in the preventive direction. There is also a second conflict between the short and long term views of congestion. If one uses smoothed measurements, at what point does responsiveness suffer? If one reduces the time over which observations are averaged, when does one start responding to transients and unnecessarily throttle traffic? Is it possible to find an optimal value for the averaging interval? Absolute statements about these issues must be made with caution until they are better understood.

References

[1] D.W. Davies, "The control of congestion in packet-switching networks," IEEE Transactions on Communications, COM-20, 546 (June 1972).

[2] S.S. Lam and M. Reiser, "Congestion Control of Store-and-Forward Networks by Input Buffer Limits," National Telecommunications Conference Record, 12:1, NTC-77, IEEE Press (December 1977).

[3] L. Pouzin, "Congestion Control Based on Channel Load," Reseau Cyclades Report MIT-600, (August 1975).

[4] P.J. Sevcik and P.J. Nichols, "A Flow Control Technique for Datagram Subnetworks," National Telecommunications Conference Record 32:2, NTC-79, IEEE Press (November 1979).

[5] P.J. Sevcik and P.J. Nichols, "Revised Acknowledgement and Flow Technique (RAFT)," Western Union Technical Note 78-04.2 (September 1978).

[6] P.J. Sevcik and P.J. Nichols, "Packet Transport Protocol Design Considerations," Computer Networking Symposium Proceedings, IEEE Press (December 1978).

[7] V.R. Kulkarni and P.J. Sevcik, "Initial AUTODIN II Segment Interface Protocol (SIP) Specification," Western Union Technical Note 78-07.2 (October 1978).

6. A NEW CONGESTION CONTROL PROPOSAL

In our chapter on multi-path routing, we proposed a congestion control or flow apportionment scheme to be integrated with that routing algorithm. That scheme was based on explicit measurements of link utilization. In this chapter we will discuss the possibility of performing congestion control without explicitly measuring the amount of residual capacity on each network link. While it is clear that such measurements are needed in order to do flow apportionment in the multi-path throughput-oriented routing algorithm, it is much less clear that such measurements are necessary, or even useful, as input to a congestion control scheme which is to be integrated with a single-path, delay-oriented routing algorithm. Hence it is worth investigating a simpler congestion control scheme, which may share many of the properties of the more complex one.

In this simplified scheme, measurements will be performed on each link. As a result of these measurements, each link will be declared to be in one of three states: underloaded, maximally loaded, or congested. The details of the measurement procedure will be discussed later. The measurements will, however, be smoothed over a suitably long interval in order to guarantee that detected state changes reflect real state changes, rather than stochastic variations. The state of each link will be reported in the ordinary SPF routing updates. Generation of a new routing

update will be triggered by changes in these states, as well as by changes in delay.

These states can be applied not only to individual links, but to entire paths as well. A path is "congested" if it has at least one congested link. A path is "maximally loaded" if it has no congested links but at least one maximally loaded link. Paths which are neither congested nor maximally loaded are "underloaded." When specifying our multi-path algorithm, we described a procedure for computing the residual capacity from a source node to a destination node along a specific path, given the residual capacities of the individual links. This procedure involved a small modification of the SPF algorithm, and a new algorithm called CAA. With a suitable bit-coding of the link states (e.g., congested = 0, maximally loaded = 1, underloaded = 2), this procedure can be run without change to associate a state with each path from a given source node to a given destination node. If single-path routing is in effect, each source node can associate one of the three states uniquely with each destination node. Thus we can speak of a destination node as being in, e.g., the congested state, from the point of view of a particular source node.

The way in which congestion control is imposed on traffic to a particular destination will depend on the state of the destination. When a destination transitions to the congested

state, controls must be applied. As long as the destination remains in the congested state, the controls must gradually be made tighter and tighter (i.e., the throughput from the source to the destination must gradually be made less and less). At some point, the amount of control will be "optimal," and the destination will enter the maximally loaded state (assuming, of course, steady-state flows and no change in path). While a destination is in the maximally loaded state, the amount of control exerted on traffic to it should be held constant. While a destination is maximally loaded, increasing the throughput from the given source would overload it, while decreasing the throughput would result in an underload. When a destination enters the underloaded state, controls can be loosened slightly. We do not, however, want to remove all controls on packets to that destination, since it is possible that doing so will result in overload, putting the destination right back into the congested state. Rather, we want to loosen the controls gradually, in the hope that the destination will eventually reach the maximally loaded state, at which point the controls can be clamped. It may even be desirable to prevent an increase in throughput until the destination remains underloaded for a certain period of time; this should enhance the stability of the scheme.

We have not yet spoken of the sort of "controls" to be used in this scheme. There are many different sorts of controls we might wish to investigate. A very simple sort of control scheme might attempt to limit the number of packets to a given congested destination which can simultaneously reside on the modem output queue of their source node. At present, the ARPANET allows eight packets to be on a modem output queue simultaneously, with no restrictions at all as to how many may have the same source or destination. One means of applying congestion control would be to associate with each destination D a number c(D), $1 \leq c(D) \leq 8$, such that no packet originating from this IMP for destination D can be enqueued for modem output if there are already c(D) packets on the queue. At IMP initialization time, the values of c(D) would be set to 8 for all D. Whenever a destination D enters the congested state, c(D) would be set to some lower value. Periodically, the values of c(D) would be decreased by 1 for all destinations D which are in the congested state. However, c(D) would not be allowed to decrease below 1. Whenever a destination D enters the underloaded state, c(D) is increased by 1. The value of c(D) is increased by 1 periodically, until it reaches 8, as long as D remains in the underloaded state. If destination D is in the maximally loaded state, c(D) is left unchanged.

Note that the amount by which c(D) is increased or decreased need not be uniform, but can be a function of c(D) itself. For the sake of fairness, we may wish to try to equalize the flows to a particular destination from the various sources, instead of permitting one source to have a very large flow while forcing another to have a very small flow To effect this equalization, c(D) should be increased by a large amount if it is small, and by a small amount if it is large. If it is necessary to decrease c(D), it should be decreased by a large amount if it is already large, and by a small amount otherwise.

This sort of control should be relatively easy to implement. It may be particularly appropriate if one is worried about congestion being caused by a few sources which create large amounts of datagram traffic. Thus in the ARPANET, a particularly worrisome source of potential congestion is the ability of speech hosts or internetwork gateways to bombard the network with datagrams, which bypass the end-end flow control mechanism. If the scheme just outlined were to be applied only to datagrams (so that c(D) would specify the number of datagram packets to D which may be on a modem output queue simultaneously, with no restrictions on other packets), it is possible that this potential source of congestion would be eliminated. Furthermore, the scheme so restricted would have no effect on non-datagram traffic; it would control only that class of traffic which needs to be controlled.

We have suggested that controls be placed only at the source node. It is worth considering whether the same controls should not also be applied at intermediate nodes. Whereas applying control only at the source node causes excess traffic to remain queued outside the net, applying controls at intermediate nodes can cause traffic to back up within the network. Since the latter effect is much less desirable than the former, it seems preferable to apply controls only at the source. However, this should be tested through the use of simulation.

A disadvantage of this means of control is its coarseness and inflexibility. If a destination is still congested even though each source node will enqueue only one packet at a time for that destination, the proposed scheme allows no further method of reducing the flow to the congested destination. Furthermore, when increasing or decreasing the maximum allowable flow, one does not have very fine control over the step size. Also, the precise nature of the "one packet at a time" restriction depends on the queue length at the source, as well as the characteristics of the line. This may make the effects of the scheme difficult to understand, and may make it more difficult to bring about fairness. Therefore it is desirable to consider a somewhat more complex but more flexible means of controlling the flow. One possibility is to use a method similar to that used in the multi-path routing algorithm. Each source

node would measure its flow to each destination node D (call it f(D)). The flow measurement would be suitably smoothed over a measurement interval. The control variable c(D) would specify a maximum amount of flow allowable from that source to destination D within a fixed interval. Initially, c(D) would be set to infinity for all values of D. When destination D enters the congested state, c(D) is set to some fraction of f(D). The value of c(D) is periodically decreased as long as D remains in the congested state. When D enters the uncongested state, c(D) is increased, and it continues to be increased periodically as long as D remains in the uncongested state. If D is in the maximally loaded state, c(D) remains constant. Note that when a path becomes congested, the initial controls are based on f(d), rather than c(D). This is necessary, since, in such a situation, c(d) may be much larger than f(D) and it is known that f(D) is already too large. Of course, the fraction of f(D) to which c(D) is set is a parameter, as are the amounts by which c(D) may be increased or decreased. Optimal values for these parameters will have to be determined empirically.

We have yet to discuss the way in which it is determined whether a given link is in the congested state, the maximally loaded state, or the underloaded state. There will have to be a measurement process at each node which determines the state of that node's outgoing links. The measurement process must be such

that it declares lines to be underloaded only if the node itself is underloaded. Thus the measurement process is similar to that which we proposed for the multi-path routing algorithm. It can, however, be much simpler, since it need not specify the precise amount by which a line is underloaded or overloaded. In the ARPANET, empirical investigation has shown that certain easily detectable events are highly correlated with (and causally related to) various sorts of overloads. If a link is overloaded (i.e., an attempt is being made to send more than 50 kbps of traffic on it), large numbers of packets will be refused because there are no logical channels for them. If the node's buffer space is overutilized, many packets will be refused due to a lack of buffers. A heavily utilized CPU tends to produce long TASK queues. The TASK queue length rarely exceeds two unless the CPU is overutilized. Therefore, by keeping a count of the number of refusals plus the number of times a packet was placed on the TASK queue while two or more packets were already queued, one can get a good indication as to the loading of a node and its links. Refusal counts would be kept separately for each link; the TASK queue length count would be common to all links. We will refer to these counts as "congestion counts," and will mark a line as being in the congested state when its congestion counts exceed a certain threshold within a certain time interval. In order to distinguish a true, steady-state overload from a momentary surge, it may be desirable to require the congestion counts to exceed

the threshold in several successive intervals (or in k out of n successive intervals, where k and n are parameters) before declaring the line to be in the congested state. A line can be regarded to be in the maximally loaded state whenever its congestion counts are below the "congestion" threshold for several successive intervals, but are above zero. It is possible, however, that this measurement technique will not distinguish between "maximally loaded" and "underloaded," since surges in traffic on an underloaded line may cause the congestion counts to take on small but non-zero values. If this turns out to be the case, we could try to refine the measurement by also counting "near-refusals," i.e., packets which, although not refused, take the last buffer or logical channel. A line will be declared to be underloaded as long as it is neither congested nor maximally loaded.

This congestion control scheme, if implemented in the ARPANET, would add very little overhead on the lines, since only two bits per line would need to be added to each update (a maximum of 10 additional bits), and there are already several unused bits in the update packet. There might be some increase in the frequency with which routing updates are sent, since update generation will be triggered by changes in congestion state, as well as by change in delay. However, since the controls have been designed to operate in a smooth manner, and

since the controls are tightened or loosened periodically without the need for additional updates to be received, we would not expect the congestion state of a line to change much more frequently than the delay in the line. As long as delay changes are more frequent then congestion state changes, the frequency with which routing updates are generated will not increase.

The congestion control scheme described here is certainly more efficient in its use of line bandwidth and nodal bandwidth than is the congestion control scheme which is integrated with our multi-path throughput-oriented routing algorithm. The latter requires a more complex measurement process in each node to determine the residual capacity of each link, and it requires larger updates in order to disseminate this information. It may seem, however, that the latter scheme would also be much more effective at controlling congestion. It does, after all, provide a more precise indication of the amount of residual capacity along each path, enabling source nodes to effect their throughput controls with a greater degree of accuracy. However, as we have discussed in section 4.8, this additional accuracy can only be achieved on the assumption that changes in the identity of a path to a given destination occur much less frequently than changes in the residual capacity along the path to the destination. In a throughput-oriented routing algorithm we can force this assumption to be true (see section 4.8). In a delay-oriented

algorithm, however, we cannot make any statement in general about the frequency of path identity changes relative to the frequency of residual capacity changes along a non-changing path. Trying to base our throughput controls on precise measurements of residual capacity, therefore, may be as likely to reduce accuracy as to enhance it.

The congestion control scheme described here should work well even when an offered overload of traffic causes the delay-oriented routing to oscillate. For example, suppose a source node attempts to send 70 kbps of datagram traffic to a single destination. In the ARPANET, with its 50 kbps lines and single-path routing, the load cannot be handled. With no congestion control, any path used will be congested, and each line on the path will experience extremely large delays. This will cause the routing algorithm to switch from path to path, even using some very long and unsuitable paths, in a futile attempt to find a path which can handle the traffic. With the proposed congestion control scheme, throughput controls would be in effect, with gradually increasing severity, no matter what path is being used. Eventually the throughput controls would force the traffic load down to a feasible amount, enabling the routing algorithm to settle on the true path of least delay. This is precisely the sort of congestion control that would be needed in the ARPANET to prevent congestion due to datagrams.

7. SIMULATION NETWORK DESCRIPTION

7.1 IMP

Each ARPANET IMP in the simulation is represented by a Simula object, also called an IMP. The IMP is just an object which contains some data structures, pointers to all the active IMP processes, and some procedures. Descriptions of the IMP processes (Task, ModemOut, ModemIn, HostOut, HostIn and Timeout processes) are to be found in the following pages, as are the functions which implement routing and buffer allocation.

The IMP's other task is to create and initialize the IMP processes and to create the local host. Descriptions of the parameters passed to the processes can be found under the description of the processes themselves.

7.2 Host

There is one host per IMP. Each host is a class which contains some data structures and some procedures. Each host has (a pointer to) one packetSink process which is responsible for accepting packets from the net, an array of (pointers to) messageGen processes which generate messages for each destination, and a messageOut process which transfers the messages to the IMP. In the current simulation there is no distinction between destination IMP and destination host, since there is just one host per IMP.

The host implements two procedures, Start and Close. Start takes a destination IMP number, the average message rate (in messages per unit of simulated time), and the average message length, and starts p a messageGen process to that destination sending at that rate. If there already was a generator to that destination, it is closed. Close takes the destination IMP number and stops the messageGen process which is sending to that destination. The process is then garbage collected.

The messageGen process is a loop which waits for a randomly determined period of time and then generates another message. The time between message "arrivals" 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 other two parameters, also supplied by the call to Start, are the destination for all messages from this generator, and the average message length. The message length is selected at random from a negative exponential distribution with the given mean. When each message is created, the source, destination, 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 copies the message into a packet, fills in the nodentryTime and netEntryTime fields, and puts the packet on HostIn's input queue. When HostInterface returns, MessageOut takes the next message off its queue.

The packetSink process is started by the host when the host is started by the local IMP. The packetSink process has one parameter, the processing time per packet. This is set by the command interpreter when the host is created, and may be changed by the appropriate command.

The packetSink process has an input queue. It waits till 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. Then the process waits for its service time, returns the packet to the Simula garbage collector, and loops to wait for another packet.

7.3 Task

There is one task process per IMP. The IMP starts the task process, specifies the priority at which it will run, and the amount of time each packet takes. Task has the lowest priority below Timeout (see Sec. 7.10 for details). There are no facilities for setting or changing this. The service time is a parameter to the process, which is set when the IMP is created, and may be changed by the appropriate command

Task has two queues: a queue for data packets, and a queue for routing update packets. All the routing updates are always processed before any data packet. If there is a routing update on the queue it is removed, and Task calls the IMP routine ProcessUpdate. When 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 the HostOut process or to one of the ModemOut processes. If the channelBit specified in the packet is the same as the receive flag for the line and channel the packet came in on, then the packet is a duplicate; it is discarded. If the packet's destination is this IMP, then 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 discarded.

If the packet's destination is some other IMP, then Task attempts to allocate a store-and-forward buffer. If it cannot allocate a buffer, the packet is discarded. If it can allocate a buffer, it looks up the destination in its IMP's routing table, which specifies the output line to use. If the output line is specified as zero, then the destination IMP is inaccessible; the packet is discarded. If the output line is not zero, Task attempts to allocate a logical channel on that line (by calling the routine "GrabChannel" in the output process for that line).

If a channel cannot be allocated, the packet is discarded. If a channel is allocated, the packet is put on the queue of the correct output process. If the packet is passed to hostOut or to one of the output processes, or is dropped because the destination IMP is inaccessible, then it must be acknowledged. Task calls the routine AckPacket in the ModemIn process 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.

7.4 Routing and Forwarding

Forwarding in the present simulation is simple. Task looks up a table in the IMP which tells it which output line to use to send a packet to a given destination. Routing is the process involved in building that table.

Each output line has a current delay measure and a delay accumulator. The current delay is used by the routing algorithm in calculating the shortest (minimum delay) paths. The routing algorithm is implemented in a single function, UpdateRouting, in each IMP. The function uses a shortest path first (SPF)

algorithm to calculate the minimum delay path from its IMP to any other. As the shortest path to any IMP is calculated, the output line to use in forwarding is noted in the table.

7.5 Delay Measurement

When a packet arrives at a node, via a call to either HostInterface or ModemInterface, the time is noted in the field "nodeEntryTime." When a packet is transmitted out a line the time is noted in the field "nodeExitTime." When the line protocol (q.v.) gets an acknowledgment for the packet, the node delay (nodeExitTime - nodeEntryTime) is computed and the routine TallyDel is called. This routine increments the packet counter totalPackets, and adds the delay into the delay accumulator totalDel. Although TallyDel is called by ModemInput, it is implemented as part of ModemOut, since that is where the relevant data structures are kept.

Periodically, the slow 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. First, the IMP routine reduces the delay threshold by a fixed amount; 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 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 tables. Finally, the update serial number is incremented, the update age is reset, and the IMP routine SendUpdate is called to send out the update on each line.

7.6 HostOut

There is one HostOut process for each IMP. If we were to implement multiple hosts, each host would have its own HostOut process, just as each output line has its own output process.

When the HostOut process is started by the IMP, the IMP sets its priority; the priority is set lower than HostIn and higher than Timeout (see Sec. 7.10 for details). The serviceTime is set when the IMP is created, and may be altered by the appropriate command.

The HostOut Process has an input queue and a pointer to the local Host. Task puts packets on the input queue. HostOut waits for a packet to be put on the queue, removes it, executes some simulated CPU time, 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.

7.7 HostIn

There is one HostIn process for each IMP. If we were to implement multiple hosts, each host would have its own HostIn process. HostIn is responsible for input of messages from the host, and the division of messages into packets. Currently, messages are not divided up, rather, it is assumed that all messages from the host are single packet messages.

When the hostIn process is started by the IMP, the IMP sets its priority; the priority is set lower than ModemOut and higher than HostOut (see Sec. 7.10 for details). The serviceTime is set when the IMP is created, and may be altered by the appropriate command.

To transfer a message to the IMP, the host calls the routine HostInterface, implemented as part of HostIn. HostInterface allocates a packet for the message and puts it on HostIn's input queue. HostIn takes it off the queue and gives it to the task process.

HostInterface takes a message as an argument and attempts to allocate a buffer. If a buffer cannot be allocated, the message is discarded and the routine returns. If a buffer can be allocated, the nodeEntryTime field is set to the current time, the priority is read from the message, and the input line field is cleared so Task knows that this packet came from the host and

does not have to be acknowledged. Then HostInterface puts the packet into the input queue and returns.

7.8 ModemOut

There is one output process for each output line in the IMP. The output process implements procedures for allocating and freeing logical channels on the line, and for computing the average delay and invoking the routing computation for each IMP. The output process contains a standard input queue, a pointer to the line, the delay measurements, the send flag for each logical channel, an array of packets to keep track of packets which have been allocated to a channel but not yet acknowledged, and a flag which is set if a packet is acknowledged while it is actually being transmitted. Channel allocation and handling of unacknowledged packets are described in Section 7.11. Delay measurements are described in Section 7.5.

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 7.10 for details). The serviceTime is set when the IMP is created and may be altered by the appropriate command.

The ModemOut code is in two parts. First, it checks to see if there are old packets to be retransmitted or new packets to be transmitted; if packets are waiting, it goes on to the second

part, if not, it goes to sleep. The second section involves actually putting the packets on the line.

In the first section ModemOut checks to see if there are any packets sent but not acknowledged, that were sent more than retryInterval units of simulated time ago. RetryInterval is a parameter to ModemOut which can be set at any time by the corresponding command. If there are packets to be retransmitted, it goes on to the second section. If not, it checks to see if its input queue is empty or if it should send a null packet; if there is nothing to send, it goes to sleep, otherwise, it goes on to the second section. When ModemOut goes to sleep, it can be woken by task calling the transmit routine to send a packet, by a wakeup to send an acknowledgment (see "line protocol"), or by a periodic wakeup from fastTimeout.

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

7.9 ModemIn

There is one ModemIn process for each input line in the IMP. ModemIn contains the receive flag for each channel on the line, a standard input queue, and a flag which, when set, indicates that there is an acknowledgment to be sent. ModemIn implements a routine, ModemInterface, which is called by the line to pass a packet to the process, and a routine, AckPacket, which is called by task to return an acknowledgment for a packet which has been accepted.

When the IMP creates a ModemIn process, the IMP sets its priority. Each ModemIn process is given a different priority. The ModemIn processes have the highest priority, above the ModemOut processes (see Sec. 7.10 for details). The serviceTime is set when the IMP is created and may be altered by the appropriate command.

When the line calls ModemInterface, it passes a pointer to the packet from the sending IMP. First, the routine attempts to allocate a buffer for the packet by calling the IMP routine Grabpacket. If this succeeds, and the packet is not marked "error" by the line, then all the fields of the old packet are copied into the new packet, except that the line number is changed, the priority is reset, and the node EntryTime is set to the current time. The line number is changed since this is just

the IMP's internal index for the line (or modem), not a unique global number. The buffer is then added to ModemIn's (own) input queue. If a packet cannot be assigned, the packet is discarded, and the routine returns. If a buffer was allocated but the packet was marked in error, then the buffer is returned, the packet is discarded, and the routine returns. The packet from the sending IMP is never changed.

The AckPacket routine simply flips the receive flag for the specified channel and sets the flag which indicates that an acknowledgment should be sent. It is described further in Section 7.11.

The ModemIn process waits until a packet arrives on its input queue (i.e., is put there by a call to ModemInterface), executes serviceTime units of simulated CPU time, compares the acks in the packet to the send flags in the output process for this line, and frees the output channel where they are the same. (For more details see Section 7.11.) If the packet is a null packet, it is discarded; if it is not, it is put on task's input queue and the process loops.

7.10 Priority Structure

Every process in the IMP has a separate priority, including each separate ModemIn and ModemOut process. The order of priorities, highest first, is: ModemIn, ModemOut, HostIn,

HostOut, Timeout, Task. Within ModemIn and ModemOut, modem 1 has the highest priority, followed by modem 2 and so on. Thus, for an IMP with two modems the priority structure would be:

process	priority
ModemIn(1)	1
ModemIn(2)	2
ModemOut(1)	3
ModemOut(2)	4
HostIn	5
HostOut	6
Timeout	7
Task	8

The process scheduling is pre-emptive. 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 "service 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.

7.11 The Line Protocol

The line protocol is the mechanism used to ensure that one and only one copy of a packet is passed from IMP to IMP. The line protocol incorporates facilities for acknowledging receipt of a packet, and for discarding duplicates of a packet. The line protocol uses logical entities called channels. The number of channels per line is currently fixed at 8. Only one packet at a time can be unacknowledged on any channel. The fields necessary for the line protocol are carried in the packet. They are:

an array of flags (1 flag per channel)
a channel flag
a channel number

In addition, the IMP keeps track of the line number in the packet. At both the send and receive ends of each channel, the line protocol uses flags called the send and receive flags. The send flags for each line are stored in the output process for the line; the receive flags are stored in the input process. The other participant in this process is the task process. The idea of the line protocol, in short, is that successive packets on a given channel get opposite values of the channel flag, which is copied from the send flag into the packet, carried over the line to the receiving IMP, and copied into the receive flag.

When Task removes a packet from its input queue, it first checks to see if it is a duplicate. If the receive flag for the line and channel is the same as the channel flag, the packet is a duplicate and is discarded. If the packet is accepted (for forwarding or transmission to the host), Task calls the routine AckPacket in the input process which received the packet. This routine flips the receive flag and calls the routine Ack in the output process, which sets a flag indicating that an acknowledgment should be sent, and wakes the output process if it is asleep. Any packet traveling in the opposite direction has all the receive flags from the input process copied into the outgoing packet by the output process, including the new value of the receive flag for the channel of the just arrived packet. When the packet arrives at the input process of the other IMP, the flags in the packet are compared with the send flags in the corresponding output process. Any matches indicate that a packet has arrived at the first IMP and has been accepted by Task. The input process calls the routine Free in the output process for each such channel. Free flips the send flag (so the next packet sent on that channel will get the opposite value channel flag to the last one), calls the routine TallyDel to calculate the packet delay and update the delay accumulators, clears the UnackedPacket array for this channel, and frees the buffer.

The output process also implements a routine GrabChannel for allocating a logical channel. This routine takes the packet as its argument and returns true if a channel could be allocated, and false otherwise. The routine simply searches UnackedPacket for a slot, and if it finds one, puts a pointer to the packet in the slot, fills in the channel number and channel flag, and returns true. If there is no empty slot, the routine returns false.

It is possible to tell the difference between a packet which has been allocated a channel but not sent, and a packet which has been sent but not acknowledged, since in the former the nodeExitTime is zero. The output process needs to make this distinction correctly when it is deciding whether to retransmit a packet, since packets are stored in UnackedPacket as soon as a logical channel is allocated, before they are sent.

7.12 Routing Update Protocol

The routing update protocol is the protocol that ensures that every IMP sees a copy of every update, and that duplicate, old, or out-of-date information is ignored. Every update contains a serial number, an age, and a retry bit. The serial number is a mod 64 number which is unique (mod 64) 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 8-second intervals. When

the age field reaches zero the update is neither discarded nor broadcast to other IMPs. Each line has a timer for each IMP. When the timer expires, the update for that IMP is sent out on that line with the retry bit set. The timer is shut off when the update is echoed by the receiving IMP. The retry bit is simply a request for an echo.

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.

The IMP routine TimerTick (called by fastTimeout every fast tick) 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.

The IMP routine AgeTick (called by slowTimeout every 12 slow ticks) 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.

7.13 Timeout Process

Each IMP has one Timeout process. When the IMP starts up the Timeout process, it supplies as a parameter the perio. between successive wake-ups.

Timeout process has a counter which counts up to 25 and is then reset to 1. Normally when the process wakes up it calls the routine FastTimeout. On every 25th execution, it calls SlowTimeout and then FastTimeout.

FastTimeout calls the IMP routine TimerTick, which decrements the update timers on each line. This is described in Section 7.12. Each time FastTimeout runs, it checks a line to see if it is idle (i.e., if the corresponding ModemOut process is asleep) and, if so, it wakes the line so it can check whether a retransmission or a null packet should be sent. FastTimeout checks successive lines on successive calls.

SlowTimeout calls the IMP routine AverageDelay every 15 ticks, and the IMP routine AgeTick every 12 ticks. AverageDelay is described under "Delay Measu-ement" and AgeTick is described under "Routing Update Protocol."

7.14 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 25.6 msec clock. 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, there is a parameter in each IMP process which is the ratio of IMP clock to simulated time. This parameter is set by the RATE subcommand. If this parameter is R, then the 25.6 msec clock in fact runs every R * 25.6 msec, and a process whose service time is S msec executes for R * S msec on the simulated CPU.

Second, each IMP is started at a different time, specified by the parameter OFFSET. Suppose this parameter is set to 0, and assume the simple case where R = 1 (the IMP clock is neither fast nor slow). Then the 25.6 msec clock will go off at 0 + 25.6, 0 + 51.2, 0 + 76.8 and so on.

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 OFFSET 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 OFFSET to particular values for each

IMP. Finally, if we want to approximate the asynchronous behavior of real IMPs, we can set both RATE and OFFSET to particular values for each IMP.

7.15 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 re-allocates (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 numbers 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.

8. THE SIMULATION COMMAND LANGUAGE

8.1 Introduction

In the simulation, we need a way to specify, create and modify IMPs, lines and hosts. It is also necessary to have a way of setting up defaults in a convenient manner.

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

<command> <object-specification> <arguments>

The format of the other commands is given in the list of commands below.

For IMPs, the IMP number must be specified, and the number of lines must be given when the IMP is being created:

IMP 53 3 <arguments>

For hosts, the host and IMP number must be specified:

HOST 2/53 <arguments>

which refers to host 2 on IMP 53. At the moment, there is only one host per IMP, so the host number is ignored. To connect two IMPs together one simply gives their numbers:

LINE 53 60 <arguments>

except in the case of multiple lines connecting two IMPs, in which case one must also specify the line number:

LINE 53 60 2 <arguments>

Each line must be specified in a separate command, multiple lines are not yet implemented.

In order to change parameters, the commands IMP, HOST and LINE are used. The command language interpreter will recognize when an IMP, host or line is mentioned more than once, and interpret all occurrences after the first as being for changing parameters rather than for creating the IMP, host or line.

The user will be able to set up defaults by using "models." One can think of 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. The default model is 0, and model parameters not specified in the model statement will be defaulted to the model 0 default. That is, model 0 gives the defaults for other models, as well as being

the default model for IMPs, hosts and lines whose model is not specified.

If a model is specified when an IMP, host or line is created, it must be specified first in the list of arguments. When parameters are changed by a subsequent statement, the model number must not be repeated.

The <arguments> section of the command is used to specify or change parameters of the IMP, host or line. The parameters which can be specified are entirely arbitrary, and not at all restricted by the command language, except that they must have the general form:

<name> <value>

Typically, they will refer to timing parameters, error rates, or tracing flags:

TASK 0.63	RETRANSMIT 10.0
ERROR 0.001	SPEED 9600
TRACE OFF	DEBUG ON

In the case of parameters which may be different for different lines, an alternate command form can be used:

<names>/<index> <value>

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

RETRANSMIT/54 60.0

The retransmit command sets the interval between retransmissions of an unacknowledged packet. The above example sets this time to 60.0 (seconds), but only for the output line (and process) which connects this IMP to IMP 54.

8.2 List of Commands

INIT i j

Initialize the number of IMPs to i and the number of simplex lines to j. This command must occur before any IMPs, hosts or lines are created. The number of IMPs must be less than 100.

NUMBUFFERS n

Set the number of buffers in an IMP to n. This value applies to all IMPs created after this statement, until another NUMBUFFERS statement.

This command will be eliminated and replaced with an IMP command, so that defaults can be set up using models, each IMP can specify a value when it is created, and so on.

FIXEDROUTING

Set a flag in every IMP so that delay computations and routing updates will not occur. The IMPs routing tables can be set with the ROUTE command, which allows the route to be given explicitly, or with the UPDATE command, which forces the specified IMPs to calculate the SPF route to each IMP. With this flag set, the routing tables will never be changed, except by the above commands.

RANDOMROUTING

Set a flag in every IMP so that packets will be forwarded to a neighbor chosen at random. An output line is chosen, and if a logical channel is available on that line, the packet is forwarded; otherwise another random choice is made. If no line has a channel available, the packet is dropped.

This command also sets the fixed routing flag so delay and routing updates will not occur.

ROUTE i n1 n2 n3 ...

Set the routing table in IMP i to n1, n2, n3 and so on. This means that packets for IMP 1 will be forwarded via neighbor n1, packets for IMP 2 will be forwarded via neighbor n2, and so on. If one does not specify a neighbor for every destination, omitted destinations will be defaulted to the last neighbor mentioned.

DEBUG i1 i2 i3 ...

Set the debugging flags to i1, i2, i3 and so on. Each i is either 0 or 1. Each debugging flag controls one particular piece of debugging output.

The different flags are documented in Section 8.7, tracing and debugging.

DEBUGFILE f

Direct debugging output to file f. Simula restricts file names to six alphameric characters with a three-character extension. A device name may be specified, but not a directory.

TRACE i1 i2 i3 ...

Set the trace flags to i1, i2, i3 and so on. Each i is either 0 or 1. Each trace flag controls one particular piece of trace output.

The different flags are documented in Section 8.7, tracing and debugging.

TRACEFILE f

Direct trace output to file f. Simula restricts file names to six alphameric characters with a three-character extension. A device name may be specified, but not a directory.

IMP i n <arguments>

IMP i <arguments>

Create IMP i with n lines, or modify IMP i. The argument n must be given if and only if IMP i is being created (i.e. mentioned in an IMP statement for the first time); i must be positive, and less than or equal to the number of IMPs specified in the INIT statement. All IMPs must be created before the simulation is started. See below for a list of parameters that can be given in the argument list.

LINE i j <arguments>

Create or modify the simplex line connecting IMP i to IMP j. Both IMP i and IMP j must have been created. See below for a list of parameters that can be given in the argument list. The number of lines created must be equal to the number of lines specified in the INIT statement. All lines must be created before the simulation is started.

HOST i/j <arguments>

Modify host i on IMP j. IMP j must have been created. Currently there is only one host per IMP; i is ignored. See below for a list of parameters that can be given in the argument list.

MODEL IMP i <arguments>

MODEL HOST i <arguments>

MODEL LINE i <arguments>

Create or modify the specified model. The parameters which can be given in the argument list are the same as for the specified real object. The argument i must be between 0 and 10 inclusive.

START s d r l

Start transmitting messages from IMP s to IMP d. Message arrivals will be Poisson, with a mean message arrival rate of r. The message lengths will be negative exponential with mean l. IMP s must have been created; l must be positive; if r is less than or equal to zero, the flow of messages from s to d will be stopped. A subsequent START command with the same source (s) and destination (d), overrides (rather than adds to) this command.

RUN t

Run the simulation for t secs. All IMPs and lines must have been created. After t seconds of simulated time, the program will prompt for more commands.

READ f

Read commands from the file f. Initially, the program reads from the terminal. Multiple READ commands are stacked so that when all commands have been read from f and executed, the program will resume reading commands from after the READ command.

QUIT

Exit the simulation.

8.3 Parameters

There are two subcommand formats, depending on whether the subcommands are on a single line or several lines.

The single line format is:

```
sc  v  sc  v...
```

where sc is the subcommand and v the value. The multiple line format is:

```
( sc  v  sc  v  ...
      sc  v  sc  v  ...
          ...
          ...       )
```

where line breaks may occur anywhere except between an sc and the following v.

8.4 IMP parameters

TASK t	task processing time is t seconds
MODEMIN t	modemin processing time is t seconds
MODEOUT t	modeout processing time is t seconds
TIMEOUT t	timeout period is t seconds
DELAY t	initialize line delay to t seconds
RETRANSMIT t	retransmission period is t seconds
HOSTIN t	hostin processing time is t
RATE r	IMP clock runs at r times real time
OFFSET t	IMP (and clock) will start running at time t
THRESHOLD t	Initial threshold for delay is t seconds
DECAY t	Threshold will decay t seconds each period
TRACE ON/OFF	turn tracing on or off for this IMP
DEBUG ON/OFF	turn debugging output on or off for this IMP

8.5 Host parameters

SINK t	packet sink processing time is t seconds
TRACE ON/OFF	turn tracing on or off for this host
DEBUG ON/OFF	turn debugging output on or off for this host

8.6 Line parameters

ERROR r	bit error rate is r
LAG t	propagation delay is t seconds
SPEED s	line speed is s bits/second
TRACE ON/OFF	turn tracing on or off for this line
DEBUG ON/OFF	turn debugging output on or off for this line

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.

8.7 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 accepted by the 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. The files may be the same. Each IMP, host and line has a flag which turns tracing on and off, controlled by the TRACE subcommand, and a flag which turns debugging on and off, controlled by the DEBUG subcommand. There are a number of global flags, one for each event which may produce tracing output. These flags are controlled by the TRACE command. If the event occurs in an IMP, host or line which has its local tracing flag set, then tracing output is generated, if and only if, the global tracing flag for that event is set. That is, output is generated only if both the local flag and the global flag are set. There is a similar set of global flags for debugging output, 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 TRACE command, and set the local flags in some number of IMPs, hosts or li es. 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).

8.8 Global Debugging Flags

The flags are specified by index:

1 task process (IMP)

2 modem input process (IMP)

3 modem output process (IMP)

4 host input prccess (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 changes (IMP)

12 message output process (host)

13 packet input process (host)

14 IMP-to-IMP link protocol (IMP)

All of debugging output is in a standard form:

process i [j] time n [m] event [other]

Process is the name of the process. For IMPs, i is the IMP number and j is either the neighboring IMP number (for input and output processes) or is omitted; for hosts, i is the IMP number and j will be the host number when multiple hosts are implemented; for lines, i and j are the IMP numbers on each end of the line. Time is the current simulation time. 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. 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.

Omitted fields are filled with blanks so that all fields are lined up.

8.9 Tracing Output

There is only one tracing output implemented so far. When a packet arrives at the destination host, if tracing flag 1 and the tracing flag for that host are set, the following fields are output:

the source and destination IMP number
the priority
the time the message was created
the time the message entered the network
the network delay
the packet length
the number of IMPs in the route taken
the route taken by the packet

9. PROBLEMS IN THE ANALYSIS OF SIMULATION DATA

In the process of building our simulation facility we have devoted a great deal of effort to developing both a methodology and a set of automated tools for the rigorous analysis of simulation data. The general problem which this methodology must address has two components: estimating the values which performance measures take on in a given simulation model and comparing the performances of two or more different simulation models. In this context, we associate a strict meaning with the term model, namely, a given model is defined by a given simulation program and input parameters. Thus, two different models may have grossly different simulated protocols or may have the same protocols but different input parameters.

One can give additional clarity to the problem which an experimental methodology must address with the following example. Suppose that one is given two alternative routing algorithms, each of which has 3 parameters. Each of these parameters can assume one of 10 values. Suppose further that one is given a performance criterion (say, mean packet delay) and a range of possible traffic matrices and network topologies. It is obvious that deciding which algorithm is "best" is a very complex problem. For a given algorithm and choice of parameters, one must estimate the value that the performance measure assumes for each traffic matrix and topology. (As we shall see, this is, in

itself, a difficult problem.) For a given algorithm, one must estimate which combination of parameter choices yields the "best" performance. Finally, one must compare the "best" performance achievable for the two different routing algorithms. It is important to note that the difficulties associated with this analysis are not solely related to mathematical rigor. If one investigates all combinations of algorithms and parameters and if one proceeds at the healthy clip of two simulation runs per day, the problem as stated is, for all practical purposes, unbounded in time, requiring years to solve.

In the theory of experimental design, an input variable is called a "factor" and the value it assumes in a given experiment is called its "level." A factor may be qualitative or quantitative, e.g., it may be an algorithm or an input parameter. There is a large body of literature devoted to the design of so-called factorial experiments in which one attempts to efficiently estimate the impact of different factors and particular levels of factors upon performance, and to rank these effects [3]. Our implementation of these techniques will be discussed in a subsequent report. In this report we confine our attention to the more basic problem of determining, with confidence, the performance of a given simulation model. This area, central to the analysis of simulation experiments, is quite difficult; we have found that a satisfactory solution requires a

number of approximate techniques drawn from the areas of statistics and time series analysis.

In the following section, we define the elementary statistical concepts employed in parameter estimation. In the subsequent sections, we discuss the problems encountered when one attempts to apply these concepts to the analysis of simulation data, the methodologies used to overcome these difficulties and the way in which these approaches are implemented in our analysis routines.

9.1 Basic Statistical Concepts

A particular network simulation experiment might typically take the following form. A stream of random numbers is used to generate packet interarrival times, packet lengths, nodal processing delays, etc. A finite number of simulated packets are produced and, for each packet, the end-to-end delay is recorded. These end-to-end delays are averaged and the resultant number is taken to be the measure of the performance of the particular model being examined.

This description suggests a fundamental problem. At the end of the experiment, one does not know the "true" value of the mean packet delay belonging to the model. Instead one has an estimate for the mean delay which is dependent upon the specific random number sequence used in the experiment and the number of packets generated. This estimate, which is known as the sample mean, is itself a random variable which may be very far from the "true" system mean. A different random number stream and a different number of observations might have yielded grossly different results.

The analysis of simulation experiments requires the ability to estimate, with confidence, the value of some system property based upon observations made upon the system during a finite interval of simulated time. Thus, two questions are immediately

posed. Can one determine how good an estimate is? Will additional observations produce a better estimate? This section lays the groundwork for an attempt to answer these questions by presenting the elementary definitions and theorems drawn from probability theory and statistics which are used in the theory of parameter estimation. As was suggested in the introduction to this chapter, few of these concepts directly carry over to the analysis of simulation experiments. The necessary modifications are discussed in the next section.

Definitions

Given a random variable X which can assume discrete values $\{x_k\}$, its mean is defined as:

Eq. 9-1
$$\mu_X = E(X) = \sum_{\text{all } k} x_k P_X(x_k)$$

where $P_X(x_k)$ is the probability that the variable X assumes the value x_k. If X is a continuous random variable, i.e., the probability that it assumes any particular value x is zero for all x, then

Eq. 9-2
$$\mu_X = \int_{-\infty}^{\infty} dx\, f_X(x)$$

where $f_X(x)dx$ is the probability that the value of X is in the interval $(x,x+dx]$ The functions $P_X(x_k)$ and $f_X(x)$ are called the probability mass and probability density functions of the distribution.

The second central moment of the variable X, or variance, is defined as:

Eq. 9-3 $$\sigma_X^2 = E[(X - \mu_X)^2]$$

The standard deviation of X is defined as the square root of the variance:

Eq. 9-4 $$\sigma_X = \sqrt{\sigma_X^2}$$

The preceding equations are drawn from probability theory. In actual experimentation one operates in the domain of statistics, in which one does not have a known probability distribution at one's disposal but must infer information about an unknown underlying distribution from a finite set of observations made upon a random variable which has that unknown distribution. In other words, one is given $\{X_i : 1 \leq i \leq n\}$, a set of n observed values for some random variable, and one attempts to estimate parameters associated with the underlying

distribution from which the $\{X_i\}$ are drawn. In order to accomplish this end, one defines an estimator, $\hat{\theta}_n$, of a parameter, θ, of X to be a random variable which depends upon the observations $X_1,\ldots,X_n$. One says that an estimator $\hat{\theta}_n$ of θ is unbiased if

Eq. 9-5 $$E[\hat{\theta}_n] = E[\theta]$$

One says that it is consistent if

Eq. 9-6 $$\lim_{n\to\infty} \text{Prob}\,[|\hat{\theta}_n - \theta| < \varepsilon] = 1 \qquad \forall \varepsilon$$

If the above equation holds, $\hat{\theta}_n$ is said to converge in probability to θ. Thus, an unbiased estimator will have as its expected value the true value of the parameter. A consistent estimator can be expected to be increasingly accurate as the number of observations from which it is produced increases.

Of particular importance are estimators for the mean and variance, called the sample mean and sample variance. The sample mean of a random sample $X_1,\ldots,X_n$ is given by:

Eq. 9-7 $$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

The sample variance is given by:

Eq. 9-8 $$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})^2$$

While the ability to estimate means and variances is important, it is also important to determine the range of possible errors in such estimates. In order to compute the expected error for an estimate, one introduces the notion of a confidence interval. We say that an interval $[x - \delta , x + \delta]$ is a $(1 - \alpha)$ confidence interval for a parameter θ if the probability that the true value of θ lies in the interval is $1 - \alpha$. Thus, if the .95 confidence interval for the mean packet delay is [1.07, 1.10] seconds, there is a 95% chance that the "true" value of the mean delay is greater than or equal to 1.07 seconds and less than or equal to 1.10 seconds.

Distributions

There are three probability distributions which will be particularly important in the following material. The first is the normal distribution. For a normally distributed random variable X, the probability that X assumes values in $(x, x + dx]$ is given by:

Eq. 9-9 $$f_X(x)dx = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\} dx$$

where $f_X(x)$ is the probability density function. Here μ is the mean and σ^2 the variance of X. The standard notation for a normal random variable of mean μ and variance σ^2 is $N(\mu, \sigma^2)$. A special and important case is the standard normal distribution, N(0,1), whose probability density function is given by:

Eq. 9-10 $$f_X(x)dx = \frac{1}{\sqrt{2\pi}} \exp\left\{\frac{-x^2}{2}\right\} dx$$

A second important distribution is the chi-square distribution [4]. If one has a family of n independent random variables $\{U_i\}$ drawn from N(0,1) then one can form

Eq. 9-11 $$\chi^2_n = \sum_{i=1}^{n} U_i^2$$

The distribution of χ^2_n is called the chi-square distribution with n degrees of freedom.

A third important distribution is the Student-t distribution [4]. It can be shown that the sample mean, $\bar{X}$, of n independent observations drawn from $N(\mu, \sigma^2)$ is normally distributed according to $N(\mu, \sigma^2/n)$. Therefore

Eq. 9-12 $$U = \frac{(\bar{X} - \mu)\sqrt{n}}{\sigma}$$

is from N(0,1). In real experiments, however, one does not know σ^2 so that it must be estimated from the observed sample. Thus, one replaces σ in the above formula with the sample standard deviation. One therefore defines a new statistic,

Eq. 9-13 $$t_{n-1} = \frac{(\bar{X} - \mu)\sqrt{n}}{s}$$

the Student-t distribution with n-1 degrees of freedom.

Theorems

The concepts which will most directly concern us in analyzing simulation experiments are those of the sample mean and the confidence interval. There are strong theoretical reasons for believing that the sample mean is normally distributed about the true system mean. If one knows the variance of this distribution, one can then compute a range of values within which the system mean can be expected to fall. The computation of confidence intervals consequently has two important preconditions. First, the sample mean must be normally distributed. Second, one must have a method of computing the sample variance of the sample mean. It turns out that both of

these preconditions are easily met for certain types of data (although not for the sorts of data we will typically encounter in simulation experiments). The following two theorems express this fact.

Theorem 9-1 Given that $\{X_i : i=1,\ldots, n\}$ is a sequence of independent observations on a random variable X with mean μ and finite variance σ^2 and that $\bar{X}_n$ and s^2 are the sample mean and sample variance then

Eq. 9-14 (a) $E[\bar{X}_n] = E[X] = \mu$

(b) $E[s^2] = Var[X] = \sigma^2$

(c) $Var[\bar{X}_n] = Var[X]/n = \sigma^2/n$

(d) $Z = (\bar{X}_n - \mu)/ \sigma / \sqrt{n}$, for large n, has the standard normal distribution N(0,1) [1].

Thus the sample mean and sample variance are unbiased estimators of the mean and variance [1]. From c, one can also conclude that the sample mean is a consistent estimator of the mean.

Theorem 9-2 Given $X_1,\ldots, X_n$, a random sample drawn from X whose mean is μ and whose variance σ^2. Suppose that X is normally distributed or that n is large enough so that $\bar{X}_n$ can be considered to be normally distributed by Theorem 1d above. Then the $(1-\alpha)$ <u>confidence interval</u> is given by

Eq. 9-15

$$\bar{X}_n \pm E$$

where

$$E = z_{\frac{\alpha}{2}} \sigma/\sqrt{n}$$

$z_{\frac{\alpha}{2}}$ is the $100(1-\frac{\alpha}{2})$ percentile of the standard normal distribution. In other words, if X is distributed according to N(0,1) then:

Eq. 9-16

$$\text{Prob}\,[X > z_{\frac{\alpha}{2}}] = \frac{\alpha}{2}$$

If neither the mean nor variance are known and are therefore estimated by the sample mean and sample variance, then the $100(1-\alpha)\%$ confidence interval is given by

Eq. 9-17

$$X_n \pm E$$

where

$$E = t_{\frac{\alpha}{2};n-1} \; s/\sqrt{n}$$

$t_{\frac{\alpha}{2};n-1}$ is the $100(1-\frac{\alpha}{2})$ percentile of the Student-t distribution with n-1 degrees of freedom [1].

Hypothesis Testing

In the following material, we will frequently be concerned with testing various hypotheses about some underlying distribution. For example, in testing the random number generator, one obtains a sequence of observations $X_1,\ldots,X_n$ and tests the hypothesis that the $\{X_i\}$ are uniformly distributed on the interval [0,1].

One can think of a given sequence of observations, $\{Y_1,\ldots, Y_n\}$, as a point, $\vec{Y}$, in the "sample space," S, of all possible sequences of observations. The sample space is divided into two domains, a rejection region, R, and an acceptance region, S-R, such that if the sample point $\vec{Y}$ is in R then one rejects H_0, the hypothesis being tested. If $\vec{Y}$ is in S-R then one accepts H_0 [4].

Of course, there is always a possibility that a single sample point will be in the rejection region R even if H_0 is actually true. For example, there is always a possibility that a finite set of observations on a truly "random" number generator will produce nothing but zeroes. If this occurs, one would probably (although wrongly) reject the generator as being non-random. This possibility is intimately related to the notion of the significance level or size of the test. One says that a specific test of H_0 is of size α if the probability that $\vec{Y}$ falls

in R when H_0, in fact, obtains is α . Thus, when testing H_0 at the .10 level, there is a 10% chance of rejecting H_0 when it does, in reality, hold.

9.2 Statistical Aspects of Simulation Experiments

There are a number of problems pertaining to the analysis of simulation data. Some of these are quite fundamental in nature; others are more practical. An example of the former is the problem of determining the conditions under which it is sensible to attempt parameter estimation from simulations. An example of a practical problem is that of determining the conditions under which it is feasible to attempt such estimation from a single simulation run. An even more practical problem is that of producing confidence intervals for system parameters from simulation data.

The central topic of this section is a review of these issues. Emphasis should be placed upon the word "issue" inasmuch as only a few of the problems discussed admit of definite answers. We have two general purposes in presenting this material. The first is to convey a sense of the difficulties inherent in what may seem, at first glance, to be a rather straightforward problem, namely, computing average quantities associated with simulation models. The second purpose is to motivate the particular algorithms described in subsequent sections. The material presented here is taken primarily from the excellent discussion in [2].

Steady State Distributions

The general problem in the analysis of simulation experiments may be stated as follows. One is given a series of observations $\{X_i : 1 \leq i \leq n\}$ made upon a simulation model. For example, the $\{X_i\}$ might be a sequence of end-to-end packet delays, recorded when each packet leaves the simulated network. A typical goal might then be to compute an estimate for the mean packet delay and a confidence interval around that estimate. The heart of the problem is that one cannot simply apply the formulas contained in the theorems of the preceding section to the data $\{X_i\}$ in a straightforward manner.

There is one important and immediately obvious difference between a sequence of packet delays and the sets of independent observations assumed in the theorems of the previous section. Namely, there is a definite ordering associated with the sequence of packet delays. Thus, if $i>j$ then delay X_i was recorded *after* delay X_j. This ordering is neither arbitrary nor meaningless. Two packets which entered the network at the same time are more likely to have encountered similar network conditions than are two packets which entered the network at greatly different times, other things being equal. As an example, consider three packets whose delays were X_i, X_j and X_k and assume that these packets traversed the same route. If $|i-j| << |i-k|$, then there is a fair probability that $|X_i - X_j| << |X_i - X_k|$. The series of packet delays is both *ordered* and *correlated* in time.

These notions may be given a formal expression via a number of mathematical concepts. We form a sequence of n random variables, $\{X_i\}$, indexed on i. We express the values assumed by the $\{X_i\}$ in a particular experiment as $\{x_i\}$. (Thus, in the sequence of packet delays, x_5 corresponds to the value that the random variable X_5 assumed in the particular experiment, i.e., x_5 is the delay seen by the fifth packet.) We call the sequence of random variables $\{X_i\}$ a stochastic process.

For some stochastic processes the $\{X_i\}$ are independent and identically distributed (i.i.d.). Independence is used here in the formal sense of statistical independence. That is, if the $\{X_i\}$ are independent, then a particular observation made upon X_m yields no information about the expected value of an observation made on X_n if m and n are not equal. Identically distributed means that the probability distribution functions for all X_i are the same. If the $\{X_i\}$ are i.i.d., then a single function, namely, the probability distribution function associated with any particular X_j, yields a complete description of the system.

As was discussed above, we do not, in general, have statistical independence for the sequence of packet delays. Thus, even if all X_j were to be identically distributed, more information than the distribution function would be required in order to describe the entire sequence $\{X_i\}$. For example, since information about a given observation on X_5 yields information

about the expected value for X_6, we also require the conditional probability distribution, i.e., the distribution of X_6 given X_5, in order to characterize the process.

This discussion, although abstract, is not academic. The fact that the series of packet delays has internal correlations, i.e., is autocorrelated, strongly impacts the attempt to estimate both the mean delay and a confidence interval for the mean. To see this, we recall the important conclusions of the theorems of section 9.1. In particular, for independent data, we were able to conclude that the sample mean and sample variance were unbiased and consistent estimators of the mean and variance. In addition, we were able to find expressions for the sample mean, the sample variance, and for confidence intervals about the mean. For correlated data, those theorems do not apply. Hence, a more detailed discussion of estimation for autocorrelated processes is required.

To begin, we let $\bar{X}_n$ be a random variable computed as the mean of n successive observations on $\{X_i\}$, and $\text{var}(\bar{X}_n)$ be the variance of $\bar{X}_n$. One can then form the mean-square error:

Eq. 9-18 $$\text{MSE}(\bar{X}_n) = \text{var}(\bar{X}_n) + (E[\bar{X}_n] - \mu)^2$$

The importance of the mean square error lies in the fact that if the limit of the mean square error approaches zero as n approaches infinity, then $\bar{X}_n$ is a consistent estimator of μ, the "real" mean delay [2]. This result can be understood intuitively in that the convergence of the second addend in Eq. 9-18 to zero implies that $E(\bar{X}_n)$ really does approximate μ, for large n; the convergence of the first addend guarantees increasingly small confidence intervals about μ, as n increases.

If the $\{X_i\}$ are i.i.d., if each X_j is distributed according to X, and if the variance of X is finite, then one knows from Theorem 9-1 that

Eq. 9-19 $$E(\bar{X}_n) = \mu$$

and that

Eq. 9-20 $$var(\bar{X}_n) = var(X)/n$$

Thus, for i.i.d. $\{X_i\}$, $var(\bar{X}_n) \rightarrow 0$, which implies $MSE \rightarrow 0$. Thus, we have convergence in the mean which implies convergence in probability. This convergence is reflected in decreasing confidence intervals as n increases. Unfortunately, for correlated time series data, Theorem 9-1 does not hold and

convergence of the mean square error to zero is not obvious. It is consequently clear that if one does not have independent $\{X_i\}$, it is not guaranteed that one's estimates are consistent.

The preceding discussion is meant to be merely illustrative of the fact that the ability to make meaningful estimates of the parameters belonging to a given system from a finite set of observations upon that system is not to be assumed. The question then arises as to whether there exist general properties belonging to stochastic processes which do allow for parameter estimation. Such properties do exist; they are stationarity and ergodicity [2].

For a given random variable, X, we define the probability distribution function, F_X, such that

Eq. 9-21 $$F_X(x) = \text{Prob}\,[X \leq x]$$

The distribution function, which is the integral of the density function, completely characterizes the particular random variable. Given n random variables, $\{X_i\}$, the joint distribution, $F_{X_1,..,X_n}$, is defined such that

Eq. 9-22 $$F_{X_1,\cdots,X_n}(x_1,\cdots,x_n) = \text{Prob}\,[X_1 \leq x_1 \cap \cdots \cap X_n \leq x_n]$$

If we are given a stochastic process $\{X_i\}$ we can select, for arbitrary $i_1,\ldots,i_n$, $X_{i_1},X_{i_2},\ldots,X_{i_n}$, and form the joint distribution function. If one has

Eq. 9-23 $$F_{X_{i_1},\cdots,X_{i_n}}(x_1,\cdots,x_n) = F_{X_{i_1+s},\cdots,X_{i_n+s}}(x_1,\cdots,x_n)$$

the stochastic process is said to possess strict stationarity. In other words, the joint distribution function is invariant under time translation. In a physical sense, this means that one can enter the stochastic sequence at any point and observe the same distributions that one observes if one enters at some other point. If the probability distribution for a particular parameter is unchanging with time, then the mean for that parameter can be considered as a performance measure (albeit, not necessarily useful) of the system. This is because stationarity implies that the mean for that quantity, as estimated from the sample mean, is a constant. It is this property which enables us to define a parameter as a performance measure. To see this, consider a trivial example where stationarity does not obtain,

i.e., an M/M/1 queueing system where the offered traffic exceeds the capacity of the server. In this case mean packet delay is a useless notion. If one observes 100 packets, beginning at t = 0, and computes the sample mean, one will almost certainly get an estimate for the mean which is much smaller than it one had observed the system beginning at t = 1000. The expected sample mean based upon 100 observations is therefore dependent upon which 100 packets one observes. Thus, the distribution function for the mean delay is time dependent and the system is not stationary. For stationary queueing systems, e.g., an M/M/1 system with a server utilization less than 1, the expected value of the sample mean delay based upon 100 observations is not dependent upon which 100 packets one observes.

Stationarity is only one important criterion. Given $\{X_i\}$, one can define the autocovariance function $\{R_s\}$ where

Eq. 9-24 $$R_k = E[(X_i - \mu)(X_{i+k} - \mu)]$$

In other words, a given R_k measures the degree to which observations k units apart are correlated. If our observations correspond to packet delays, then the autocovariance function tells us the amount of information that a given packet delay yields about the expected delay for the kth subsequent packet.

For most queueing networks it is reasonable to expect that $R_k \dashrightarrow 0$ as $k \dashrightarrow \infty$. In other words, there is very little correlation between the delays experienced by packets which transit the network many time units apart.

If the $\{R_s\}$ converges to zero with s fast enough, then one can show that

Eq. 9-25
$$\lim_{n\dashrightarrow\infty} n * \mathrm{MSE}(\bar{X}_n) = \lim_{n\dashrightarrow\infty} n * \mathrm{var}(\bar{X}_n) = V$$

where

Eq. 9-26
$$V = \sum_{s=-\infty}^{\infty} R_s$$

If this property holds, $\{X_i\}$ is said to be ergodic in the mean [2]. The importance of ergodicity cannot be overstated. In order for the mean square error to converge to zero, $\bar{X}_n$ must converge to μ and the variance of X_n must converge to zero. It can be shown that, under very general conditions, the sample mean chosen from the steady state is an unbiased estimator of the true mean [2]. Thus, the mean square error will converge to zero if the sample variance of the sample mean so converges. This will happen if the system is ergodic in the mean. Thus, the

property of ergodicity is related to the ability to produce consistent estimators inasmuch as the convergence of the mean square error to zero is a sufficient condition for consistency. For reasons which will be discussed below, if the variance of the sample mean decreases as the number of observations increases, the size of the confidence intervals about the mean will also decrease. Thus, additional observations upon ergodic systems will produce better interval estimates for the "true" system mean. For systems which are not ergodic, additional observations do not necessarily yield additional information. An example of a non-ergodic system is a simple queueing system with unit activity level [2]. (Parenthetically, one may observe that $R_0 = \text{var}(X)$ and that, for an i.i.d. stochastic process, $R_k = 0$ for $k>0$. Thus, Theorem 9-1 above becomes a special case of Eq. 9-25, appropriate when the observations are i.i.d.)

The above discussion defines the circumstances under which one can make meaningful estimates for system variables in simulation experiments. As such, it generalizes the discussion of section 9.1 to correlated stochastic processes. There is another aspect of section 9.1 which has not thus far been addressed. In particular, having determined the conditions under which one <u>can</u> construct point and interval estimates, one must determine <u>how</u> to do this. The theorems in section 9.1 which provide formulas for interval estimates assume independent

observations and infer that the sample mean $\bar{X}_n$ is normally distributed about the true mean. These theorems also provide formulas for the variance of this normal distribution. If the sample mean is normally distributed about some value with variance σ^2, then one can determine a range of values within which the sample mean will fall with a specified probability. Thus, the condition of normality allows one to estimate confidence intervals. For stochastic processes, to which the theorems of section 9.1 do not apply, one is consequently left with two questions.

1) How does one compute $\text{var}(\bar{X}_n)$?

2) Is the sample mean $\bar{X}_n$ normally distributed for some suitably large n?

The variance of the sample mean can be estimated from the autocovariance structure of the data. Unfortunately, this computation is sufficiently complex so as to be virtually infeasible. In addition, such a calculation would be misguided in that one ends up with an excessive amount of information [4]. We therefore seek to develop efficient techniques to estimate the variance. These will be discussed in the next chapter when we specify the particular algorithms which will be used for interval estimation. For the time being, we concentrate on question 2. We note, once again, the significance of normality. If $\bar{X}_n - \mu$

has the normal distribution, then the 1-α confidence interval can be computed according to

Eq. 9-27 $$X_n \pm E$$

where

Eq. 9-28 $$E = \sqrt{\text{var}(X_n)}\, z_{\frac{\alpha}{2}}$$

(From Equation 9-28, one can understand the practical significance of ergodicity. If $\text{var}(\bar{X}_n)$ decreases with n, then the computed confidence intervals will become progressively narrow as the number of observations increases.) If Equation 9-28 is used and the $\bar{X}_n$ are not normally distributed, then incorrect interval estimates will result. Asymptotic normality has been demonstrated for certain analytically tractable queueing models [2]. Unfortunately, such proofs have limited relevance for complex queueing networks. Kleijen [3] quotes a form of the central limit theorem which states that asymptotic normality holds for "r-dependent strictly stationary" stochastic processes. An r-dependent stochastic process $\{X_i\}$ is defined as one for which X_j and X_{j+s} are uncorrelated for all j if s>r.

Transient Analysis

The above discussion has ignored a central aspect of simulation experimentation, namely, the effect that the initial conditions of the simulation have upon the estimates of system parameters. In fact, the notation used in the previous section has been somewhat casual. We have assumed the existence of some sort of steady state distribution function, F_X, and that the sequence of packet delays $\{X_1,\ldots,X_n\}$ were drawn from the steady state distribution. In fact, reality is considerably more complex. What one has in general is not a single distribution function, but a conditional F_X given some starting condition S_0, or $F_{X|S_0}$. Thus, given observations $X_1,\ldots,X_n$, the expected value of the conditional sample mean $E(\bar{X}_n|S_0)$ is given not by μ but by:

Eq. 9-29 $$E[\bar{X}_n|S_0] = \mu + b_{n|S_0}$$

The second summand is called the bias. Assume one runs J independent replications of an experiment, with each replication consisting of n observations. If one forms the sample mean, $\bar{X}_{n,j}$, for each replication, and computes $\tilde{X}_{n,J}$ as the sample average of the $\bar{X}_{n,j}$, one has

Eq. 9-30 $$MSE(\bar{X}_{n,J}|S_0) = \frac{var(\bar{X}_n|S_0)}{J} + b^2_{n|S_0}$$

Under certain mild conditions it can be shown that

Eq. 9-31 $$\lim_{n\to\infty} b_{n|S_0} = 0$$

Thus $\bar{X}_n$ is an unbiased estimator of μ as n increases (within a replication). In addition, one can show, for a large class of problems, that the consistency property also obtains [2].

The preceding paragraph has been concerned with the convergence of a conditional distribution to a steady state distribution. There is another type of convergence which is also important, namely, convergence of the distribution of $\bar{X}_n$ to the normal distribution. The lack of certainty associated with statements about such convergence under steady state conditions for a given simulation run were discussed above. Here we are concerned with the degree to which the initial conditions impact such convergence. Once again, we assume J independent replications of the experiment each with n observations. We form the sample mean for each replication, $\bar{X}_{n,j}$, and the mean of the means, $\tilde{X}_{n,J}$. Then, since the J replications are i.i.d., $\tilde{X}_{n,J}$ will be normally distributed for J sufficiently large. The problem is that it will be distributed with mean $\mu + b_{n|S_0}$ and

not with mean μ . That is, one has data which is normally distributed about the wrong mean. Therefore, an experiment consisting of many independent replications must still be constructed so as to eliminate the bias on each individual run. The convergence to normality across simulation runs is to be distinguished from such convergence within a run. In dealing with the latter type of convergence, we are concerned with the question of whether or not the sample mean for a given replication (as opposed to the mean of the means) is chosen from a normal distribution. Convergence to normality within a run is essential for interval estimation on a single simulation run and has been demonstrated to hold for G/G/1 queueing systems. For more complex systems, it is not clear that such convergence always obtains [2].

The problem of bias due to initial conditions appears to be the area of simulation analysis which is the least understood and the area where practical conclusions are scarcest. The single general conclusion is that the number of observations must be sufficiently large so that the bias introduced is negligible. The practical conclusion that is typically drawn and implemented by researchers is that bias should be reduced by discarding some number of initial observations. (Unfortunately, a number of authors have indicated that this approach can increase the mean square error [2].)

One may summarize the principal conclusions of this section as follows. From a sequence of observations taken upon a simulation model, one wishes to make point and interval estimates of various performance measures. The two system properties which allow meaningful estimation are stationarity and ergodicity. The former ensures that it is possible to define performance parameters which are characteristic of the system over its entire life; the latter guarantees that additional observations on the system produce more accurate results. While ergodicity and stationarity are necessary theoretical properties, they are hard to demonstrate for complex systems. Furthermore, they do not provide practical guidance in the computation of point and interval estimates. In order to effect such estimates, one requires two types of convergence. The first is convergence of the system to the steady state distribution; the second is the convergence of sample means to a normal distribution. The former is frequently facilitated via judicious choice of initial conditions; the latter is difficult to demonstrate. The key practical prerequisite for interval estimation is the ability to estimate the sample variance of the sample mean. One of the key tasks of any analysis algorithm is to accomplish this calculation in an efficient manner.

References

1. Allen, Arnold O. Probability, Statistics, and Queueing Theory, Academic Press, 1978.

2. Fishman, George, Principles of Discrete Event Simulation, Wiley-Interscience, 1978.

3. Kleijnen, Jack P.C., Statistical Techniques in Simulation, Marcel Dekker Inc., New York, 1974.

4. Kobayashi, Hisashi, Modeling and Analysis, Addison-Wesley, 1978.

10. SIMULATION ANALYSIS PACKAGE

The previous chapter was largely concerned with the theoretical issues related to the analysis of simulation experiments. In this section, we describe the practical approaches that we have implemented in our analysis package in order to effect point and interval estimates.

10.1 Discussion of Algorithms

Given the discussion of Chapter 9, we can formulate the outline of an analysis algorithm as follows:

1. Make m observations $X_1,...X_m$ of some performance measure belonging to the simulation model.

2. Discard those observations, $X_1,...X_j$, $j<m$, corresponding to the transient period and relabel the steady state observations as $X_1,...X_n$, $n=m-j$.

3. Compute the sample mean as

Eq. 10-1
$$\bar{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i$$

4. Estimate $var(\bar{X}_n)$.

5. Estimate a $1-\alpha$ confidence interval for the mean as

Eq. 10-2 $$\bar{X}_n \pm E$$

where

Eq. 10-3 $$E = \sqrt{var(\bar{X}_n)}\, t_{\frac{\alpha}{2};\, df}$$

From the above statement of a program, the central problems which the analysis algorithms must solve become clear. There is no obvious means by which to perform Task 2. Additionally, Task 4, the computation of the variance of the sample mean, is a necessary input to Task 5, yet there is no obvious means by which to perform the calculation. Were our observations independent, we would rely on Theorem 9-1 which relates the variance of a set of n independent observations to the variance of the sample mean based upon those observations. However, for correlated data, Theorem 9-1 is not applicable. Yet, even given $var(\bar{X}_n)$, we still cannot complete Task 5, the computation of an interval estimate. Since we use the sample variance, we must use the Student-t distribution. It will be recalled that there are an infinite number of Student-t distributions, indexed by degrees of freedom.

Were our data independent, the number of degrees of freedom would be, by Theorem 9-2, n-1. But, once again, Theorem 9-2 is not applicable to our data. Thus, the central problems to be addressed in the analysis routines are: (1) the computation of the sample variance of the sample mean, (2) the computation of the number of degrees of freedom, and (3) the analysis of the transient period. We defer a discussion of the transient to the end of this section and, in what follows immediately, assume all observations are drawn from the steady state distribution.

Before describing the detailed algorithms used for simulation analysis, a few words are in order about those methods which we decided not to implement. An obvious means of analyzing simulations is to run many replications of a simulation experiment, each with a different random number stream. Then, the mean delay for each run does constitute an independent observation, and the classical statistical formulas are applicable to the set of sample means. While this fact certainly recommends the method of independent replications, we intend to use this procedure only as a last resort. There are several reasons for this decision. First, in each simulation run there is a transient period during which the observations are biased toward the initial conditions. When analyzing a simulation experiment, one typically discards data collected during the transient period. Thus, the transient period constitutes

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

"wasted" simulation (and therefore CPU) time. On ten replications of a single experiment, there will be ten such wasted periods. If data can be obtained on a single simulation run, then the amount of wasted simulation time is 90% less. This increase in the ratio of productive simulation time to CPU time is the strongest argument for a method that requires a single simulation run. However, it is not the only such argument. There is a certain overhead incurred in each simulation run apart from that associated with the transient. This overhead is measured both in CPU time and in analyst's time. Additionally, reliance on many short simulation runs introduces the risk that long-term processes present in the simulation will be obscured. Finally, at least one author suggests that asymptotic normality is a better assumption for a single long simulation run than for many shorter runs [4].

A second method for producing point and interval estimates which was not implemented is widely discussed in the literature. This is the so-called regenerative approach [3]. A regeneration point of a simulation is defined as a particular state such that whenever the system is in that state, all future observations on the simulation are independent of its prior history. The intervals between regeneration points are called epochs and have the nice feature that observations made in different epochs are independent and identically distributed. A simple example of a

regeneration process is an M/M/1 queueing system for which one can define the point at which a customer arrives at the empty and idle state as a regeneration point.

Regeneration processes possess a number of properties which greatly facilitate data analysis. However, not all systems possess the regenerative property. Furthermore, for complex systems, the expected simulation time between regeneration points may be too large to permit sufficient observations within a reasonable amount of elapsed time. Thus, in a simulated network with dozens of queues and servers the probability associated with any particular state, where a state is defined as a specific assignment of customers to queues, may be so negligible that the expected size of an epoch is excessively large.

There is a certain advantageous feature that both the regenerative and independent replication methods have which the algorithms that we have implemented do not possess. These methods are exact in the sense that the observations made on a given replication or in a particular epoch are truly independent. Each of the methods which we have implemented involves some approximation and is not necessarily applicable to all types of data. Thus, in all cases, prior analysis of the structure of the data is required to determine the applicable method.

10.1.1 Batch Means

The first method that we have implemented is known as the batch means method. The approach is conceptually similar to the regenerative approach and is, in its mathematical details, related to the independent replications method. Batch means has an advantage over the latter in that only one transient period is incurred. Its disadvantage is that, unlike the independent replications method or the regenerative approach, the assumption of independent observations is an approximation.

The basic idea behind batch means is that the further apart in time two observations on the simulation are made, the more "independent" they become. This is a simple statement of the obvious fact that packets transiting a network at not greatly different simulated times are more likely to encounter similar network conditions than are two packets transiting at greatly different times. (Note that this is not necessarily the case if there are long-term periodic processes occurring in the simulation.) Hence if one divides one's observations on the simulation into large groups, or batches, such that if $i<j$ than any packets in batch i left the network before all packets in batch j, one can treat observations on different batches as being independent and identically distributed. Thus, one can compute the mean of the observations in each batch, treat each batch average as an independent observation and apply the classical statistical formulas to the set of batch means.

This approach can be represented mathematically as follows. For a set of observations, $\{X_1,\ldots X_n\}$, and a batch size, m, compute the batch averages as:

Eq. 10-4 $$Y_i = \frac{1}{m} \sum_{j=1}^{m} X_{m(i-1)+j} \qquad i=1,\ldots,k$$

where k is the greatest integer in n/m. Since each batch average is an "independent" observation, we can apply Theorem 9-1 as follows. If we compute the sample variance of the batch means:

Eq. 10-5 $$s_Y^2 = \frac{1}{k-1} \sum_{i=1}^{k} (Y_i - \overline{X}_n)^2$$

then we can represent (by Theorem 9-1)

Eq. 10-6 $$\mathrm{var}(\overline{X}_n) = s_Y^2/k$$

Finally, we can compute a $1-\alpha$ interval estimate as

Eq. 10-7 $$X_n \pm \frac{t_{\alpha/2;k-1}\, s_Y}{\sqrt{k}}$$

The central computational difficulty with the batch means method is the choice of the appropriate batch size, m. There is

an important tradeoff involved in the selection of a batch size. If the batch size is too small, then the approximation that the batch means are independent is inevitably a poor one. As the batch size grows, then the number of independent observations becomes smaller. This may be reflected in a larger estimate for var $(\bar{X}_n)$.

We have implemented an iterative approach which is designed to select a small batch size which, nevertheless, ensures independent observations [3]. Namely, we begin with a batch size of 1 observation per batch. Each iteration of the algorithm results in a doubled batch size. For each iteration, we compute the batch means and test for independence of the batch means at the α level. The algorithm proceeds until the test for independence is successful or until there are fewer than 8 batches. If the test for independence succeeds and there are more than eight batches, we can compute point and interval estimates according to Equations 10-4 through 10-7 above. If there are fewer than eight batches, we cannot compute such estimates and must instead run the simulation for a longer amount of time.

The choice of eight as the minimum number of batches is dictated by the particular test for independence that we have implemented. This test examines the hypothesis, H_0, that the observations in the sequence $\{Y_i\}$ are independent. The test statistic used is:

Eq. 10-8

$$C_k = 1 - \frac{\sum_{j=1}^{k-1} (Y_j - Y_{j+1})^2}{2 \sum_{j=1}^{k} (Y_j - Y_k)^2}$$

If the $\{Y_i\}$ have a monotonically decreasing autocovariance function we accept H_0 at the β significance level if

Eq. 10-9

$$C_k \leq c(2\beta) \sqrt{(k-2)/(k^2-1)}$$

where $c(2\beta)$ is defined according to

Eq. 10-10

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{c(\alpha)} e^{-x^2/2} \, dx = 1 - \alpha/2$$

If the autocovariance function is not monotonically decreasing then we accept H_0 if

Eq. 10-11

$$C_k \leq c(\beta) \sqrt{(k-2)/(k^2-1)}$$

The batch means algorithm has been implemented as a SIMULA class (see section 10-3). A sample output is displayed in Fig. 10-1. Each line contains the results of an iteration of the algorithm. The column labelled "c" contains the value of the test statistic C_k. The column labeled "critical value" contains the value to which we are comparing C_k (right hand side Eq. 10-9). When c is less than the critical value, we accept the hypothesis that the batches defined on that iteration are independent. Thus, in Figure 10-1, 64 was the minimum batch size which allowed the assumption that the batch means were statistically independent. In this case, there were 125 batches, the sample variance of the sample mean was .019, and the .95 interval estimate for the mean was [1.89, 2.43].

In order to test our implementation of the batch means method, we built an M/M/1 queueing simulator. One hundred replications of the simulator were run, each with a different and independent random number stream. (On each replication the seed for the random generator was chosen to be the last random number generated in the previous replication.) For each simulation replication, the server utilization was .75 (average arrival rate of 1 per second, .75 second average service time).

0.95 interval estimation----batch method

sample mean= 2.16266

no. of batches	no. of obs. per batch	sample variance of sample mean	0.95 interval estimate lower	upper	c	critical value
8000	1	0.000811	2.106804	2.218510	0.929	0.018
4000	2	0.001565	2.085072	2.240242	0.898	0.026
2000	4	0.002974	2.055680	2.269633	0.825	0.037
1000	8	0.005437	2.017936	2.307378	0.692	0.052
500	16	0.009143	1.974748	2.350566	0.507	0.073
250	32	0.013710	1.931997	2.393317	0.275	0.104
125	64	0.018601	1.892654	2.432660	0.055	0.146
62	128	0.020823	1.874040	2.451274	0.053	0.206
31	256	0.025882	1.834025	2.491289	*****	0.286
15	512	0.028223	1.802246	2.523068	*****	0.396

125 batches suffice for independence at 0.05 level

Figure 10-1 Sample Output -- Batch Means Method

Our estimate of the transient period for the simulation was based on analytic results. According to a diffusion model of G/G/1 systems due to Kobayashi [5], the transient period of such systems is equal to

Eq: 10-12 $$T_0 = 5 \frac{E[S](C_s^2 + C_a^2\rho)}{(1-\rho)^2}$$

where C_s^2 and C_a^2 are the squared coefficients of variation of the service and interarrival times, respectively. For an M/M/1 system with a utilization of .75 and a service time of .75 seconds, the transient period is therefore computed as roughly 100 seconds. In order to be safe, for each replication of the M/M/1 simulator, we discarded statistics on the first 1000 arrivals into the system.

For each replication of the system we accumulated the queueing delays experienced by 8000 packets and ran the batch means algorithm over these results. The decision to accumulate data for a specified number of customers as opposed to a specified amount of time was not an arbitrary one. If one accumulates data for a specified amount of simulated time, then the number of data points accumulated is a random variable. The

estimate of average queueing delay, i.e., the ratio of total delay experienced by all customers to the number of customers, computed as the ratio of two random variables, is, according to statistical theory, a biased estimate of the mean delay. Rather than apply the theory applicable to ratio estimation, we instead fixed the number of observations collected (as opposed to the time during which the observations were collected). This means that the denominator in the computation of mean delay is not a random variable.

The results of the analysis for the first 10 replications are summarized in Figure 10-2. Figure 10-2 shows, for each of the 10 replications, the analysis results for the first iteration on which we accepted the hypothesis, at the .05 level, of independent batch means. It will be observed that, for the first replication, 8000 observations were not sufficient to produce an interval estimate. That is, on that replication, the test for independence of the batch means was not successful for any iteration in which there were eight or more batches.

According to elementary queueing theory, the expected queueing delay for an M/M/1 system with a server utilization of .75 is 2.25. Since we computed .95 interval estimates, we expect that 2.25 should be within the computed interval on 95% of the replications for which we are able to make an estimate. (Reference to Figure 10-2 reveals that on replication 7 the

replication	batch size	$\bar{X}_n$	var($\bar{X}_n$)	.95 interval estimate
1	---	3.56	---	---
2	128	2.20	.039	[1.81 , 2.60]
3	128	2.33	.028	[2.00 , 2.66]
4	64	2.09	.022	[1.80 , 2.38]
5	128	2.34	.056	[1.87 , 2.81]
6	64	2.16	.019	[1.89 , 2.43]
7	128	2.99	.102	[2.35 , 3.63]
8	128	2.43	.058	[1.95 , 2.91]
9	64	2.09	.019	[1.82 , 2.36]
10	256	2.25	.044	[1.82 , 2.68]

Figure 10-2 Summary of Batch Means Analysis for First 10 Replications of an M/M/1 Simulation

interval estimate did not contain the system mean.) We were able to make interval estimates on 97 of the 100 replications. On 93 of these 97 replications, our computed .95 interval estimate for the mean contained the theoretical mean delay of 2.25. Our "coverage rate" was therefore 93/97 or 96%, which is very close to the expected coverage rate of 95%.

As an empirical method of checking the choice of batch size we introduce the normalized autocovariance function or "correlogram." It will be recalled that the autocovariance function of a time series $\{X_i\}$ is given by:

Eq. 10-13 $$R_k = E[(X_i - \mu)(X_{i+k} - \mu)]$$

and that R_k measures the covariance of observations which are k observations apart. If one normalizes R_k by dividing by R_0, one obtains the autocorrelation sequence

Eq. 10-14 $$\rho_k = \frac{R_k}{R_0}$$

A given ρ_k is called the serial correlation coefficient of lag k. Clearly ρ_0 is equal to unity, which is merely the statement that an observation is always perfectly correlated with

itself. ρ_k can assume any value between -1 and 1. Values of ρ_k near zero indicate very little correlation.

For the time series of steady-state queueing delays produced by an M/M/1 system, one expects that the correlation between observations closely spaced in time will be high (near one) and that between observations distant in time will be low (near zero). This should result in a series of ρ_k which are monotonically decreasing from 1 to zero with k. In order to see this, we introduce the sample autocorrelation sequence or correlogram computed by

Eq. 10-15
$$\hat{\rho}_k = \frac{\frac{1}{n-k}\sum_{i=1}^{n}(X_i - \bar{X}_n)(X_{i+k} - \bar{X}_n)}{\frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2}$$

The correlogram is an asymptotically unbiased estimator of ρ_k [5].

One can use the correlogram to estimate the batch size in the following manner. If the serial correlation coefficients are monotonically decreasing with the lag, then, for some lag i and for all higher lags, the value of ρ_k will be close to zero. This means that the jth observation and the (j+i)th observation

are approximately statistically independent for all j. Thus, one would expect that i should roughly approximate the batch size.

If one uses (as we do) some other method to choose a batch size, one can use the correlogram to verify that the computed batch means are independent. If the correlogram of independent data is computed, then the correlation coefficients for all lags (with the exception of lag 1) should be near zero. In addition, there should be no discernible bias in the sequence of correlation coefficients toward either positive or negative values. In other words, the coefficients should be distributed more or less randomly about and close to zero.

The correlogram lends itself nicely to graphical analysis. Figure 10-3 shows the correlogram, computed out to lag 128, of queueing delays for a sample run of an M/M/1 simulation. As can be seen, the serial correlation coefficients are roughly zero for lags greater than 120. It will be observed from Figure 10-2 that the batch means method generally selected a batch size of 128. The results produced by the correlogram and the batch means independence test are therefore consistent.

Unfortunately, the batch means method is not appropriate for all varieties of data. An important and interesting example is provided by measurements of delays for packets traversing an actual IMP. In order to produce such data, we generated messages

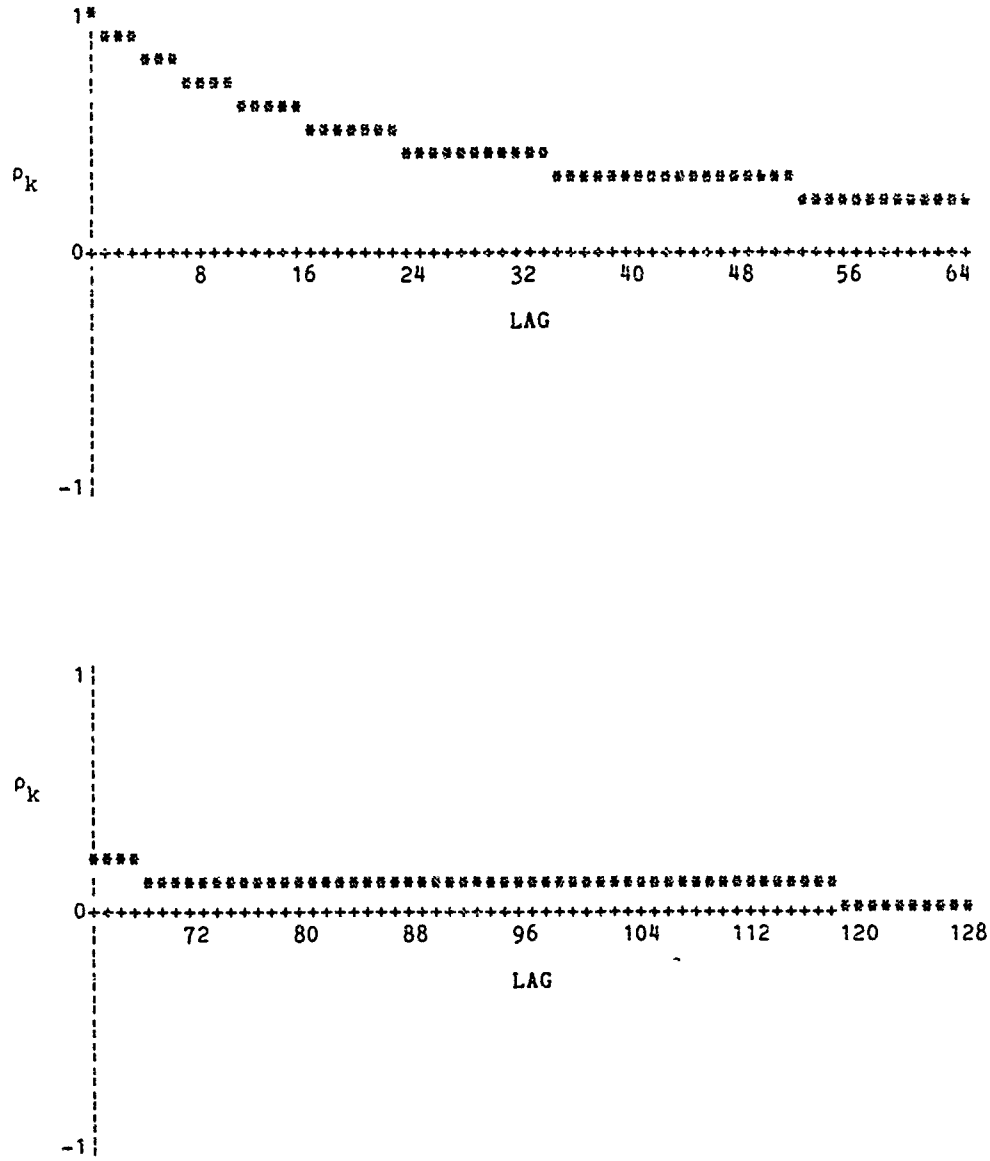


Figure 10-3 Correlogram -- 8000 Observations of M/M/1 Queueing Delays

in a deterministic manner using a test host and accumulated the time spent by every tenth packet on the modem queue in a particular IMP. Under light load, the sequence of packet delays produced the time series displayed in Figure 10-4. The correlogram associated with this time series, shown in Figure 10-5, clearly has a damped harmonic structure to it. This is exactly what one would expect the correlogram of a periodic deterministic process to look like.

We ran our batch means analysis routines over this data. The output of this analysis is shown in Figure 10-6. As can be seen in Figure 10-5, the serial correlation coefficients are significantly different from zero for lags extending out to at least 30. This means that observations spaced as much as 30 observations apart have a significant degree of correlation. Yet, the batch means method decided that a batch size of only 16 was sufficient to produce statistically independent batch means.

The clearly erroneous choice of batch size for the IMP data points to two weaknesses in the batch means method. The first weakness is not necessarily fatal. The test statistic used to determine serial independence of the batch means, which was given in Equation 10-8, may be rewritten as

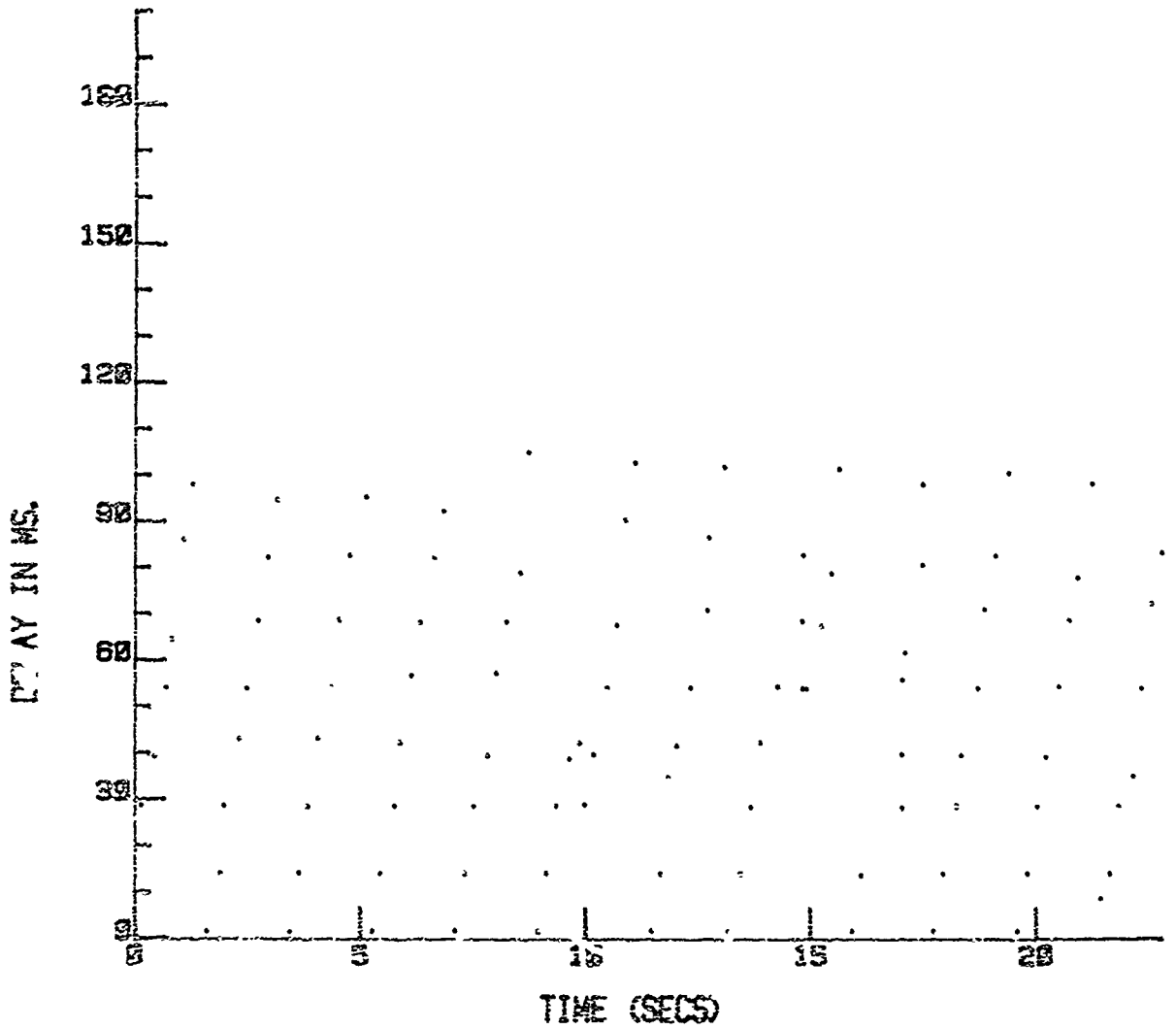


Figure 10-4 Time Spent on Modem Queue -- Test Host Data

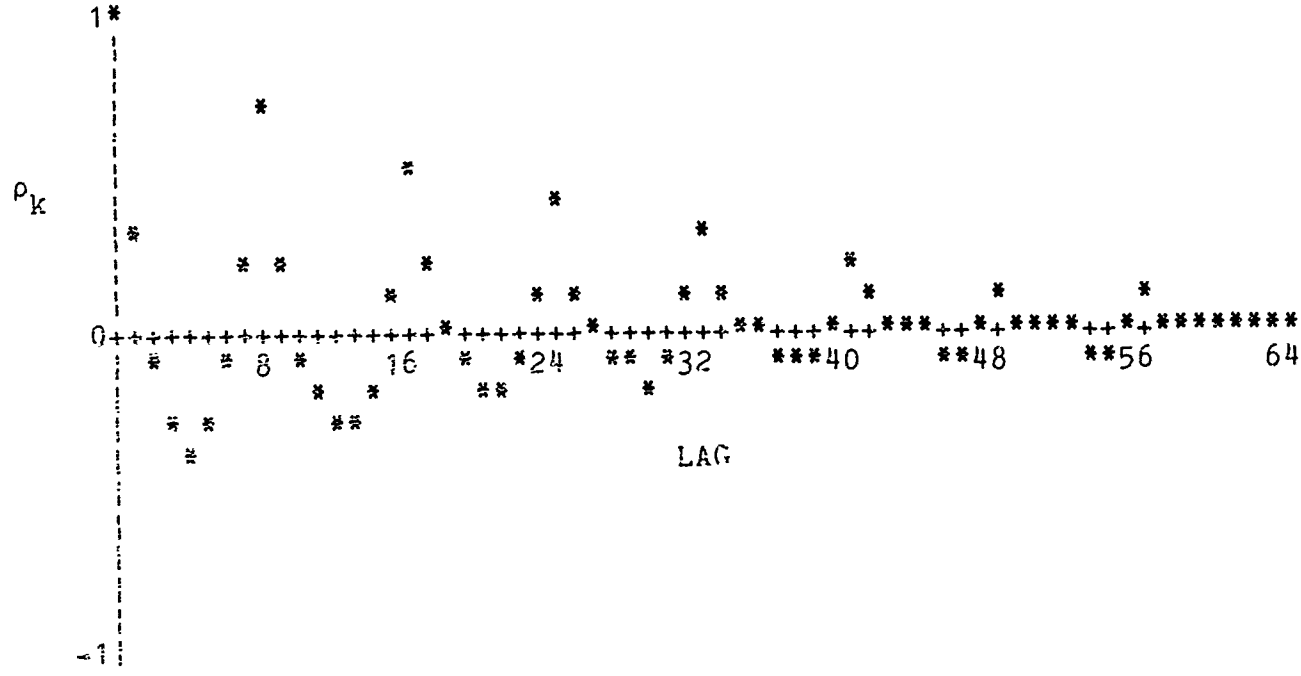


Figure 10-5 Correlogram -- Modem Queue Waiting Time for Test Host Data

0.95 interval estimation----batch method

sample mean= 49.45718

no. of batches	no. of obs. per batch	sample variance of sample mean	0.95 interval estimate lower	upper	c	critical value
4458	1	0.227964	48.520921	50.393434	0.324	0.029
2229	2	0.283732	48.412377	50.501978	0.057	0.042
1114	4	0.258024	48.460289	50.454066	0.713	0.059
557	8	0.07380	48.923440	49.990915	0.132	0.083
278	16	0.079905	48.900592	50.013763	0.015	0.117
139	32	0.066618	48.946712	49.967644	0.126	0.165
69	64	0.061782	48.961074	49.953281	0.051	0.233
34	128	0.059643	48.960195	49.954161	0.217	0.326
17	256	0.061207	48.932583	49.981772	0.212	0.447
8	512	0.054229	48.906378	50.007977	0.389	0.605

278 batches suffice for independence at 0.05 level

Figure 10-6 Batch Means Output -- Modem Queue Waiting Times

Eq. 10-16 $$C_k \approx \hat{\rho}_1 + \frac{(Y_1 - \bar{Y})^2 + (Y_k - \bar{Y})^2}{2(k-1)\hat{R}_0}$$

where, once again, there are k batch means. For large k, the test statistic therefore approximately equals the first order serial correlation coefficient. Thus, the test for independence roughly assumes that if adjacent batch means are uncorrelated, all batch means are independent. Unfortunately, for harmonic autocovariance functions it is possible to have ρ_1 approximately zero and ρ_k not equal to zero for k>1 [3]. If the correlogram has this sort of structure, then the test for independence which is based upon ρ_1 will invariably be wrong. This particular problem may be corrected by resorting to a different test for independence or a different approach to the selection of a batch size. We have therefore implemented 4 tests for serial independence as part of our statistics package and we are examining other algorithms for the selection of a batch size.

There is a second problem with the batch means method which is more profound than a problematic test for independence. The underlying assumption in the batch means approach is that, for a large enough batch size, the batch means can be considered to be independent and identically distributed. If there are long-term periodic processes present in the simulation, the batch means algorithm may either fail to detect such processes or fail to select a batch size. In the first case, erroneous results will

be produced; in the second case there will be no results. For large classes of experiments, the method would therefore appear to be inappropriate. When we have more data from simulated networks, we will be better able to assess the general applicability of the various batch means algorithms to queueing systems more complex than our M/M/1 model.

10.1.2 Autoregressive Models

In addition to the batch means algorithm, we have implemented a more powerful method for the analysis of simulation data [3]. This approach is borrowed from the techniques of linear prediction used in the analysis of discrete signals and involves the construction of a model for the output of the stochastic process being analyzed.

Consider the time series $\{X_i\}$ formed by observations on packet delays recorded in the simulation's steady state. We study the properties belonging to this time series by constructing a model for it in which the jth packet delay X_j, is represented as a linear combination of the previous p packet delays. That is, we assume that we can find a set of coefficients, b_i, such that we can approximate

Eq. 10-17

$$X_j = -\sum_{i=1}^{p} b_i X_{j-i}$$

In the literature on time series this sort of model is called on "all-pole" model. In the statistical literature, it is called an "autoregressive representation" of $\{X_i\}$ [6]. The value of p is called the order of the autoregressive representation.

As it turns out, there are extremely efficient methods for determining the values $\{b_i : i=1,p\}$ for any given choice of an autoregressive order p. We have implemented an approach due to Durbin [6]. If we represent the set of $\{b_i\}$ for the pth order representation as $b_{p,i}$, then we compute $\hat{b}_{p,j}$, an estimate for $b_{p,j}$ recursively according to the following formulas:

(a) $E_0 = \hat{R}_0$

(b) $$\kappa_i = \frac{-\hat{P} + \sum_{j=1}^{i-1} \hat{b}_{p-1,j}\hat{R}_{i-j}}{E_{i-1}}$$

(c) $\hat{b}_{p,p} = \kappa_i$

Eq. 10-18

(d) $\hat{b}_{p,j} = \hat{b}_{p-1,j} + \kappa_i \hat{b}_{p-1,i-j}$ $1 \leq j \leq i-1$

(e) $E_i = (1-\kappa_i^2)E_{i-1}$

In the above formulas $\hat{R}_i$ is the sample autocovariance function.

Using Durbin's algorithm, we can produce many representations for the same data, each of a different order p. We therefore require a means to decide upon a specific value for p. In order to do this a statistical test is used. We form

Eq. 10-19 $$\hat{\sigma}_i^2 = \sum_{s=0}^{i} \hat{b}_{i,s} \hat{R}_s \qquad b_{i,0} \equiv 1$$

Then

Eq. 10-20 $$T_{q-j} = n\left(1 - \frac{\hat{\sigma}_q^2}{\hat{\sigma}_j^2}\right)$$

converges to the chi-square distribution with q-j degrees of freedom for q sufficiently large [3]. Thus, we arbitrarily choose q to be some large integer (say 50). Starting with j=0, we compute, for successively higher values of j, the value of the test statistic T_{q-j} and compare it with the $1-\alpha$ percentile value of the chi-square distribution with q-j degrees of freedom. Here α is the size of the chi-square test (typically, .10). The first value of j for which the test statistic T_{q-j} is less than the computed critical value is chosen as the order of the representation.

Once one knows p and $\{\hat{b}_{p,i}\}$ one has completed the autoregressive model. It remains to compute an estimate for $\mathrm{var}(\bar{X}_n)$ and the number of degrees of freedom to be used in computing confidence intervals from the t statistic. Given:

Eq. 10-21 $$\hat{\sigma}_p^2 = \sum_{s=0}^{p} \hat{b}_{p,s} \hat{R}_s$$

and

Eq. 10-22 $$\hat{b} = \sum_{s=0}^{p} \hat{b}_{p,s}$$

then the number of degrees of freedom is given by [3]

Eq. 10-23 $$f = \frac{n\hat{b}}{2 \sum_{s=0}^{p} (p-2s)\hat{b}_{p,s}}$$

Thus, the $1-\alpha$ confidence interval is given by

Eq. 10-24 $$\bar{X}_n \pm \sqrt{\mathrm{var}(\bar{X}_n)}\, t_{\frac{\alpha}{2};f}$$

The autoregressive technique is extremely powerful since one is, in fact, contructing an analytic model for one's data.

Having developed such a model, quite a bit more information is accessible than simple interval estimates for the mean delay. In particular, one can efficiently estimate the serial correlation coefficients, estimate the transient period of the simulation and produce the spectrum of the stochastic process. The spectrum contains much information about whatever periodic phenomena are present in the simulation.

For a representation of order p, if one computes the serial correlation coefficients $\hat{\rho}_k$ for $k \leq p$ one can estimate $\hat{\rho}_k$ for $k > p$ recursively by [3]

Eq. 10-25
$$\hat{\rho}_k = - \sum_{s=1}^{p} \hat{b}_{p,s} \hat{\rho}_{j-s}$$

The use of this formula allows an enormous savings in CPU time over the computation using Equation 10-15.

In order to analyze the transient period, we define the conditional mean for the jth observation as

Eq. 10-26
$$\mu_j = E(X_j | X_{j-1}, \cdots, X_{j-p}, \cdots, X_1)$$

Of course, $\mu_j \rightarrow \mu$ as j increases. That is, the expected value of the sample mean based upon j observations converges to the steady state mean with j. The expression

Eq. 10-27

$$I_j = \frac{|\mu_j - \mu|}{|\mu|}$$

therefore measures the relative influence on X_j, the jth observation, of the initial conditions. The closer I_j is to zero, the closer the system is to the steady state. One can easily express the value of the conditional mean μ_j in terms of the autoregressive coefficients by [3]

Eq. 10-28

$$\mu_j = \mu - \sum_{s=1}^{\rho} b_{p,s}(\mu_{j-s} - \mu)$$

The computation of the transient period then follows directly. We assume that $X_1 = X_2 = \ldots X_p = 0$. That is we assume that the simulation is initially in the empty state. Then, $\mu_{p+1} = \mu b$. We substitute $\hat{b}_{p,s}$ for $b_{p,s}$ and $\bar{X}_n$ for μ in Equation 10-28 and compute μ_j recursively until the value of I_j is sufficiently small. The value of j for which this is so is the size (in number of observations) of the transient period.

Finally, cyclic phenomena in the simulation can be directly investigated using the spectrum of the stochastic sequence of observations. The spectrum is the fourier transform of the autocovariance function R_s:

Eq. 10-29 $$g(\lambda) = \frac{1}{2\pi} \sum_{s=-\infty}^{\infty} R_s e^{-i\lambda s} \qquad g(\lambda) = g(-\lambda)$$

and

Eq. 10-30 $$R_s = \int_{-\pi}^{\pi} g(\lambda) e^{i\lambda s} d\lambda$$

Since

Eq. 10-31 $$R_0 = \int_{-\pi}^{\pi} g(\lambda) d\lambda$$

one can think of the variance of the stochastic process as being caused by a continuum of random variables indexed by λ. The spectrum can be represented as [3]

Eq. 10-32 $$g(\lambda) = \frac{\sigma^2}{2\pi \sum_{r,s=0}^{p} b_{p,r} b_{p,s} \cos\lambda(r-s)} \quad |\lambda| \leq \pi$$

The autoregressive analysis routines were applied to the IMP data for which the batch means method performed so poorly. The results are shown in Figure 10-7. As can be seen, a complete analysis is carried out for all autoregressive orders up to and

including 50. For each iteration, the sample variance of the sample mean and interval estimates are computed. The column labelled T contains the value of the test statistic (T_{q-j}). This is compared with the 90th percentile value of the chi-square distribution which is contained in the column labeled "critical value." The order, p, of the autoregressive representation is chosen as the lowest order for which T is less than the critical value. For the analysis shown in Figure 10-7, the order equals 39. The output from the analysis routines displays the sample autoregressive coefficients $\hat{b}_{p,j}$ for the chosen autoregressive order.

Below the sample coefficients, additional information is displayed. First, the least values of j for which the absolute value of $\hat{\rho}_j$ is less than .05 and .01 are displayed. Next, the least values of j for which I_j (see Eq. 10-27) is less than .01 and .05 are given. These latter values are estimates for the transient period.

Figure 10-8 displays the correlogram. For j>39 the values for ρ_j are estimated using Equation 10-25. A comparison between these $\hat{\rho}_j$, which are based purely upon the autoregressive model for the stochastic process, with the $\hat{\rho}_j$ computed directly from the time series (and shown in Figure 10-5) shows that the representation is fairly good.

0.95 interval estimation--autoregressive approach

sample mean = 49.45718

order	sample variance of sample mean	0.95 interval estimate lower	upper	degrees of freedom	t	critical value
0	0.22791	48.52103	50.39333	4457	2554.311	63.169
1	0.44630	48.14613	50.76822	1138	2331.166	62.040
2	0.27916	48.42036	50.49400	1205	2211.942	60.909
3	0.17176	48.64393	50.27042	1271	2076.876	59.776
4	0.09595	48.84941	50.06494	1449	1869.295	58.643
5	0.05298	49.00558	49.90877	1590	1634.274	57.507
6	0.03370	49.09700	49.81736	1459	1487.243	56.371
7	0.03332	49.09885	49.81551	892	1487.147	55.232
8	0.10050	48.83212	50.08224	211	486.158	54.092
9	0.05720	48.98665	49.92771	339	162.268	52.950
10	0.05817	48.98238	49.93197	289	161.964	51.807
11	0.06298	48.96266	49.95170	237	155.165	50.662
12	0.06232	48.96504	49.94931	216	155.043	49.514
13	0.05724	48.98551	49.92885	214	147.264	48.365
14	0.05060	49.01375	49.90060	221	130.894	47.214
15	0.05479	48.99529	49.91907	185	124.051	46.061
16	0.06517	48.95248	49.96188	144	91.359	44.905
17	0.05457	48.99579	49.91857	162	56.830	43.747
18	0.05390	48.99841	49.91595	153	56.664	42.586
19	0.05448	48.99595	49.91871	141	56.538	41.423
[illegible]	0.05676	48.98564	49.92871	128	54.685	40.258
21	0.05528	48.99171	49.92264	124	53.913	39.089
22	0.05426	48.99586	49.91849	120	53.531	37.918
23	0.05039	49.01273	49.90162	122	47.487	36.743
24	0.05192	49.00564	49.90871	113	46.500	35.565
25	0.04821	49.02218	49.89218	115	40.442	34.383
26	0.04762	49.02466	49.88970	111	40.277	33.198
27	0.04576	49.03317	49.88118	110	38.513	32.008
28	0.04476	49.03769	49.87666	107	37.976	30.815
29	0.04387	49.04176	49.87259	104	37.530	29.617
30	0.04258	49.04783	49.86652	103	36.547	28.413

Figure 10-7 Output of Autoregressive Analysis -- Modem Queue Waiting Times

order	sample variance of sample mean	0.95 interval estimate lower	upper	degrees of freedom	t	critical value
31	0.04357	49.04273	49.87162	96	35.958	27.205
32	0.04675	49.02723	49.88713	86	30.473	25.991
33	0.04482	49.03626	49.87810	86	28.491	24.770
34	0.04393	49.04032	49.87404	85	28.052	23.543
35	0.04468	49.03646	49.87789	80	27.739	22.308
36	0.04736	49.02336	49.89100	73	23.970	21.065
37	0.04593	49.02994	49.88442	73	22.930	19.813
38	0.04706	49.02431	49.89004	69	22.281	18.551
39	0.04313	49.04318	49.87117	73	13.850	17.276
40	0.04121	49.05259	49.86177	74	11.550	15.988
41	0.04198	49.04846	49.86590	70	11.168	14.685
42	0.04102	49.05309	49.86126	70	10.575	13.363
43	0.03865	49.06514	49.84922	72	6.637	12.018
44	0.03950	49.06052	49.85384	68	6.120	10.646
45	0.03971	49.05917	49.85519	66	6.086	9.237
46	0.04168	49.04882	49.86553	61	3.491	7.781
47	0.04003	49.05709	49.85726	62	1.675	6.253
48	0.03923	49.06103	49.85333	61	1.227	4.609
49	0.04028	49.05534	49.85901	58	0.458	2 734
50	0.04110	49.05084	49.86351	55	0.000	0.000

autoregressive order = 39

sample autoregressive coefficients

1.00000	-0.27656	0.15267	0.15591
0.17342	0.15037	0.07032	0.10709
-0.45784	0.22803	0.01251	-0.03771
0.00969	0.01527	0.07496	-0.02557
-0.08234	0.06957	0.01437	0.00299
-0.00880	0.02258	-0.00059	0.04934
0.00149	0.03610	-0.00232	0.01432
0.02058	-0.00916	0.03153	-0.02309
-0.03962	0.01640	0.01469	0.00760
-0.02835	0.02526	-0.02412	0.04356

minimal j= 18 for abs(autocorrelation(j)) <= .05

minimal j= 42 for abs(autocorrelation(j)) <= .01

minimal j= 101 for abs((conditional mean(j)-mean)/mean) <= .01

minimal j= 80 for abs((conditional mean(j)-mean)/mean) <= .05

Figure 10-7 (cont.) Output of Autoregressive Analysis -- Modem Queue Waiting Times

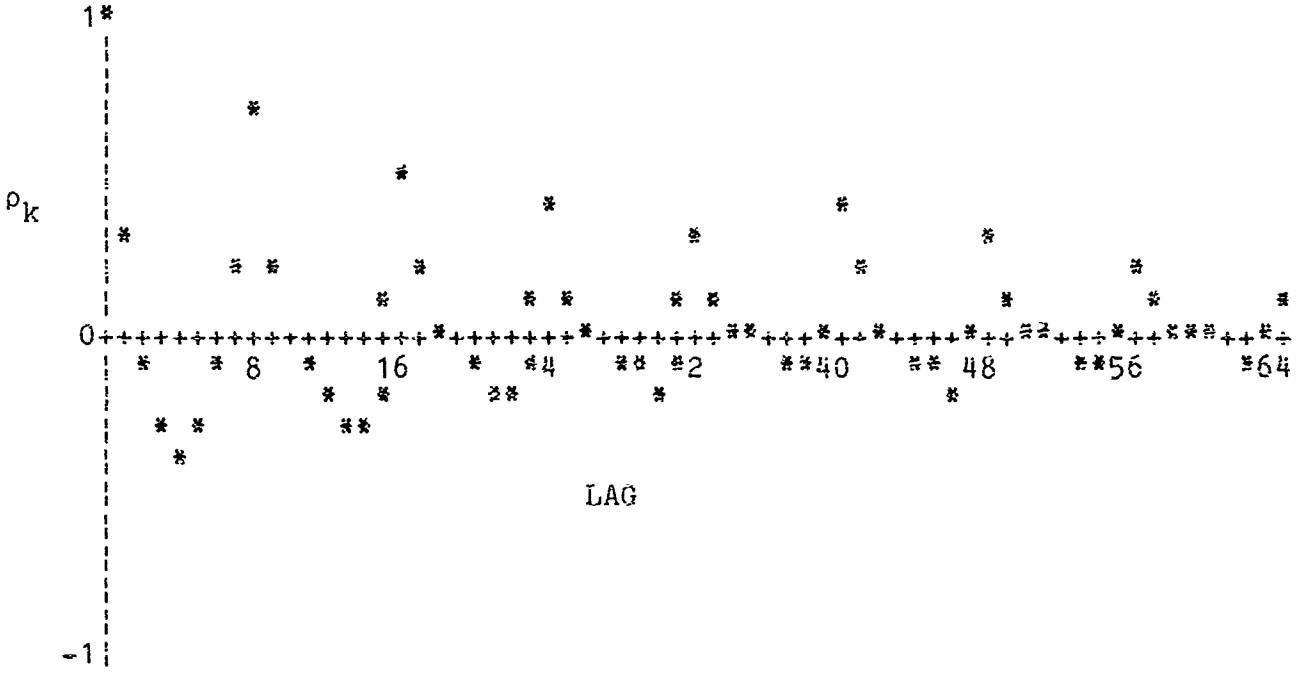


Figure 10-8 Correlogram for Modem Queueing Delays

Finally, the analysis routines also computed and graphed the spectrum of the series of IMP modem queue waiting times. This is shown in Figure 10-9. The x-axis is in units of $\pi/64$. The major peak in the spectrum occurs at $16\pi/64$. This corresponds to a period of $2\pi/(16\pi/64)$ or 8. Examination of the actual time series of delays, shown in Figure 10-4, shows that the period of the series of IMP delays is indeed 8.

As a final check on the autoregressive method, we applied it once again to 100 independent replications of an M/M/1 queueing system with a .75 activity level. A sample output for a single replication is shown in Figure 10-10. It will be observed that the autoregressive order was chosen as 10 and that the computed confidence interval for the 10th order scheme does indeed include the theoretical mean queueing delay of 2.25. It will also be observed that the estimated transient period is, in this case, 533, somewhat higher than the transient of 100 which was estimated from the diffusion model but, nevertheless, safely less than the 1000 observations which were discarded during the actual data collection.

A summary of the results for the first 10 replications of the M/M/1 simulator is contained in Figure 10-11. Comparison of the interval estimates for these replications as computed by the autoregressive method and the batch means method (Fig. 10-2) shows very good agreement.

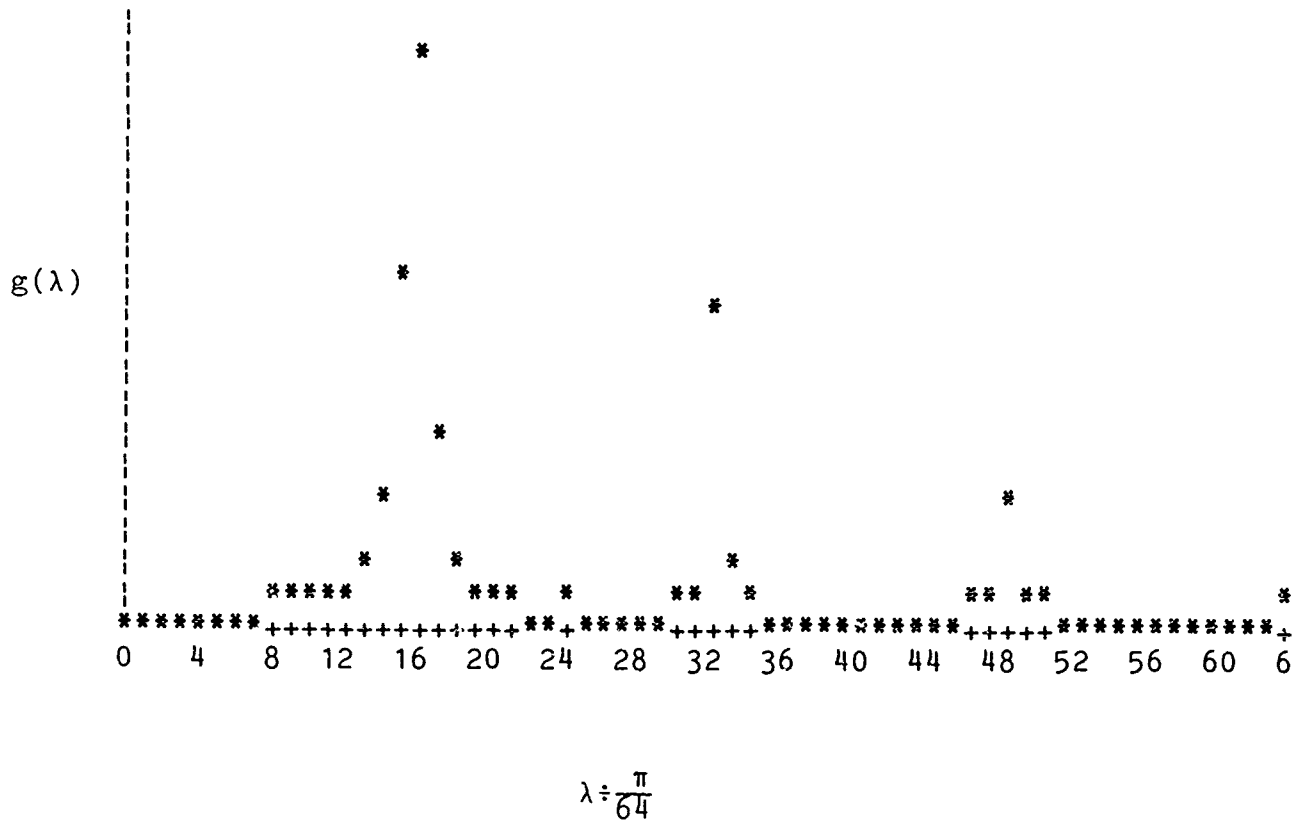


Figure 10-9 Spectrum for Sequence of Modem Queueing Delays

0.95 interval estimation--autoregressive approach

sample mean = 3.55527

order	sample variance of sample mean	0.95 interval estimate lower	upper	degrees of freedom	t	critical value
0	0.00444	3.42456	3.68598	7999	7782.090	63.169
1	0.63480	1.92284	5.18769	28	109.106	62.040
2	0.71407	1.81350	5.29704	25	81.757	60.909
3	0.76936	1.73949	5.37105	23	70.743	59.776
4	0.77918	1.72604	5.38450	22	70.424	58.643
5	0.82497	1.66610	5.44444	21	63.957	57.507
6	0.86437	1.61523	5.49530	20	59.639	56.371
7	0.88139	1.59309	5.51744	19	58.884	55.232
8	0.89061	1.58086	5.52968	19	58.669	54.092
9	0.92303	1.53965	5.57088	18	56.130	52.950
10	0.99343	1.45226	5.65828	17	45.397	51.807
11	1.01377	1.42683	5.68371	17	44.580	50.662
12	1.03662	1.39846	5.71207	16	43.592	49.514
13	1.03424	1.40064	5.70990	16	43.581	48.365
14	1.08960	1.33340	5.77713	15	38.172	47.214
15	1.15890	1.25053	5.86001	14	30.602	46.061
16	1.21264	1.18682	5.92372	14	26.507	44.905
17	1.21503	1.18326	5.92728	14	26.499	43.747
18	1.28465	1.10173	6.00880	13	20.308	42.586
19	1.28732	1.09784	6.01270	13	20.300	41.423
20	1.30747	1.07381	6.03673	12	19.819	40.258
21	1.33037	1.04666	6.06388	12	19.217	39.089
22	1.36666	1.00423	6.10631	12	17.771	37.918
23	1.41068	0.95317	6.15736	11	15.766	36.743
24	1.43365	0.92618	6.18436	11	15.245	35.565
25	1.47696	0.87621	6.23433	11	13.478	34.383
26	1.47355	0.87916	6.23137	11	13.467	33.198
27	1.51203	0.83474	6.27580	11	12.140	32.008
28	1.49438	0.85378	6.25676	11	11.865	30.815
29	1.53453	0.80746	6.30307	10	10.462	29.617
30	1.53601	0.80486	6.30567	10	10.460	28.413

Figure 10-10 Output of Autoregressive Analysis
M/M/1 Queueing Simulation

order	sample variance of sample mean	0.95 interval estimate lower	upper	degrees of freedom	t	critical value
31	1.59434	0.73815	6.37238	10	7.685	27.205
32	1.61124	0.71818	6.39236	10	7.463	25.991
33	1.59305	0.73770	6.37283	10	7.206	24.770
34	1.59391	0.73578	6.37475	10	7.205	23.543
35	1.58129	0.74907	6.36146	10	7.079	22.308
36	1.57045	0.76040	6.35013	10	6.984	21.065
37	1.59904	0.72707	6.38346	10	6.333	19.813
38	1.61185	0.71163	6.39891	10	6.206	18.551
39	1.57716	0.74997	6.36057	10	5.261	17.276
40	1.57765	0.74846	6.36207	10	5.261	15.988
41	1.58830	0.73540	6.37513	10	5.170	14.685
42	1.60670	0.71354	6.39700	10	4.905	13.363
43	1.63474	0.68072	6.42981	9	4.307	12.018
44	1.68487	0.62289	6.48765	9	2.483	10.646
45	1.63973	0.67312	6.43742	9	1.108	9.237
46	1.63785	0.67427	6.43627	9	1.005	7.781
47	1.65788	0.65050	6.46004	9	0.710	6.253
48	1.65611	0.65152	6.45902	9	0.707	4.609
49	1.64060	0.66820	6.44234	9	0.530	2.734
50	1.61410	0.69749	6.41305	9	0.000	0.000

autoregressive order = 10

sample autoregressive coefficients

1.00000	-0.92384	-0.02378	-0.02974
0.02142	-0.00669	-0.01441	-0.00342
0.01218	0.01608	-0.03673	

minimal j= 339 for abs(autocorrelation(j)) <= .05

minimal j= 1154 for abs(autocorrelation(j)) <= .01

minimal j= 533 for abs((conditional mean(j)-mean)/mean) <= .01

minimal j= 350 for abs((conditional mean(j)-mean)/mean) <= .05

Fig 10-10 (cont.) Output of Autoregressive Analysis
M/M/1 Queueing Simulation

replication	order	$\bar{X}_n$	var($\bar{X}_n$)	.95 interval estimate
1	10	3.56	.993	[1.45 , 5.66]
2	2	2.20	.038	[1.81 , 2.59]
3	11	2.33	.029	[1.99 , 2.67]
4	2	2.08	.025	[1.78 , 2.40]
5	36	2.34	.086	[1.74 , 2.94]
6	1	2.16	.022	[1.87 , 2.46]
7	4	2.99	.016	[2.18 , 3.80]
8	2	2.43	.051	[1.99 , 2.88]
9	3	2.09	.021	[1.82 , 2.37]
10	1	2.25	.037	[1.87 , 2.63]

Figure 10-11 Summary of Autoregressive Analysis--
Ten Replications of M/M/1 Simulation

We once again determine the coverage rate yielded by the autoregressive analysis on all 100 replications. On 2 of the 100 replications, no autoregressive order less than 50 satisfied the chi-square test. Of the 98 replications for which we were able to make interval estimates, the theoretical delay of 2.25 was not within the computed interval on 5 replications. This represents a coverage rate of 93/98 or .949, which is remarkably close to the expected value of .95.

The autoregressive approach therefore appears to give very good results for an M/M/1 simulation. There is little doubt, however, that autoregressive models are not suitable for all types of data. We intend to investigate the suitability of this approach for data from network simulations. We also intend to determine whether better results will be provided by a model which is more complex than that described by Equation 10-17.

10.1.3 Transient Analysis

As was discussed in the previous chapter, there is a period in any simulation run in which the probability distributions which describe the system are biased toward the initial conditions. For example, if an M/M/1 simulation experiment is begun and if the simulated server is initially idle and the queue empty, then, for some period of simulated time, there will be a greater probability for the simulated system to be in the idle state than was predicted by the steady state distributions.

This point is subtle in that the empty and idle state is simply a system state to which an M/M/1 simulator will return with finite probability during its lifetime. In fact, it is the most probable state. This being the case, the source of the error introduced by starting a simulation in a state to which the simulation will invariably return is somewhat obscure.

The problem lies in the fact that the steady state probability associated with the empty and idle state for an M/M/1 simulator is given by

Eq. 10-33 $$P_0 = (1-\rho)$$

where ρ is the server utilization. As ρ increases, the probability of the empty and idle state occurring in the steady state diminishes. For ρ sufficiently large, we are therefore biasing the simulation results by starting the model in an improbable (albeit, the most probable) state. This will be reflected in a larger than expected propensity to be in the empty state for some time after the simulation is started. This is another way of stating the fact that we are observing a probability distribution which is <u>conditional</u> on the initial state of the system. If the system is stationary, then, as time proceeds, the state probabilities will approach the steady state

distribution. For a simple single server queueing system, the error introduced by starting the simulation in the empty state is probably not severe for moderate utilizations. However, for more complex systems, such as large queueing networks, the empty state may correspond to an extremely improbable state.

The overall goal of the analysis of the transient is to reduce bias. This translates into two practical tasks. The first task is to eliminate those observations obtained when the system probability distributions differ from the steady state distribution. The second task is to minimize the number of discarded observations by choosing some reasonable starting state for the simulation. Unfortunately, neither of these two goals of transient analysis admits of an exact solution. It is virtually impossible to decide what the steady state distribution is until the simulation is run and biased observations eliminated. It is difficult to eliminate bias unless the steady state distribution is known. In what follows, we discuss the approximate techniques that we will utilize in our simulation work.

Thus far, we have discussed two mechanisms for the analysis of the transient period. The first method predicts the transient via a diffusion model for the system under study. The second method is based upon the construction of an autoregressive representation of the time series of observations on the system (see Eq. 10-27). Eq. 10-12, due to Kobayashi, provides a formula

for the transient time of a G/G/1 queueing system. For an M/M/1 simulator, with an average interarrival time of 1 second and an average service time of .75 seconds, Equation 10-12 yields a transient period of 105 seconds. On 95 replications of the M/M/1 simulator, the transient period computed by the autoregressive analysis routines and averaged over all replications was observed to be 112. The agreement between the analytic and autoregressive methods is, in this case, very good.

Unfortunately, neither of these methods is available for general use. There are no diffusion models for the complex networks, algorithms and protocols which we will investigate in the course of this study. Furthermore, the autoregressive approach to transient analysis is an ex post facto method. That is, the autoregressive method analyzes steady state data in order to estimate the transient period of the simulation. Access to steady state data requires knowledge of the system transient. Hence, we can use the autoregressive method to retroactively verify a particular choice for the transient period but not to estimate the transient period in an a priori manner. One is therefore forced to rely on more primitive means by which to perform transient analysis.

The basic concept behind the method implemented is to attempt to observe the passage of the system into the "steady state." The following describes the outline of the analysis.

1. Choose an interval size of m observations.

2. Start the simulator.

3. When the (k*m)th customer, k = 1,..., leaves the system, compute the kth cumulative mean as

Eq. 10-34
$$M_k = \frac{1}{km} \sum_{i=1}^{km} X_i$$

Where X_i is the parameter of interest, e.g., system response time seen by the ith customer.

4. Plot the M_k as a function of m*k.

The expected form of the output of the above process is as follows. If the simulation is begun in the empty and idle state the sequence of cumulative means should begin at some number which is small relative to the steady state mean, then increase and asymptotically approach the steady state mean. The projection onto the x-axis of the point where the curve is reasonably flat represents the size (in observations) of the transient period.

There are any number of objections that one might raise to this procedure. First, since we are using a cumulative mean, we are always including the transient period in our computations.

This will result in an estimate for the transient which is probably too large. For example, assume (unrealistically) that the sequence of packet delays is $\{x_i\} = \{0,1,2,3,4,5,5,5,...\}$. Then, using an interval size, m, of 5, the sequence of cumulative means is $\{M_k\} = \{2,3.5,4,4.3,4.5,4.6,4.6,...\}$. In this case, one computes 6 cumulative means before one observes the "steady state." Hence, one might choose a transient of 6X5 = 30 observations despite the fact that the actual transient is only 6 observations long.

There are two other problems with the cumulative means approach to the transient. First, it is possible that one can be confused by real stochastic fluctuations in the data. Second, since the number of observations used in calculating the mean is monotonically increasing, it would seem that one is forcing the curve to flatten, which is the result one is trying to observe. Thus, it is hard to tell whether the slope of the plotted means has decreased because the "steady state" has been reached, or because the denominator in the computation of the cumulative average has grown so large that a transient period is no longer observable. This would seem to result in an estimate of the transient which is too small.

To the extent that the cumulative means method yields a conservative (too large) estimate for the transient, the procedure is acceptable. That is, the real risk one takes in an

analysis of the transient is that the results that are produced are biased toward the initial conditions. Biased results will not be obtained if one overestimates the transient period.

We tested the cumulative average method on three independent replications of our M/M/1 simulator, each with a server utilization of .75. In computing the cumulative average, we used an interval size of 100. The results are displayed in Figures 10-12 through 10-14. The y-axis represents the cumulative average delay; the x-axis is in units of 100 observations. It will be observed that the three curves have the same overall shape. The average starts at a small value, rapidly rises and overshoots the steady state mean and then falls back, eventually becoming flat. The consistent increase of the cumulative mean to a point greater than the system mean is a result which has been observed by others [7].

If we identify the end of the transient period as that point on the curve where the slope has become flat, we estimate the transient period as roughly 500 observations on each replication. This is clearly longer than the 100 observations which were estimated by using the analytic and autoregressive approaches. The cumulative means method therefore seems to produce very conservative results. While this result is somewhat disturbing, the fact that we lack a suitable alternative procedure, and the fact that an excessively large estimate of the transient is

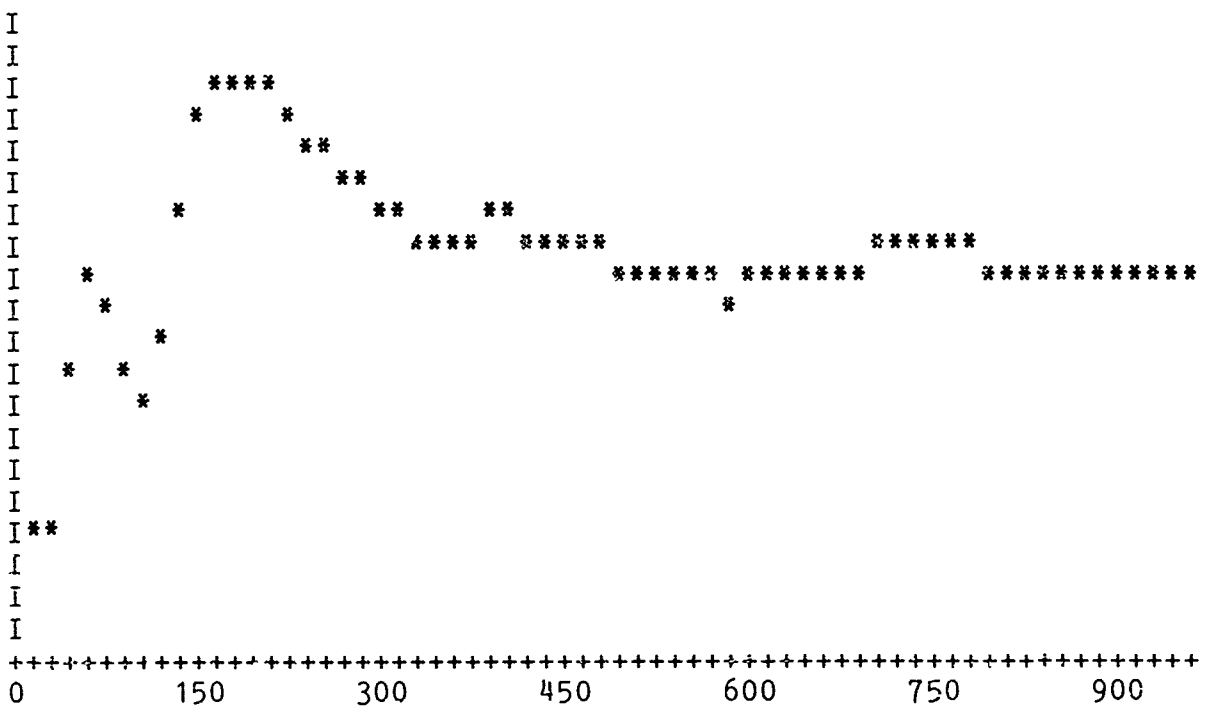


Figure 10-12 Cumulative Means -- M/M/1 Simulation

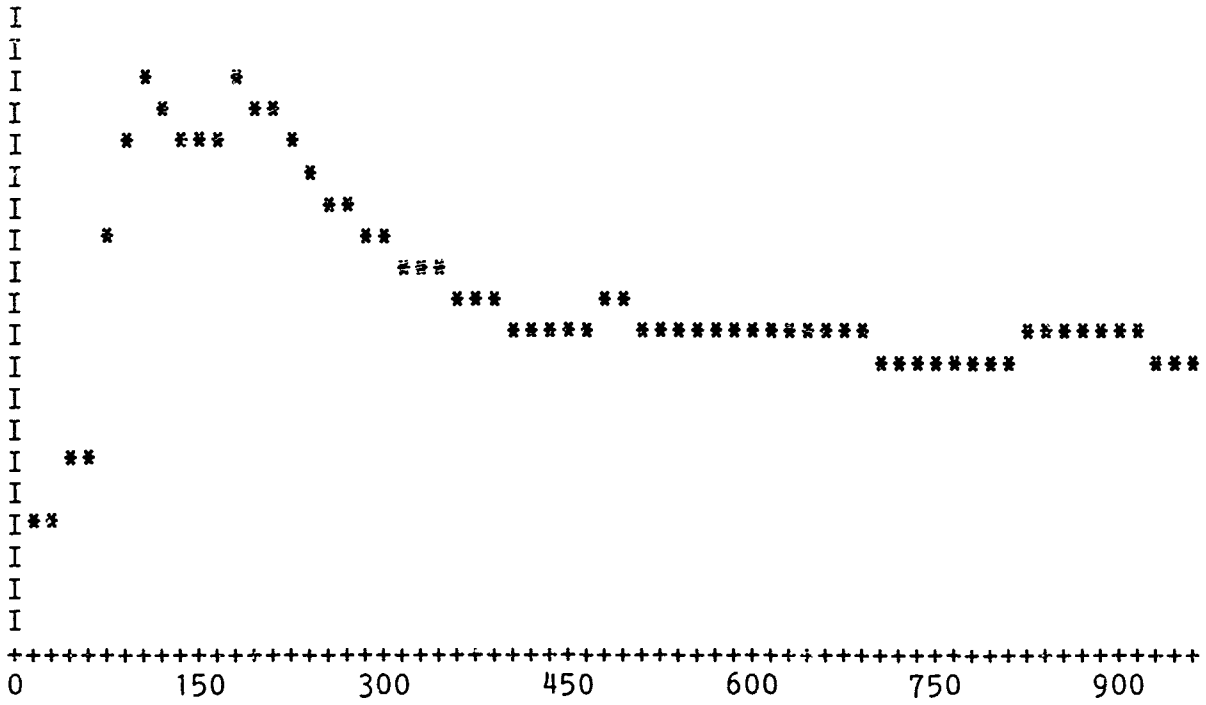


Figure 10-13 Cumulative Means -- M/M/1 Simulation

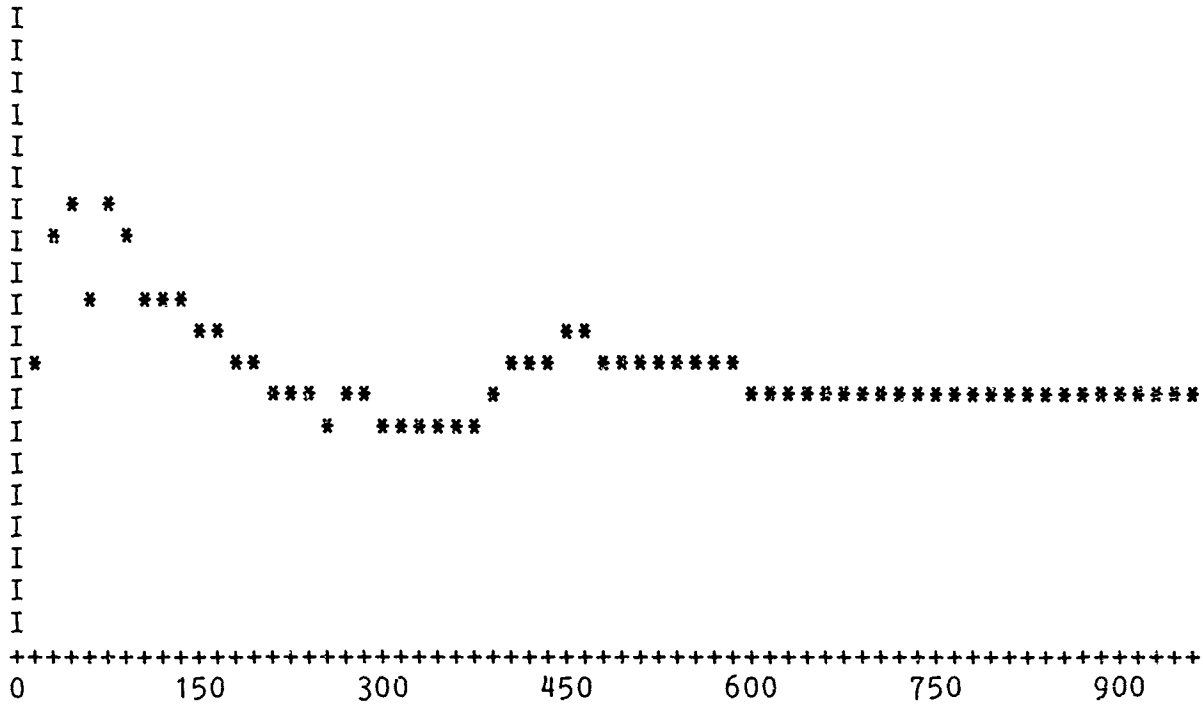


Figure 10-14 Cumulative Means -- M/M/1 Simulation

better than an excessively small estimate, make the cumulative average an "acceptable" procedure.

Fishman [3], in discussing transient analysis, makes the somewhat surprising statement that various proposed methods (such as cumulative means) which rely upon data to decide upon a transient response have weak theoretical foundations. He suggests an approach which seems to rely on gross assumptions about the behavior of the conditional mean as a function of time. We have not yet implemented this approach since we want to further investigate the conditions under which Fishman's assumptions hold.

The second goal of the analysis of the transient is to reduce the size of the transient period by choosing a starting state which is more representative of the system state. We have deferred a detailed study of this question until we have more simulated network data to examine.

The overall conclusion that one can draw about the problem of the analysis of the transient is that totally satisfactory methods do not seem to exist. We intend to actively pursue this issue.

10.2 Specification of the Analysis Package

The simulation analysis package has been implemented as a set of SIMULA procedures and classes which are called and created by a command module. The package will ultimately be extended to incorporate the facility to design and analyze multifactor experiments although these routines have yet to be specified and written. In this section, we describe the structure of the analysis package (software and algorithms) as currently implemented.

Command Language

All analysis routines are callable from a SIMULA program called CMD. The commands in CMD have a tree structure as depicted in Figure 10-15. As can be seen, there are three main branches to the CMD tree: ANALYZE, TEST and COMPUTE.

The ANALYZE branch (Fig. 10-16) currently has one main option, ANALYZE SIMULATION, which is used for the analysis of simulation output data via either BATCH MEANS or AUTOREGRESSION. Once one is in the AUTOREGRESSION branch of the ANALYZE SIMULATION subtree, one has the additional options of computing and displaying the correlogram and spectrum of the data as estimated from the autoregressive coefficients. A sample ANALYZE session is shown in Figure 10-17.

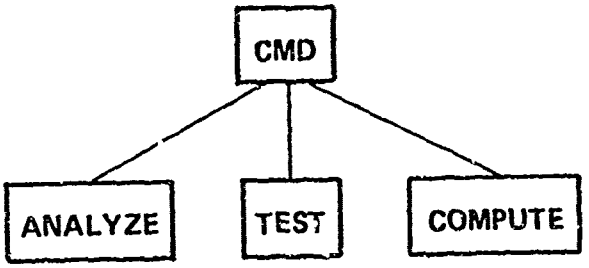


Figure 10-15 CMD Analysis Modes

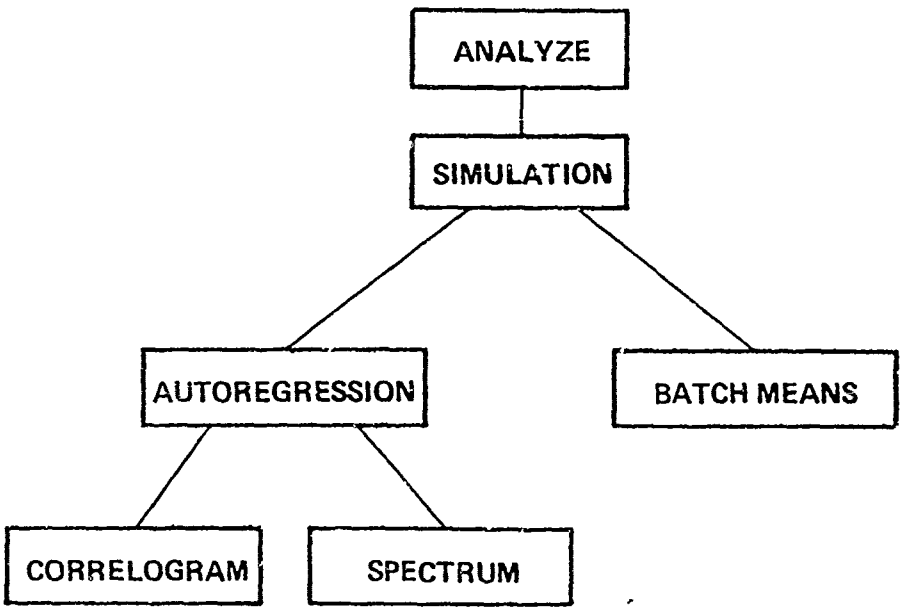


Figure 10-16 ANALYZE Subtree

```
@run cmd
> mode is analyze
>> analyze simulation
>> simulation analysis via batch means
>> input file: x1.trc
>> output file: x1.bat
>> size of independence test: .10
>> confidence interval: .95
>> one sided or two sided test for independence (1 or 2): 1
>> beginning batch means analysis
>> done
>> simulation analysis via
>> analyze
> mode is

4 garbage collection(s) in 36 ms

End of SIMULA program execution.
CPU time: 12:0.45       Elapsed time: 57:39.75
```

Figure 10-17 Sample ANALYZE Session

The TEST branch (Fig. 10-18) currently has two main options: TEST INDEPENDENCE and TEST DISTRIBUTION. The former tests the hypothesis that the data contained in a user-specified file are independent and identically distributed; the latter tests the hypothesis that the data are distributed according to a user-specified distribution function. Currently, two tests for independence, that described in Equations 10-8 through 10-11 and the periodogram are included as options. The distributions that may be tested for are the CHISQUARE, EXPONENTIAL, NORMAL and UNIFORM distributions. The parameters belonging to the chi-square (degrees of freedom) and uniform (lower, upper bound) distributions must be specified by the user. The parameters belonging to the normal (mean, variance) and exponential (mean) distributions may be either supplied by the user or estimated from the data. A sample session is shown in Figure 10-19.

The COMPUTE branch of the command tree (Fig. 10-20) has two options: FILE STATISTICS and CORRELOGRAM. The FILE STATISTICS option computes the sample mean, variance, standard deviation and confidence intervals. The CORRELOGRAM option computes the correlogram out to any lag and displays the results graphically on the terminal or outputs the graph to a file. A sample session is shown in Figure 10-21.

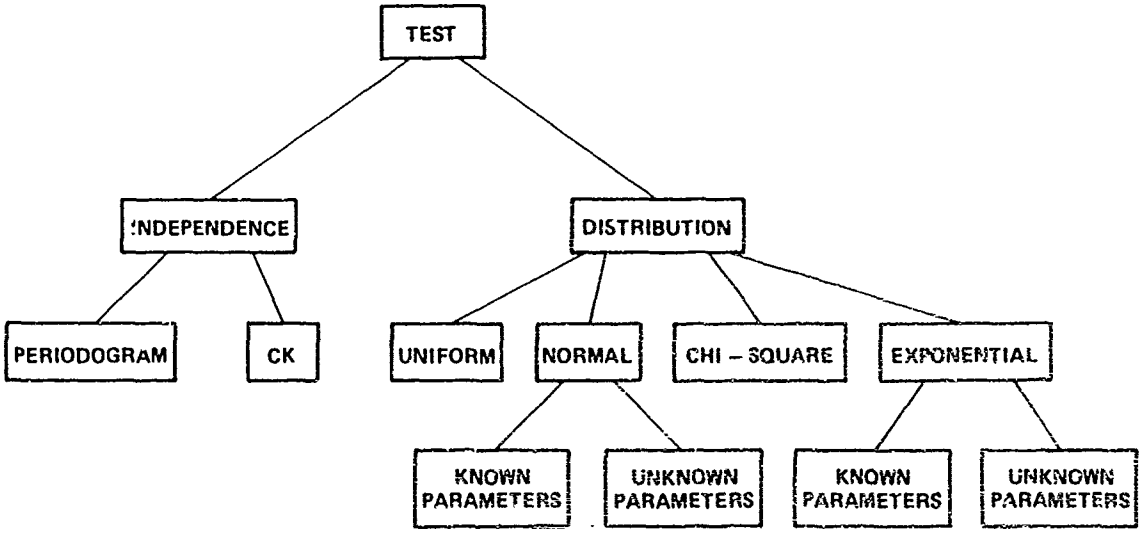


Figure 10-18 TEST Subtree

```
 run cmd
> mode is ?
> options are: analyze, test, compute
> mode is test
>> test ?
>> options are distribution or independence
>> test distribution
>> distribution type: ?
>> options are uniform, exponential, normal, chi-square
>> distribution type: chisquare
>> input file: rngchi.out
>> degrees of freedom: 999
>> size of test: .01
>> kolmogorov-smirnov critical value =    0.23052
>> computed test statistic =    0.24345
>> hypothesis of chi-square distribution not accepted at 0.010 level
>> distribution type:
>> test
> mode is

1 garbage collection(s) in 6 ms

End of SIMULA program execution.
CPU time: 1.60  Elapsed time: 1:23.21
```

Figure 10-19 Sample TEST Session

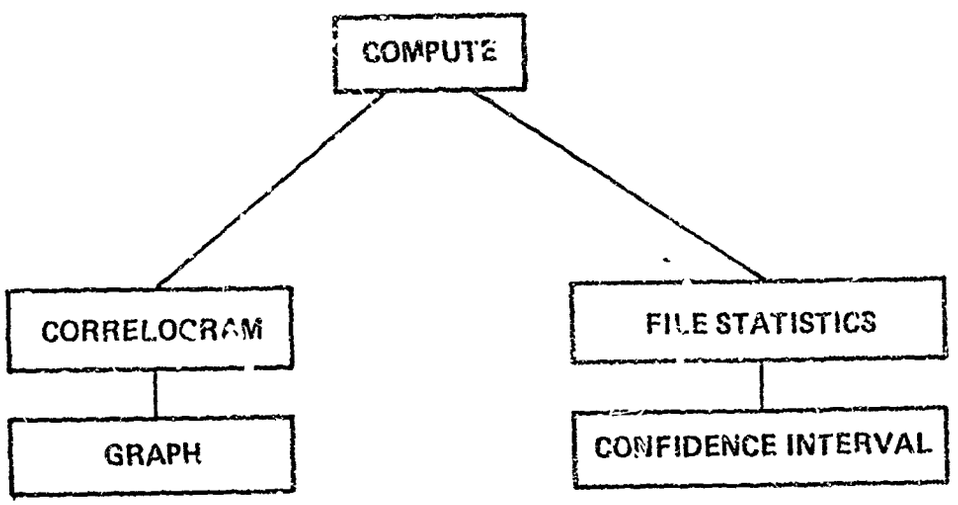


Figure 10-20 COMPUTE Subtree

```
@run cmd
> mode is ?
> options are: analyze, test, compute
> mode is compute
>> compute ?
>> options are correlogram, file statistics
>> compute file statistics
>> input file: rngchi.out
>> file size:          50
>> mean:       1016.3793943787
>> variance:       1482.7828710812
>> standard deviation:          38.5069197813
>> confidence interval: .99
>> 0.990 confidence interval:    14.59668
>> confidence interval: 0
>> compute
> mode is

1 garbage collection(s) in 7 ms

End of SIMULA program execution.
CPU time: 0.67  Elapsed time: 46.26
@
```

Figure 10-21 Sample COMPUTE Session

10.3 Specification of Analysis Routines

In this section we describe the major SIMULA modules which are called from the command level of the analysis package.

10.3.1 Data Handling Utilities

10.3.1.1 CLASS IOFILE(FILENAME); TEXT FILENAME;

Each IOFILE object references a disk file named FILENAME. The object can then be accessed and controlled via tape control-like procedures. IOFILE objects can be repeatedly read or overwritten in a sequential manner. They may be open for either reading or writing although not for both simultanecusly.

procedures

CLOSEFILE;	closes the file
RESET;	allows the file to be read again in sequential fashion.
REWRITE;	allows the file to be written over.
real procedure READ;	reads the next (real) record in the file
WRITE (X,D,W);	writes the real constant X to the file in format, D,W. (places to right of decimal, total field width)

attributes

boolean EOF;	is set to true if the last record in the file has been read

10.3.1.2 IOFILE CLASS FILE;

A FILE object is created and can be treated in the same manner as an IOFILE object with the additional feature that when a FILE object is created, various statistical characteristics of the file are computed and become permanent attributes of the FILE object.

attributes

long real MEAN;	
long real SAMPLEVARIANCE;	
long real STDDEV;	standard deviation
integer COUNT;	size of file

10.3.2 Statistics Utilities

10.3.2.1 CLASS RANDOM;

RANDOM is a dummy class which allows convenient access to the properties of various standard distributions. These distributions are used, in turn, by the various statistical procedures.

10.3.2.2 RANDOM CLASS CHISQUARE (PARAMETER);
LONG REAL ARRAY PARAMETER;

A CHISQUARE object contains useful information about the chi-square distribution with PARAMETER(1) degrees of freedom.

procedures

long real procedure CDF(Z); real Z;	cumulative distribution function

formulas

CDF	[3, eqs. 26.4.4, 26.4.5, 26.4.14]

10.3.2.3 RANDOM CLASS EXPONENTIAL (PARAMETER);
LONG REAL ARRAY PARAMETER;

An EXPONENTIAL object contains properties belonging to an exponential distribution with mean (PARAMETER(1))-1

procedures

long real procedure CDF(Z); real Z;	cumulative distribution function

formulas

CDF(Z) = 1-exp(-Z*PARAMETER(1))

10.3.2.4 RANDOM CLASS NORMAL (PARAMETER);
LONG REAL ARRAY PARAMETER;

A NORMAL object contains properties of the normal distribution with mean PARAMETER(1) and standard deviation PARAMETER(2).

procedures

long real procedure CDF(Z);real Z;	cumulative distribution function

formulas

CDF	[1, Eq. 26.2.19]

10.3.2.5 RANDOM CLASS UNIFORM (PARAMETER);
LONG REAL ARRAY PARAMETER;

A UNIFORM object contains properties of the uniform distribution on the interval [PARAMETER(1), PARAMETER(2)].

procedures

long real procedure CDF(Z); real Z;	cumulative distribution function

formulas

CDF(Z)=0	Z < PARAMETER(1)
1	Z > PARAMETER(2)
$\frac{Z-PARAMETER(1)}{PARAMETER(2)-PARAMETER(1)}$	otherwise

10.3.2.6 LONG REAL PROCEDURE CHISQP (DF,P);
INTEGER DF; REAL P;

A call to CHISQP(DF,P) computes the Pth quantile of the chi-square distribution with DF degrees of freedom.

formulas

[3, p. 498]

10.3.2.7 LONG REAL PROCEDURE CONFIDENCE (PROB, DEV,N);
REAL PROB, DEV; INTEGER N;

A call to CONFIDENCE returns the PROB confidence interval for data having standard deviation DEV and degrees of freedom N-1.

formulas

see Eq. 9-17

10.3.2.8 LONG REAL PROCEDURE STDZ(P);
REAL P;

A call to STDZ(P) returns the Pth quantile of the standard normal distribution.

formulas

[3, p.498]

10.3.2.9 LONG REAL PROCEDURE STUDTP (DF,P);
INTEGER DF; REAL P;

A call to STUDTP (DF,P) returns the Pth quantile of the Student-t distribution with DF degrees of freedom.

formulas

DF<6	[2, Appendix A, Table 5]
else	[3, p. 499]

10.3.3 Major statistical Routines

The following are the primary modules which implement the various hypothesis testing methods necessary for the analysis of simulation data.

10.3.3.1 CLASS IIDTST (DATA,BETA,TYPE);
REF(FILE)DATA; REAL BETA; INTEGER TYPE;

An IIDTST object is created to test, at the BETA level of significance, the hypothesis that the data contained in FILE object DATA are independent and identically distributed. For a one-sided test, TYPE should be set to 1; otherwise TYPE should be set to 2.

attributes

real CK;	test statistic
real CRITVAL;	critical value
boolean RESULT;	true if hypothesis is accepted, else false

formulas

[3, p. 239]

10.3.3.2 CLASS KSGENL (INPUT, ALPHA, TYPE, PARAMETER);
REF(FILE) INPUT; REAL ALPHA;
TEXT TYPE; LONG REAL ARRAY PARAMETER;

A KSGENL object is created to test the hypothesis that the data contained in FILE object INPUT have distribution TYPE with parameters contained in the PARAMETER array. TYPE may be "CHISQUARE," "EXPONENTIAL," "EXPKNOWN," "NORMAL" or "UNIFORM." The contents of array PARAMETER are as described in 10.3.2.2 - 10.3.2.5.

The test used is a version of the Kolmogorov-Smirnov test. It is assumed that the characteristics of the distribution tested for are completely known. For example, we do not use this test to determine whether or not the data are normally distributed, but instead, whether or not they are normally distributed with a specified mean and variance. The exception is the test for exponentiality. If a test for the exponential distribution is desired and the characteristics of the data are known, TYPE should be set to "EXPKNOWN." If the characteristics are unknown, then TYPE should be set to "EXPONENTIAL." In the latter case, the mean will be estimated from the data itself.

attributes

boolean RESULT;	test result
real MAXD;	test statistic
real CRITVAL;	critical value

formulas

[2, Algorithm 8.4.2]

10.3.3.3 CLASS KSNMLF(INPUT, ALPHA);
REF(FILE) INPUT; REAL ALPHA;

A KSNMLF object is created to test, at the ALPHA significance level, the hypothesis that the data in FILE object INPUT are normally distributed. The test used is the Kolmogorov-Smirnov test and the mean and variance of the distribution tested for are assumed unknown and are estimated from the data in INPUT. This test is used by CMD for sample sizes which exceed 50. For smaller samples, the Wilke-Shapiro test for normality is used [see 10.3.3.6].

attributes

boolean RESULT;	test result
real MAXD;	test statistic
real CRITVAL;	critical value

formulas

[2, algorithm 8.4.3]

10.3.3.4 CLASS PERIODOGRAM (DATA, ALPHA);
REF(FILE) DATA; REAL ALPHA;

PERIODOGRAM is used to examine, at the ALPHA significance level, the hypothesis that the data contained in FILE object DATA are independent and identically distributed. The class code computes the periodogram {I
the periodogram in a temporary FILE object, and then uses the Kolmogorov-Smirnov test to determine if the {I
exponentially distributed.

attributes

boolean RESULT;	test results
ref(FILE)IM;	temporary file object containing the periodogram
ref(KSGENL)TEST;	Kolmogorov-Smirnov test object for exponential distribution

formulas

[5, eqs. 5.68, 5.69, 5.70]

10.3.3.5 CLASS SERIAL (DATA, MAXLAG);
REF(FILE) DATA; INTEGER MAXLAG;

A SERIAL object contains, for the data in FILE object DATA, the autocovariance and autocorrelation sequence out to lag MAXLAG. A call to procedure LAGGRAPH places the results of the computation of the correlogram for lags FIRSTLAG through MAXLAG

in array GRAPH. When the values in GRAPH are written to a peripheral device, the result is a graph of the correlogram. The x-axis is the lag, the y-axis is the estimated serial correlation coefficient. Only 65 lags may be plotted at one time. The values of the serial correlation coefficient for lags for which the absolute value of RHO exceeds MAXRHO are not plotted.

attributes

real array R(0:MAXLAG);	estimated autocovariance sequence
real array RHO(0:MAXLAG);	correlogram
character array GRAPH; (0:64, 0:20)	graph of correlogram

procedures

LAGGRAPH (FIRSTLAG, LASTLAG, MAXRHO);

formulas

(see Eq. 10-15)

10.3.3.6 CLASS SWTEST (DATA, BETA);
REF (FILE) DATA; REAL BETA;

Performs, at the BETA significance level, the Shapiro-Wilke test for normality on the data contained in FILE object DATA. The number of observations cannot exceed 50.

attributes

boolean RESULT;	indicates the outcome of the test.
long real CRITVAL;	critical value of the test
long real WK;	test statistic
real array COEFFICIENTS;	Shapiro-Wilke coefficients for computation of the test statistic [3, Table B.1]
real array DATAARRAY;	critical values [3, Table B.2]

formulas

[3, Eq. 2.80]

10.3.4 Primary Analysis Routines

The following are the major simulation analysis routines.

10.3.4.1 CLASS AUTOREGRESSION (DATA, ALPHA, INTERVALSIZE, OUTPUT);
REF (FILE) DATA; REAL ALPHA, INTERVALSIZE;
REF (OUTFILE) OUTPUT;

The creation of an AUTOREGRESSION object produces an autoregressive representation of the data contained in FILE object DATA. (See Sec. 10.2.) The chi-square test used to determine the order of the representation is of size ALPHA. An INTERVALSIZE interval estimate about the mean is produced. The results are written to OUTFILE object OUTPUT. The autoregressive coefficients, interval estimates and the sample variance are computed for all orders up to Q.

attributes

long real array B(I,J);	the autoregressive coefficients for the Ith order representation
long real array R(I);	autocovariance function
long real array RHO(I);	serial correlation coefficient
long real array V(I);	sample variance of sample mean for the Ith order representation
long real array G(I);	spectrum of data (indexed in units of pi/64);
long real array F(I);	degrees of freedom in the Ith order representation
long real array T(I);	test statistic for chi-square test for the Ith order representation
long real array CRITICAL (I);	critical value of the chi-square test for Ith order representation
character array RHOGRAPH;	graph of correlogram
character array SPECTRUMGRAPH;	graph of spectrum
integer Q;	maximum order representation computed
real array LOWER(I),UPPER(I);	end points of confidence interval for Ith order representation
integer P;	order of representation chosen by test

procedures

COMPUTEAUTOCOVARIANCE;	computes the autocovariance function up to lag Q.
COMPUTEB;	computes the autoregression coefficients for each order representation up to Q. [3, Eq. 5.61]
COMPUTESIGMA2;	[3, Eq. 5.62]

COMPUTET;	computes the test statistic for determining the order of the autoregressive representation. [3, Eq. 5.64]
COMPUTEVBAR;	computes an estimate for the sample variance of the sample for each order representation. [3, Eq. 5.54]
COMPUTEF;	estimates the number of degrees of freedom for each order representation. [3, Eq. 5.74]
COMPUTESPECTRUM;	computes the spectrum of the input data. [3, Eq. 5.90]
COMPUTERHO (LAG); integer LAG;	estimates the correlogram. For LAG greater than the order of the autoregressive representation, [3, Eq. 5.77] is used. Otherwise, Equation 10-15 is used.
GRAPHRHO (FIRSTLAG, LASTLAG, MAXRHO); integer FIRSTLAG, LASTLAG; real MAXRHO;	produces a character array containing a graph of the correlogram for all lags between FIRSTLAG and LASTL The correlation coefficients for lags which exceed MAXRHO are not displayed.
integer procedure TRANSIENT(VAL); real VAL;	computes the number of observations required for the I VAL. [3 eqs. 5.79, 5.82]
procedure COMPUTECRITICALVALUE;	computes the critical values of the chi-square distribution.
procedure COMPUTEINTERVALESTIMATE	computes the interval estimates for each representation up to order Q.
integer procedure AUTOCORRANGE (VAL); real VAL;	returns the value of the first lag for which the serial correlat coefficient is less than VAL.

procedure GRAPHSPECTRUM;	puts the graph of the spectrum into a character array.
procedure OUTPUTRESULTS;	writes a summary of the autoregressive analysis to the output file.
procedure COMPUTEP;	computes the autoregressive order via [3, p.251].

10.3.4.2 CLASS BATCHMEANS (BETA, INTERVAL, DATA, TYPE, OUTPUTRESULTS); REAL BETA, INTERVAL; REF (FILE) DATA; INTEGER TYPE; REF (OUTFILE) OUTPUTRESULTS;

When a BATCHMEANS object is created, the batch means simulation analysis is performed on the data contained in FILE object DATA (see sec. 10.1). INTERVAL interval estimates are computed. The test for independence of the batch means is performed at the BETA significance level. The test for independence is a one-sided or two-sided test (TYPE = 1 or 2) depending upon whether or not the autocovariance function of the original data is monotonically decreasing. The results of the analysis are written to OUTFILE object OUTPUTRESULTS. The algorithm as implemented is described by Fishman [3, pps. 237-247].

attributes

boolean IIDDATA;	are batch means iid?
real LOWER, UPPER;	end points of confidence interval
integer FIRSTIIDDATA;	number of observations in first iteration for which the batch means are independent

```
ref (IIDTST) IIDRESULT;        IIDTST object for
                               examining independence of
                               batch means
```

10.3.4.3 TRANS.SIM

Performs the analysis of the transient period via the cumulative moving average method (see sec. 10.2).

10.3.4.4 TRUNC.SIM

Deletes the first n observations from a file (n is prompted for).

References

1. Abramowitz, Milton and Stegun, Irene, Handbook of Mathematical Functions, Dover Press, 1965.

2. Allen, Arnold O., Probability, Statistics, and Queueing Theory, Academic Press, 1978.

3. Fishman, George, Principles of Discrete Event Simulation, Wiley-Interscience, 1978.

4. Kleijnen, Jack P. C., Statistical Techniques in Simulation, Marcel Dekker, Inc., New York, 1974.

5. Kobayashi, Hisashi, Modeling and Analysis, Addison-Wesley, 1978.

6. Makhoul, John, "Linear Prediction: A Tutorial Review," Proc. IEEE, vol. 63, April 1975, pp. 561-580.

7. Sargent, Robert G., "Statistical Analysis of Simulation Output Data," Simuletter, vol. 8, no. 3, April 1977.

11. RANDOM NUMBER GENERATOR

Central to the acquisition of useful results from a simulator is the ability to generate numbers which have the properties of a random sequence. For a given simulation run, the random number generator will be called upon to determine the interarrival times between packets, the length of packets, the execution times for certain IMP processes, when line errors occur, etc. For some simulation experiments it is not unrealistic to assume that we will generate 10^5 packets and make as many as 10^2 calls to the random number generator for each packet. Thus, one can estimate that, for some simulation runs, as many as 10^7 calls will be made to the random number generator.

The quality (or lack thereof) of the random number generator impacts the results of simulation experiments in innumerable ways. If the generator cycles during the experiment, then the output data may possess correlations which are not properties of the model under investigation, but artifactual results. Such correlations will undoubtedly impact the accuracy of interval estimates, although the errors may not be easily detectable. If the sequence of numbers produced by the random number generator is not sufficiently random or has a distribution different from that expected, then similarly erroneous results can occur. Thus, it is worthwhile to devote time to studying the properties of the SIMULA generator.

Any given call to a random number generator specifies a particular distribution. For example, calls to generate message interarrival times specify an exponential distribution, calls to produce line errors (as currently implemented) specify a uniform distribution. In all cases, however, the distribution is produced by first generating a stream of deviates uniformly distributed in the interval (0,1) and then transforming these uniform deviates to produce a stream of variates with the desired distribution function. This transformation can be accomplished in the following manner [4]. Assume X is a random variable with distribution function F_X. Define F_X^{-1} as the inverse of F_X such that

Eq. 11-1 $$F_X^{-1}(y) = \min\{x:F_X(x) > y\}$$

If U is a random variable uniformly distributed as on (0,1) then if Y is defined as

Eq. 11-2 $$Y = F_X^{-1}(U)$$

we have

Eq. 11-3

$$\text{Prob}\,[Y \leq y] = \text{Prob}\,[F_X^{-1}(U) \leq y]$$
$$= \text{Prob}\,[U \leq F_X(y)] = F_X(y) = \text{Prob}\,[X \leq y]$$

Thus Y and X have the same distribution. For example, to produce an exponential distribution, we produce uniform deviates on (0,1) and take the negative logarithm.

From the above discussion, we can conclude that the quality of the random number generator depends roughly on the following criteria. Does it produce "enough" (in our case $>10^7$) random variates. Are these variates uniformly distributed on the interval between 0 and 1? Are they distributed in a random fashion? We have devoted a fair amount of effort in attempting to answer these questions for the SIMULA random number generator.

The SIMULA random number generator is of a class called multiplicative congruential generators. Such generators produce variates recursively according to

Eq. 11-4

$$Z_i = aZ_{i-1} \pmod m$$

In the SIMULA generator, $a = 5^{15}$ and $m = 2^{35}$. In order to produce random numbers between 0 and 1, one forms the sequence

Eq. 11-5 $$U_i = Z_i/m$$

Since there are 2^{35} non-negative integers which can possibly be formed through the application of Equation 11-5, the sequence $\{Z_i\}$ has 2^{35} possible members. However, not all of these numbers can be generated in the same sequence. That is, for any particular choice of Z_0, the period of the resultant sequence will be less than 2^{35}. In addition, not all sequences will be equally random. In practice, one wants the largest possible period, both to guarantee the ability to make a large number of calls to the generator and to ensure that the $\{U_i\}$ are sufficiently dense on (0,1).

The properties (period, uniformity, randomness) of a multiplicative congruential generator are determined by the choice of the coefficients and initial seed, Z_0. There are three categories of methods that can be used to investigate the properties of multiplicative congruential generators. The first category involves determining, via number theoretic methods, criteria for the selection of a and m. The second category involves theoretical tests of the properties of a sequence of

variates produced by the generator. The third category involves actual empirical testing of a generated sequence.

It may be shown via number theoretic methods [4] that the sequence $\{Z_i\}$ has a maximal period of m/4 if

Eq. 11-6 $$a = 3 \text{ or } 5 \pmod 8$$

it can be computed that

Eq. 11-7 $$5^{15} = 5 \pmod 8$$

so that the SIMULA generator has period $2^{35}/4$ or approximately 10^{10}. Kobayashi [4] indicates that to achieve the maximal period one requires an odd integer seed. We have arbitrarily chosen as the nominal seed for experiments the value 314159. In his discussion on random number generators, Kobayashi also cites a result due to Coveyou and Greenberger that the first order serial correlation coefficient, ρ_1, for a multiplicative congruential sequence lies between $a^{-1} \pm a/m$. Since we want ρ_1 to be close to zero, this suggests that $a = \sqrt{m}$ which results in $\rho_1 \approx 0$. In the SIMULA generator, we have $a \approx m$. While this result might seem troubling at first. Kobayasni states that generators

with $a \approx \sqrt{m}$ have been demonstrated to have undesirable randomness properties despite the low first order serial correlation. If successive 3-tuples produced by such generators are plotted, they are found not to be uniformly distributed in the unit cube but rather to lie on 15 parallel planes. However, because Coveyou's result raises questions about the first order correlation coefficient in the SIMULA-generated sequence, we have implemented an empirical test which is described below.

Of the theoretical tests for random number generators, the most important and well-known is the spectral test. The theory behind this test is extensively discussed by Knuth [3]. The spectral test is used to examine the relative merits of alternative generators. It produces a sequence of quantities, C_k, which measure the randomness of adjacent k-tuples in the sequence $\{U_i\}$. In practice, C_k is computed out to a maximum of $k = 4$. Fortunately, Knuth has applied the spectral test to the random number generator in SIMULA. Of 13 generators to which he applied the test, he found that the SIMULA generator was the best.

Finally, there is a series of empirical tests which we have applied to sequences of numbers produced by the SIMULA generator. Since no generator is "perfect," the empirical tests are designed to determine if there is a significant departure of the generated pseudorandom sequence from "true" random behavior. In

particular, three hypotheses are investigated for a sequence of variates $\{U_i\}$ [2]

H_1: $U_1,\ldots,U_n$ are independent and identically distributed.

H_2: $U_1,\ldots U_n$ each have U(0,1)

H_3: k-tuples formed from adjacent observations in $U_1,\ldots U_n$ are uniformly distributed in the k-dimensional hypercube.

We have implemented at least one test for each of the above hypotheses. The results are described below.

Chi-Square Test

The chi-square statistic [3] tests the hypothesis that $U_1,\ldots,U_n$ are uniformly distributed on (0, 1). In performing the test, one divides (0, 1) into k non-overlapping subintervals and forms the test statistic

Eq. 11-8
$$V = \frac{1}{n} \sum_{s=1}^{k} \left(\frac{Y_s^2}{P_s}\right)^n - n$$

where Y_s is the number of observations falling into the Sth subinterval and P_s is the size of that interval. The statistic V should be distributed according to the chi-square distribution with k-1 degrees of freedom. On a given run of the test, we therefore accept the hypothesis at the α level of significance if V lies between the $\frac{\alpha}{2}$ and $1-\frac{\alpha}{2}$ percentile values of the chi-square distribution.

We performed 50 independent replications of the chi-square test of size .10 with k = 1000 and n = 200,000. On these 50 replications, the test statistic V assumed a value outside of the acceptance region 5 times. This is exactly what one would expect under the hypothesis of uniformity for a test of size .10.

Since the V statistic is chi-square distributed under the hypothesis tested, a more suitable method of evaluating the test results is to examine whether or not the sequence of 50 test statistics does indeed have the chi-square distribution with 999 degrees of freedom [2]. In order to do this we applied a "goodness of fit" test known as the Kolmogorov-Smirnov test [1] to the sequence of V's. We rejected the hypothesis that the V are chi-square at the .01 level. This result implies that there is less than a 1 in 100 chance that the test statistic does have the expected distribution, a disappointing result which we are currently investigating. The results of the chi-square test are shown in Figure 11-1.

replication	statistic	replication	statistic
1	1051.86	26	1051.52
2	1000.14	27	1033.06
3	1038.59	28	950.37
4	952.57	29	1035.04
5	1029.26	30	1060.41
6	983.49	31	1073.94
7	1014.51	32	1031.18
8	965.62	33	1051.20
9	1005.16	34	1030.69
10	974.08	35	1039.60
11	976.50	36	1055.94
12	1030.48	37	1027.84
13	1104.91	38	1082.73
14	1011.53	39	1011.09
15	1036.02	40	950.73
16	1031.04	41	1080.07
17	1024.73	42	985.21
18	986.16	43	999.93
19	1028.65	44	945.99
20	1017.51	45	1033.63
21	1021.71	46	1033.42
22	1057.12	47	924.97
23	977.17	48	997.67
24	1006.67	49	978.32
25	1037.22	50	991.72

Acceptance Region for .10 size chi-square test = [926.62 , 1073.66]

Kolmogorov-Smirnov Test Statistic = .243

Kolmogorov-Smirnov .01 Critical Value = .231

Figure 11-1 Chi-Square Test Results

One subtlety of the chi-square test should be pointed out. The test statistic V will be chosen from the chi-square distribution with k-1 degrees of freedom under the hypothesis of uniformity only if the original $\{U_i\}$ observations are independent. Hence the chi-square test "proves" uniformity by assuming independence.

Serial Test

The serial test statistic [2] tests the assumption that adjacent k-tuples are uniformly distributed in the k-dimensional unit hypercube. As such, it is a generalization of the chi-square test While the test is, in theory, applicable for any value of k, the size of computer memories generally makes 2 a practical limit. Thus, the serial test as implemented tests the hypothesis that 2-tuples formed from adjacent pairs in the sequence $U_1,\ldots,U_n$ are uniformly distributed on the unit square.

In each replication of the test, we divided the unit square into 128X128 subsections, generated 200,000 random numbers, and formed 100,000 ordered pairs from the generated sequence. We then determined how many ordered pairs fell into each of the subsections. If we denote this number as Y_i for the ith subsection, then the test statistic is given by

Eq. 11-9 $$S = \frac{128*128}{200000} \sum_{i=1}^{128*128} \left(Y_i - \frac{200000}{128*128}\right)^2$$

According to the theory of the test, statistic S should have the chi-square distribution with 128*128-1 degrees of freedom.

On 50 independent replications of the test at the .10 level, the test statistic fell into the rejection region 5 times (Fig. 11-2). This result is consistent with the hypothesis that adjacent 2-tuples formed from $U_1,\ldots,U_n$ are uniformly distributed on the unit square. We also tested the hypothesis that the set of 50 values of the test statistic is chi-square and found that we could accept the hypothesis at the .05 level of significance.

Runs up/down

The runs up and runs down tests [2,3] are used to examine the hypothesis that the $U_1,\ldots U_n$ are independent and identically distributed. In what follows, we describe runs up (down). We generate a sequence of n random digits. We then form a related binary sequence by replacing each U_k by 1 if $U_k > (<) U_{k-1}$ and 0 if $U_k < (>) U_{k-1}$. Each run of 0 followed by k consecutive 1's represents a "runs up (down)" sequence of length k+1. For each k from 1 to 5, let C_k represent the number of runs up (down) of length k. Let $C(6)$ represent the number of runs up (down) of length greater than 5. Then the test statistic is

replication	statistic	replication	statistic
1	16365.39	26	16210.73
2	16528.91	27	16257.26
3	16026.57	28	16223.84
4	16423.39	29	16989.30
5	16322.80	30	16445.02
6	16333.94	31	16254.64
7	16244.15	32	16591.82
8	16215.63	33	16258.24
9	16244.49	34	16192.38
10	16284.46	35	16435.85
11	16235.30	36	16807.11
12	16199.26	37	16238.25
13	16402.75	38	16347.04
14	16330.99	39	16358.84
15	16189.10	40	16396.52
16	16508.59	41	16431.59
17	16380.47	42	16196.97
18	16630.49	43	16179.93
19	16567.24	44	16204.83
20	16020.35	45	16313.95
21	16487.62	46	16190.74
22	16458.78	47	16546.60
23	16210.07	48	16612.14
24	16038.70	49	16601.65
25	16284.78	50	16271.68

Acceptance Region for .10 size test = [16086.34 , 16681.94]

Kolmogorov-Smirnov Test Statistic = .185

Kolmogorov-Smirnov .05 Critical Value = .192

Figure 11-2 Serial Test Results

Eq. 11-10 $$V = \frac{1}{n} \sum_{i,j=1}^{6} (C(i)-nb_i)(C(j)-nb_j)a_{ij}$$

Where the b_i and a_{ij} are given in [3]. This statistic should have the chi-square distribution with 6 degrees of freedom.

On 50 replications of the runs-up test at significance level .10, the critical value fell into the rejection region 3 times, somewhat better than might be expected (Fig. 11-3). When the Kolmogorov-Smirnov test was run to examine the hypothesis that the set of 50 V statistics did have the chi-square distribution with six degrees of freedom, we found that we could accept the hypothesis at the .20 level. On 50 replications of the runs down test, the test statistic fell into the rejection region six times (Fig 11-4). Once again, we found that the Kolmogorov-Smirnov test allowed us to accept the hypothesis of chi-square distributed V at the .20 level.

Periodogram

Thus far, none of the tests described is sensitive to possible periodicities in the sequence of random numbers. The tests of uniformity are totally insensitive to the order of the sequence $U_1,\ldots,U_n$. The runs test would have a significant amount of difficulty detecting cyclic patterns. Hence, we require another empirical test to study the degree of correlation

replication	statistic	replication	statistic
1	7.77	26	4.27
2	8.90	27	3.95
3	7.84	28	2.58
4	4.41	29	4.69
5	7.29	30	11.94
6	1.67	31	2.42
7	5.15	32	8.67
8	4.30	33	11.09
9	3.55	34	4.76
10	4.91	35	6.90
11	9.98	36	2.74
12	9.22	37	6.86
13	5.75	38	4.26
14	13.57	39	11.50
15	3.47	40	4.83
16	10.35	41	2.83
17	6.03	42	9.87
18	8.00	43	3.22
19	4.00	44	5.86
20	2.20	45	7.09
21	12.77	46	6.01
22	4.58	47	3.90
23	4.30	48	8.40
24	0.94	49	4.40
25	4.79	50	8.94

Acceptance Region for .10 size chi-square test = [1.63 , 12.59]

Kolmogorov-Smirnov Test Statistic = .110

Kolmogorov-Smirnov .20 Critical Value = .151

Figure 11-3 Runs Up Test

replication	statistic	replication	statistic
1	7.59	26	4.24
2	6.36	27	15.47
3	4.69	28	5.05
4	5.21	29	4.89
5	8.05	30	4.72
6	0.89	31	4.50
7	13.02	32	8.42
8	2.68	33	8.66
9	1.06	34	6.55
10	4.97	35	9.01
11	6.75	36	1.02
12	2.99	37	3.52
13	1.19	38	12.43
14	11.45	39	3.18
15	11.02	40	4.69
16	9.13	41	5.01
17	4.18	42	11.73
18	3.81	43	6.33
19	9.14	44	12.18
20	2.95	45	7.60
21	3.27	46	4.22
22	6.74	47	4.82
23	3.78	48	7.57
24	6.69	49	2.81
25	5.96	50	12.38

Acceptance Region for .10 size chi-square test = [1.63 , 12.59]

Kolmogorov-Smirnov Test Statistic = .082

Kolmogorov-Smirnov .20 Critical Value = .151

Figure 11-4 Runs Down Test

in the $\{U_i\}$. Such a test assumes particular importance for the SIMULA generator given the result of Coveyou and Greenberger discussed above.

One can study correlations in the U_i using the correlogram. However, the results of calculating the estimated serial correlation coefficients up to some large order are difficult to interpret. That is, there is no obvious test that allows one to accept or reject the hypothesis in question based upon the computed correlogram [2]. Instead, we investigate the spectrum of the random sequence. The periodogram is one means of examining this spectrum [2,4].

If one generates $X_1,\ldots,X_n$ and forms

Eq. 11-11

$$A_m = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} X_i \cos\left(\frac{2\pi i m}{n}\right)$$

$$B_m = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} X_i \sin\left(\frac{2\pi i m}{n}\right)$$

then the sequence

Eq. 11-12 $$I_m = A_m^2 + B_m^2 \qquad m = 1,n/2$$

is called the periodogram. If the $\{U_i\}$ are independent and identically distributed, then the $\{I_m\}$ are exponentially distributed.

In conducting this test, we generated, on each replication, a sequence of 5000 numbers and computed $\{I_m\}$. If the $\{I_m\}$ are exponentially distributed, then the normalized cumulative periodogram $\{S_m\}$ formed by

Eq. 11-13 $$S_m = \frac{\sum_{i=1}^{m} I_i}{\sum_{i=1}^{n/2} I_i}$$

should be uniformly distributed. We use the Kolmogorov-Smirnov test to determine whether this is, in fact, the case. The test statistic is

Eq. 11-14 $$D = \max(|S_k - 2k/n/2|)$$

If the value of D is greater than the Kolmogorov-Smirnov critical value, we reject the hypothesis that the $U_1,\ldots,U_n$ are i.i.d. If D is less than the critical value, we accept the hypothesis. On eleven replications of the test, we accepted the hypothesis of uniformity at the .20 level (Fig 11-5).

Distributions

The final hypothesis tested was that the SIMULA exponential generator does produce variates according to an exponential distribution. In order to do this, we generated 5000 numbers, exponentially distributed with mean 1.0, on each of 5 independent replications. We tested for uniformity using the Kolmogorov-Smirnov test at the .20 significance level and accepted the hypothesis on each replication. The results of the test are displayed in Figure 11-6.

replication	seed	test statistic	accept hypothesis?
1	314159	.01081	yes
2	14473533199	.00894	yes
3	28550536943	.01263	yes
4	24327905999	.01098	yes
5	1601135279	.01859	yes
6	21854494351	.01817	yes
7	27838846575	.01588	yes
8	22852979279	.01320	yes
9	25612892719	.01713	yes
10	6867478031	.01804	yes
11	2012148175	.01458	yes

.20 critical value = .0214

Figure 11-5 Periodogram Test

replication	seed	test statistic	accept nypothesis
1	314159	.00676	yes
2	14473533199	.01084	yes
3	28550536943	.01106	yes
4	24327905999	.00975	yes
5	1601135279	.00940	yes

.20 Kolmogorov-Smirnov Critical Value = .0513

Figure 11-6 Test for Exponential Distribution

References

1. Allen, Arnold O., Probability, Statistics, and Queueing Theory, Academic Press, 1978

2. Fishman, George, Principles of Discrete Event Simulation, Wiley-Interscience, 1978.

3. Knuth, Donald, The Art of Computer Programming: Volume 2/Semi-Numerical Algorithms, Addison-Wesley, 1971.

4. Kobayashi, Hisashi, Modeling and Analysis, Addison-Wesley, 1978.

APPENDIX 1. USING SIMULA

A.1.1 Reasons for Simula

Although Simula was designed for programming simulations, it includes a general-purpose programming language as a base. It is this base language which is the primary justification for the choice of Simula for our simulation. In this section we discuss the base language; the simulation extensions will be discussed below.

The Simula base language is derived from Algol '60. In fact, apart from a small number of minor incompatibilities, Algol '60 is a subset of Simula. The major addition to Algol has been a set of modularization and encapsulation facilities. These facilities have been provided with a single syntactic construct, the CLASS. Each class is a definition of (or template for) an object. An object is an instance of a class.

Objects may contain data, procedures or code. The data is accessed, or the procedure called, using dot notation:

A.B

means that B field (or procedure) which is part of the object A. These references may be conveniently cascaded:

A.B.C.D

means the D of C of B of A. Similarly, just as it is possible to define arrays of integers, or functions that return integers, it is possible to define arrays of objects or functions that return objects. It is an important property of Simula that user-defined classes are just as easy to use as built-in types.

Using the CLASS construct, it is easy to define sophisticated queue handling facilities. This is important in our simulation. For example, we have defined a class for queues of packets which includes the function "wait until a packet is put on the queue then remove and return it." Using Simula's encapsulation facilities, this function can be included as part of the definition of the class queue, so that any change to the structure of the queue is invisible outside the class. For example, during the development of the simulation, the packet queues were changed from ordinary FIFO queues to priority queues. This required recoding the routine which adds an element to a queue (adding 20 lines of code to accommodate the extra complexity of a priority queue) and not one change to the rest of the program.

The idea of a process is an integral part of the base language, and was not added as part of the simulation extensions. A process is just an object that contains code. Simula

automatically includes a pointer which indicates where the process is in executing the code. The code is automatically started when the object is created, and the process can then transfer control to another process, or back to the main program. Simula will remember where the process was when it left off, so when control is transferred back to it, it will start again from the right place.

It is important that a process really is just an ordinary object, so that all the operations which are possible for objects are possible for processes. For example, just as it is possible to create a queue of packets, it is possible to create a queue of processes. This enabled us to write a process scheduler which simulated pre-emptive interrupt driven scheduling in the IMP.

It is a basic assumption of all simulation languages (including the Simula extensions) that a process which uses t seconds of a resource takes t seconds of simulated time. Unfortunately, this is not true when there may be other, higher priority processes running at the same time; that is why we had to write our own process scheduler.

Interprocess communication is very flexible in Simula. The language really imposes no restrictions on how processes communicate. In our simulation, in various places, processes communicate via global tables, procedure calls, and queues of packets.

In conclusion, Simula provides a powerful, convenient base language because it has added to Algol '60 a small number of constructs with wide application.

A.1.2 The Simula CLASS

A variable has a type (at compile time) and a value (at run time). The built-in types are integer, real, boolean and character. Every class definition defines a new type (possibly a subtype of some other class). One can define a variable of some class (i.e., of some class type), in which case the value of the variable is a pointer to an object of that class (i.e., of that type, or possibly some subtype) in the same way that the value of an integer variable is some integer. We can consistently say that both the variable and the value have type integer, and so it is with classes. Arrays of any type may be defined.

The word object refers to an instance of a class. Naturally enough, the definition of a class defines exactly what is in an object of that class (type). A class can be defined with any or all (or even none) of the following:

- an argument list (used when an object is created)
- variables (i.e., local variables)
- procedures
- code

For a given class, one can create an instance of that class or define a variable which refers to instances of that class (and only that class). When one creates a new object, one must give the actual arguments if the class was declared with an argument list, and one gets back a value which can be assigned to a variable of the right type.

For a given object (if one has a pointer to it, that is), one can:

- access (read/write) the variables
- call one of the object's procedures
- transfer control to the object's code (see below)

Both the procedures and the code are scoped INSIDE the object. That is, they can refer to the object's variables as ordinary variables without going through a pointer. Thus a procedure can, and typically will, have different effects and/or results depending on the particular object it belongs to. In this sense one can think of the main program as being a separate object.

Suppose one defines a class item:

```
class item ;
   begin
   ref (item) next ;
   integer key ;
   end ;
```

then every item contains two variables, next and key. Next is a pointer to another item (ref means pointer to), and key is an integer variable. One can use the class "item" as follows:

```
ref (item) head ;          ! defines a variable of type item ;
head :- new item ;         ! assigns a new item to head ;
head.key := 1 ;            ! assigns 1 to its key variable ;
head.next :- new item ;    ! assigns another new item to next ;
head.next.key := 2 ;       ! assigns 2 to its key variable ;
```

The only difference between ":-" and ":=" is that the former is used for pointer (reference) assignment.

A.1.3 Communicating with Data Structures

Data structures may usefully be implemented as separate objects. The advantages of this are:

- they have their own name space (i.e., set of local variables), which may be hidden.
- all the procedures which manipulate the structure can be collected together inside the object.
- it is easy to create separate, completely independent copies of the data structure.

There are several ways of defining operations on data structures. Suppose we have defined classes queue and message, and we have a queue variable (i.e., pointer), q, and a message, msg. Then adding the message to the queue might be defined as:

- an ordinary procedure Add(q, m), which has access to the local variables of q.

- a procedure within the class queue, invoked by a call q.Add(m)

- a procedure within the class message, invoked by a call m.GoesInto(q)

The second method is preferable and is used in our simulation since it enforces independence between separate classes.

A.1.4 Class Prefixing

A class may be used to define another class. For example:

```
class packet
   begin
   integer length, number;
   boolean error;
   ...
   end;
```

defines a packet with a length (in bits), a packet number, and a flag which is set to indicate a line error. Then we can define a data packet in terms of class packet:

```
packet class dataPkt
   begin
   integer source, dest;
   real netEntryTime;
   ...
   end
```

which defines a data packet to be a packet which in addition to length, number and error flag has a source and destination IMP number, and the time when it entered the network.

Other types of packet can be defined. For example, a routing update packet looks like:

```
packet class updatePkt (nLines); integer nLines;
   begin
   integer age, serialNumber, impNumber;
   boolean retry;
   real array del [1:nLines];
   end
```

Note that updatePkt is parameterized by the number of lines in the IMP for which the delay is being reported. The age, serial number and retry fields are used in the routing update protocol; the array del contains the delay out each line for the specified IMP.

In general, if Y is defined with X as a prefix, then Y is said to be a subclass of X. This is because every Y is also an X, so the set of Y's is a subset of the set of X's. Do not be confused by the fact that every instance of Y contains an instance of X.

As a consequence, if a data structure or routine has been defined to operate on objects of type X, then it will also handle objects of type Y. It is this technique which is used to extend Simula to include most of the simulation features. First the

main block is prefixed with the class SIMULATION, which means that all of the things defined inside SIMULATION are available inside the main program. Then a class definition can be prefixed by PROCESS, which means that not only does the body of the class have access to all the routines defined inside PROCESS, but also that any routine anywhere else which has been defined to operate on PROCESSes will work on this new class.

A.1.5 Parallel Processing

When an object is created, control is transferred to the code inside the object. In the default case the code is used for nothing more than initialization, and when the code is finished control is returned to the point of creation.

However, the object may transfer control back to the point of (its) creation in such a way that its local state (i.e., stack and program counter) is preserved. In this case, control may be transferred back to the object at any time (by anyone who has a pointer to the object). Call an object in this state a process. Note that the main program is a process.

When one transfers control to a process, it is restarted at the point it left off, like returning from a subroutine call. There is no facility for transferring control to some arbitrary point within the process (and a good thing too !).

If two processes have pointers to each other, they can transfer control back and forth. At any time, however, there is precisely one process running. Control can only be transferred explicitly.

A.1.6 Communication Among Processes

Processes may, of course, communicate via global variables. But if one process has a pointer to another process, then the first process may either:

- alter a local variable in the second process by an explicit assignment or by calling a procedure to do it,
- or call a procedure in the second process, passing information through the argument list.

Although Simula allows one object to tinker with the local variables of another object, this is probably not a good idea. An object may declare some or all of its variables to be hidden, so that they cannot be accessed from outside the object itself. This forces all accesses to be via calls to procedures within the object. This approach is recommended, since then the internal data structures of the class, or the code used to manipulate them, can be changed without affecting any code outside the class (unless the procedure calling sequences change, of course).

Most communication between processes in our simulation is implemented by procedure calls, but in some cases a process may read a variable from another process directly. This is particularly true in the case of the line protocol, which is implemented with code in the task, modem input, and modem output processes. A process never changes a variable in another process directly. This is always done with a procedure call.

A.1.7 Application to our Problem

In designing our simulation at the top level, we had to map entities in the simulation onto processes, objects, procedures and so on. It is important to realize that there is a very fine line between active and passive objects in Simula. Unless one actually tries to transfer control to it, it is impossible to tell whether or not an object is a separately running process. Thus if one calls a procedure inside a queue object to put a message on the queue, one has no way of knowing (presumably one doesn't care) whether the queue is a simple data structure, an object which is collecting statistics on each message, or an active process which is responsible for sending the message on (possibly based on some complex calculation).

Now suppose that we are simulating two IMP routines that wish to communicate via some data structure, such as a queue or table. Presumably the IMP routines correspond to Simula

processes. If the data structure is a separate object, which handles objects of some particular class, then the communication between the two routines can be made independent of the format of the information being passed, and transparent to the two routines. Suppose the data structure is a hash table with two operations, Enter and Search, which store and retrieve objects of class Item (defined above). Then the two IMP routines can communicate via objects of class particularItem:

```
Item class particularItem ;
   begin
   ! all the information that is to be
     transferred between the two IMP's ;
   end ;
```

Then one IMP routine can pass a particularItem to Enter, which will take it as an item and store it in the table, and the other routine can call Search and get it back.

The data structure routines do not know anything about what goes into a particularItem; they are only interested in dealing with Item's. The two IMP routines do not have to know anything about the internal structure of the hash table, or the routines that manipulate it (except the name of the key field in an Item). Thus the data structure can be changed without changing the IMP routines, and the IMP routines can agree to exchange different kinds of information without changing the data structure routines. It is also possible for one IMP routine to store and

retrieve extra fields in a particularItem which the other IMP routine does not look at.

A.1.8 A Problem

There is a problem with the above discussion. When a particularItem is entered into the hash table, it is no longer obvious to the compiler that it is a particularItem and not just an Item, so when Search returns the object, the type of the function value is Item, even though the type of the object is particularItem. Simula requires an explicit check that the Item returned is indeed a particularItem before any of the extra fields in it can be accessed. The problem arises because Search has been declared as a function that returns an Item, and Simula does not do the flow analysis that would allow it to conclude that if every item that goes into the hash table is a particularItem, then so is every Item that comes out. This is an annoyance more than anything else. In the simplest case, when one knows that the Item is a particularItem (and not some other special sort of Item), it requires an extra two words to say it:

```
Search (...) qua particularItem
```

instead of:

```
Search (...)
```

Note that there is the overhead of a runtime check.

A.1.9 Queues and Scheduling

Simula provides skeletal facilities for queueing and scheduling. The choice was made to abandon the built-in queueing mechanism, but to build on top of the existing scheduling facilities. The built-in queueing operations use a very clumsy syntax, are inconvenient to extend, and can easily be rewritten in a small amount of code. The scheduling operations, on the other hand, provide a minimal set of primitives with a convenient syntax. They are easy to extend, but would have required a reasonable amount of effort to recode. Either decision can be changed easily because each is implemented by a Simula CLASS. The implementation of queueing or scheduling can be changed without affecting the user interface, which is a set of functions within the class.

In our application, there is extensive interaction between queueing and scheduling. The basic queue operations of Add and Remove are supplemented by combined queueing/scheduling operations: AddWait adds an element to a queue and forces the process to wait until it has been removed from the queue; RemoveWait forces the process to wait until an element has been put on the queue, then removes and returns it. Because the basic task of the IMP is to process packets, Add and RemoveWait are the

basic mechanisms both for communication and scheduling among processes.

Another type of process is clock-driven. It is woken at each tick of a clock, performs some predetermined functions, then waits until the next clock tick. Simula provides a basic scheduling operation, "wait for t seconds of simulated time," which can be used to implement the regular ticking of a clock.

The final scheduling primitive is the function which simulates execution on an IMP CPU. Because there are many processes running at once, at several priority levels, the time a process takes to execute t seconds worth of CPU cannot be determined in advance; it will be t seconds plus the time taken by any higher priority processes which have interrupted it. The scheduler, which provides a function to delay a process by the time it takes to execute t seconds CPU at a given priority level, must simulate an interrupt stack and a priority queue.

A.1.10 Class Simulation

The extra features that make Simula a simulation language are contained in two predefined classes, Simset and Simulation. By prefixing a program with the class Simulation (which itself contains Simset), one gets all the definitions in the class included in the program in the same way that a particularItem gets everything that was declared in Item.

Simset contains declarations for class LINK, which is an element of a two-way list, and class HEAD, which is the head of a two-way list. Also included are routines for manipulating these lists. Class SIMULATION contains the definition of LINK class PROCESS, which is just a process in the sense we have been using the word, together with some routines for doing scheduling using simulated time.

Thus if one defines a class which is prefixed by PROCESS, one can create instances of the class which can be used with all the routines in SIMULATION, as well as all the routines in LINK for making two-way lists.

A.1.11 Details of Class PROCESS

In this section we will give a description of the class PROCESS. With the single exception of the syntax of the activation statements, all of the extensions for simulation can be coded in Simula. Ohlin [1] has written these extensions in about 200 lines of Simula. The activation statement could be defined in Simula as a procedure, except that as presently defined it would need optional arguments, which Simula does not support. It is important for us that the simulation features have this property, since we may want to recode them for efficiency, or to extend the fairly limited set of primitives provided by Simula.

Remember that a PROCESS is a process, that is, it is an object which has been set up so that control can be transferred to it. A process thus has a local stack, and a program counter. When control is transferred to a process, the computation is resumed at the stored value of the program counter. It is not possible to restart a process at some arbitrary point, but only at the point where it left off last time.

A PROCESS is a LINK. All a LINK is, is a pointer to the next LINK in a linked list, plus some routines for adding and removing the LINK from lists. Thus these operations are also included in a process. They are used by the scheduler and (hence) are not accessible to the user.

A PROCESS also includes a real variable which stores the time at which the process is to be woken up (possibly infinite), a flag which tells whether or not the PROCESS is scheduled (or running), and another or not which tells whether it is alive. A process can be: running (ACTIVE), scheduled (PENDING), unscheduled (IDLE), or dead (TERMINATED). A PROCESS becomes terminated when it exits its outermost block and one can no longer transfer control to it. The ACTIVE PROCESS is at the head of the scheduling list. All PENDING PROCESSes are on the scheduling list in order of their wake-up time. Note that a PROCESS must exist on some list, or as the value of some variable, or it will cease to exist, and will be garbage-collected.

CURRENT returns a pointer to the currently active PROCESS (at the head of the scheduling list). TIME returns the time. PASSIVATE puts the CURRENT PROCESS to sleep, and wakes up the next PROCESS (the CURRENT PROCESS is pulled off the head of the scheduling list; CURRENT is changed to be the new head; TIME is advanced to the wake-up time for this PROCESS; and control is transferred to it, using RESUME). Note that PASSIVATE does not return the PROCESS that called it back into the scheduling list. Thus the PROCESS which called PASSIVATE may be made inaccessible by the call, in which case it will be garbage-collected.

HOLD(DEL) does a PASSIVATE, but also inserts the process into the scheduling list, to be woken at TIME + DEL. All of the ACTIVATE statements simply add a process (possibly the process doing the ACTIVATE) to the scheduling list at some specified time, or before or after some other process. CANCEL removes a process from the scheduling list as though it had done a PASSIVATE itself. All of these operations simply involve searching the list and adding or removing a process.

A.1.11 A Note on Queues

There is an obvious operation which is not provided in Simula, but is easy to add. This is a message wait. We want an operation which puts a process to sleep until a message arrives in a given queue. A variable must be added to the queue which

specifies the process to be woken up, and Add must be changed to wake up the process when someone adds a message to the queue.

Assuming that a class queue already exists, with functions Add, Remove, and Empty (in Simula, HEAD and LINK perform this function, but not well), the code shown in figure A.1-1 will define a new class with the desired properties.

A.1.12 A Short Guide to the Simulation Architecture

In Simula, classes are used for a variety of purposes. They may be used as simple data structures, such as a packet, complex data structures, such as a queue, or processes. In our simulation, we must distinguish between Simula processes that represent real objects, and Simula processes that represent processes. There are four roughly distinct classes of objects in the simulation.

First, the IMP and hosts are Simula processes that represent real objects. They contain data structures and functions, but their code part is used only for initialization. They do not take an active part in the simulation. Part of their structure consists of pointers to other processes, which represent the processes running in the real IMP or host. The initialization code serves to create and initialize both the data structures and the processes. The data structures in the IMP are those for routing and buffer management.

```
queue class newQueue ;                  ! define newQueue as an extension
   begin                                  to queue ;
   ref (process) wakeMe ;               ! pointer to process waiting for
                                          a message to arrive ;

   ref (msg) procedure Remove;          ! procedure returns a pointer to a
                                          msg (i.e. message) ;
      begin
      if Empty then                     ! if queue is empty then process m
                                          wait for a message to arrive ;
         begin
         wakeMe :- current ;            ! set pointer to current process ;
         passivate ;                    ! go to sleep, wait for message ;
         end ;

      wakeMe :- none ;                  ! when woken, clear pointer ;

      Remove :- this queue.Remove ;     ! return message by calling
                                          ordinary queue operation ;
      end ;

   procedure Add (m) ; ref (msg) m ;    ! procedure to add message
                                          to queue ;
      begin
      this queue.Add(m) ;               ! add message to queue using
                                          ordinary queue operation ;

      if wakeMe =/= none                ! if pointer is set ... ;

         then activate wakeMe ;         !  ... then wake process which
                                          is waiting for a message to
                                          arrive ;
      end ;

   end ;
```

Figure A.1-1

Second, there are all the processes running in the IMP or host. They contain a pointer back to the IMP or host, data structures and functions, and code to simulate the operation of the corresponding real IMP or host process. The modem output process, for example, contains some of the data structures for

the line protocol, as well as the code necessary for outputting a packet to the modem.

Third, there are the objects which implement data structures, such as the packet and message queues. These contain some structure and a set of functions for operating on the structure.

Fourth, there are the packet and message objects, which are simply a set of fields with no structure. The only functions implemented by packet or message are simple functions such as clear.

A complete list of the objects and functions defined in the simulation is given in the implementation guide.

REFERENCES

[1] Ohlin, Mats. "A Working SIMULA Definition of SIMSET and SIMULATION," Swedish National Defense Research Institute, S-104 50, Stockholm 80, FOA rapport C10055-M3(E5), September 1976.

APPENDIX 2. SIMULATION IMPLEMENTATION GUIDE

Global Variables and Data Structures

parameters:

numCh	number of logical channels on a line
numBuffers	number of buffers per IMP
nImps	number of IMPs in simulation
nSimplex	number of simplex lines in simulation

variables:

nextOffset	next available entry in IMP's line delay ta
GlobalPacketCounter	counter for all data packets generated
seed	seed for random number generator
lex	input scanner
standardGremlin	default line error process
debug	file for debugging output
trace	file for trace output

data structures:

ImpMap	table of IMPs by number
OffsetTable	table of offsets into each IMP's line delay table, by IMP number

Class Packet

fields:

next	pointer to next packet on queue
number	global packet number
length	length of this packet in bits
error	flag to indicate line error
priority	packet's priority within each IMP
bufferType	type of buffer for this packet
sleeper	who is to wake up when packet is processed
nextEntryTime	time packet will arrive at next IMP
lineNumber	number of incoming line

routines:

Clear	clear all fields
Copy	copy all fields into another packet

Packet Class DataPkt

fields:

source	number of source IMP
dest	number of destination IMP
creationTime	time message was created by source hos
netEntryTime	time message entered source IMP
nodeEntryTime	time packet entered current IMP
nodeExitTime	time packet entered current IMP
channel	logical channel
channelBit	link protocol sequence bit
ackFlag	array of acknowledge bits for the line the opposite direction.
count	number of IMPs passed through
route	array of IMPs passed through

routines:

Clear	clear all fields and call packet Clear
Copy	copy all fields into another packet an call packet Copy
Register	add the given IMP to route and increment count
PrintRoute	print route on the trace file

Packet Class UpdatePkt

parameters:

nLines	number of lines whose delay is given i this update

fields:

impNumber	number of originating IMP
serialNumber	update protocol sequence number
age	time since update was created
retry	flag to indicate echo is requested
del	array of delays, one for each line

routines:

Clear	clear all fields, and call packet clea
Copy	copy all fields into another packet an call packet copy

Class Message

fields:

next	pointer to next message
source	number of source IMP and host
dest	number of destination IMP and host
length	length of message in bits
creationTime	time message was created

Class Wimp

parameters:

nImps	number of IMPs in simulation
nSimplex	number of simplex lines in simulation
number	number of this IMP
nLines	number of lines out of this IMP
numBuffers	number of buffers in this IMP

pointers to outside processes:

localHost	local host process
impCPU	scheduler process for this IMP
outLine3	array of line output processes

pointers to internal processes:

hostIn	host-to-IMP process
hostOut	IMP-to-host process
task	forwarding and routing process
modemIn	array of modem-to-IMP processes
modeOut	array of IMP-to-modem process
timeout	process for fast and slow timeouts

internal data structures:

traceFlag	tracing on or off for this IMP ?
debugFlag	debugging on or off for this IMP ?
clockRate	ratio of this IMP's clock to simulated time
clockOffset	time at which this IMP started
h	heap used in SPF routing calculation
LineDelay	delay for every line in simulation
threshold	current delay threshold
initialThreshold	initial delay threshold
thresholdDecay	amount threshold is reduced each time

UpdateAge	age of routing update from each IMP
UpdateSN	serial number of latest routing update from each IMP
UpdateTimer	timer for updates from each IMP on each line
fixedRoutingFlag	if set, do not generate routing updates
randomRoutingFlag	if set, forward packets at random
OutLine	routing table
freeBufferCount	count of total number of free buffers
BufferCount	count of buffers in each buffer class
BufferMax	maximum allowable number of buffers in each buffer class

routines:

PrintRouting	print routing table on debug file
PrintDelay	print line delays on debug file
TimerTick	decrement UpdateTimer and send routing update if necessary
AgeTick	age routing updates
SendUpdate	construct update packet from update tables and send it out given line
AverageDelay	reduce threshold by threshold decay; average delay on each line; if for any line the change in delay is greater than the threshold value, then reset the threshold, set the delay in each line to the average delay just computed, then send out a routing update for this IMP over all lines.
PrintUpdate	print update on debug file
ProcessUpdate	accept or discard update packet; echo if appropriate
Acceptable	check serial number of incoming update
UpdateRouting	recompute shortest path tree and update routing table
FreeBuffer	decrement appropriate BufferCount
AllocBuffer	create buffer
AllocateReassembly	check reassembly count (BufferCount [1])
BecomeReassembly	if possible, reassign buffer to type 1
BecomeStoreAndForward	if possible, reassign buffer to type 2
AllocateUncounted	check uncounted count (BufferCount [3])
DebugPrint	print packet on debug file
StartUp	activate host and IMP processes
InitializeBufferLimits	initialize BufferMax array

code:

Initialize and activate local processes and local host, initialize heap and buffer counts, put entry for local lines into global offsetTable, then quit.

Class TaskProcess

parameters:

priority	impCPU scheduling priority
serviceTime	(constant) service time per packet

pointers to other processes:

myImp	IMP which owns this process

internal data structures:

inQ	input queue for data packets
updateQ	input queue for routing update packets
asleep	flag indicating this process is idle

routines:

Add	add a packet to the correct queue and wake the process if it is asleep.

code:

If there is an update to be processed, call the IMP routine ProcessUpdate.

Otherwise, if there is a data packet, perform duplicate suppression, buffer allocation, and forwarding to host or modem output routines. If the packet was accepted and came from a modem, flip the receive channel bit and send an acknowledgment (wake up modemOut if it is asleep); if it was a duplicate, send a duplicate acknowledgment (wake up modemOut if it is asleep).

Class ModemInProcess

parameters:

priority	impCPU scheduling priority
serviceTime	(constant) service time per packet
lineNumber	number of line for this input process

pointers to other processes:

myImp	IMP which owns this process
myOut	IMP-to-modem routine for the other half of this duplex line.

internal data structures:

inQ	queue of packets to be input, contains a most one packet.
notReady	count of packets in inQ, at most one.
ackFlag	array of receive flags, one per logical

routines:

modemInterface	accept or reject packet from line
ackPacket	flip receive flag on specified channel

code:

For each arriving packet, add update packets to task update queue, and data packets to task input queue, unless packet is in error or buffer for next packet is unavailable. For data packets, process acknowledgment bits and free any acknowledged packets.

Class ModemOutProcess

parameters:

priority	impCPU scheduling priority
serviceTime	(constant) processing time per packet
lineNumber	number of line for this output process
retransmitWait	retransmission interval

pointers to other processes:

Imp	IMP which owns this process
myLine	line input process
myModemIn	modem-to-IMP routing for the other half of this duplex line.

internal data structures:

inQ	queue of packets waiting for output
nullPacket	null packet for sending acknowledgments
unAckedPacket	array of packets waiting for acknowledgment
packetCount	total number of packets waiting
ackToGo	signals that an acknowledgment should be se
ackFlag	array of send flags, one per logical channel
asleep	flag indicating process is idle
freeCurrent	flag set if acknowledgment received for packet being output.
del	delay out the line
totalDel	accumulator for delay averaging
totalPackets	accumulator for delay averaging
avgDel	result of last delay averaging
otherImpNumber	number of neighboring IMP

routines:

TallyDel	accumulate delay for this packet
AverageDelay	compute average delay, clear accumulators, and report difference between old delay and this delay.
GrabChannel	allocate logical channel
Free	tally delay of acknowledged packet, flip ackFlag, clear UnackedPacked entry, decrement packetCount and free buffer.
Ack	set ackToGo and wake up process if asleep
Transmit	add packet to input queue, increment packet count and wake up process if asleep

code:

ModemOut is woken up by a call to Transmit, or by a periodic wakeup by fastTimeout, or by a call to Ack to send an acknowledgment. Transmit is either called by Task, with a data packet, or from SendUpdate (a routine in Wimp), with an update packet.

When ModemOut is woken, it checks to see if it should transmit a packet. If there are any packets to be retransmitted, it retransmits one of them; if there are any packets on its input queue, it removes the first one and transmits it; if the ackToGo flag is set, indicating that a packet has been received but not acknowledged, a null packet is sent and the flag cleared; if there is nothing to do, ModemOut goes to sleep.

If ModemOut has to send a packet, it passes it to the line input process for the line, then goes to sleep. When it is woken by the simulated completion interrupt, it frees the buffer if it just sent an update packet, then checks to see if another packet should be sent.

Class HostInProcess

parameters:

priority	impCPU scheduling priority
serviceTime	(constant) processing time per packet

pointers to other processes:

myImp	IMP which owns this process

internal data structures:

inQ	queue of packets to be input

routines:

HostInterface	read in a message from the host; create a packet for it and put it on the input queue

code:

Remove a packet from the input queue; execute serviceTime units of simulated time and pass the packet to Task.

Class HostOutProcess

parameters:

priority	impCPU scheduling priority
serviceTime	(constant) service time per packet

pointers to other processes:

localHost	host for this IMP
myImp	IMP which owns this process

internal data structures:

inQ	queue of packets to be input

code:

Take packet from input queue, put it on the input queue of the host's packetSink process, then free the buffer.

Class TimeoutProcess

parameters:

priority	impCPU scheduling priority
nLines	number of duplex lines in this IMP
period	time between successive executions

pointers to other processes:

myImp	IMP which owns this process

internal data structures:

tick	counter to count 25 fast ticks
counter12	counter to count 12 slow ticks
counter15	counter to count 15 slow ticks
currLine	line to be woken during this execution

routines:

DebugPrint	print event on debug file
FastTimeout	if current line is asleep, wake it up
SlowTimeout	increment counter12; call IMP routine RoutingTick to age routing timers and

clear counter if it is 12; increment counter15; call IMP routine AverageDelay to compute average delay on each line ar clear counter15 if it is 15.

code:

Go to sleep until next tick (occurs every period seconds). Wake up and increment tick counter. If counter is 25, call SlowTime out and clear counter. In either case call FastTimeout.

Class Line

parameters:

sender	sending IMP number
sIndex	sending IMP's index for line
receiver	receiving IMP number
rIndex	receiving IMP's index for line

pointers to other processes:

lin	input process for this line
lon	output process for this line

internal data structures:

traceFlag	tracing on or off for this line?
debugFlag	debugging on or off for this line?

routines:

DebugPrint	if debugging is on for this line, and t specified debugging output is enabled, print the specified packet and message.

code:

Create input and output processes for this line. Connect together the sending IMP's modem output process, the input process, the output process, and the receiving IMP's modem input process. Start up the input and output processes.

Process Class LineInput

parameters:

lag	speed of light delay over line
bitTime	time to clock one bit onto line

pointers to other processes:

sender	modem output process on sending end
otherEnd	corresponding lineOutput process which transfers packets to receiving IMP

internal data structures:

inQ	queue of packets being accepted, contains at most one packet

code:

Remove packet from input queue, compute packet's arrival time at other end of line, add packet to transit queue at other end of line, then wake up sender (completion interrupt).

Class LineOut

parameters:

bitTime	time to clock one bit onto line
errorRate	bit error rate

pointers to other processes:

receiver	modem input process in receiving IMP
lineError	process which decides whether packet is in error.

internal data structures:

transitQ	queue of packets in flight

code:

Remove first packet from transit queue, then wait until its arrival time. Call error procedure in the lineError process to decide whether this packet should get a line error,

then pass the packet to the modem input process by calling the modem Interface routine.

Class Host

parameters:

nImps	number of IMPs
number	number of this IMP, and hence this host
ImpProcess	IMP's host input process

internal data structures:

traceFlag	tracing on or off for this host ?
debugFlag	debugging on or off for this host ?

pointers to other processes:

Source	array of message generating processes
Sink	packet accepting process
messageOut	host-IMP interface process

routines

Start	start up a new message generating proce
Close	stop a message generating process
PrintArrival	print packet on trace and/or debug file
PrintMessageOut	print message on debug file

Class messageGen

parameters:

myHost	pointer to host
dest	destination IMP and host for all messag from this generator
rate	mean message generation rate
avgLength	mean message length

code:

Generate messages with negative exponential inter-arrival times and negative exponential lengths; put each message on messageOut's queue for transmission to the IMP.

Class MessageOut

parameters:

myHost	pointer to host

internal data structures:

inQ	queue of messages

code:

For each message put on the queue, call the IMP routine HostInterface, in the IMP's host input process.

Class PacketSink

parameters:

myHost	pointer to host
serviceTime	(constant) time per packet

internal data structures:

inQ	input queue of buffers

code:

Remove packet from input queue, call TraceArrival to log packet, then give packet back to the Simula runtime system.