IEN 182

Issues in Buffer Management

Bolt Beranek and Newman Inc.

Eric C. Rosen

May 1980

Table of Contents

1 Introduction

This note is an abridged extract from BBN Report No. 4473, "ARPANET Routing Algorithm Improvements, Volume 1", by Rosen et al. It discusses the issues of buffer management in the switches which implement a network and is based on experiences gained during the evolution of the ARPANET.

Since the Internet is itself a network, and hosts or gateways implementing TCP, IP, and other protocols have similar buffer management design decisions, this IEN is intended to distill some of the ARPANET issues and present them to a wider audience currently grappling with some of the same problems.

The original report is quite large (500 pages). This is the first of several such extracts we plan to produce to serve as background for the internet project work. The report was first published in August 1980.

Some of the terminology used may cause confusion if associated with internet work, for example "reassembly". This note discusses mechanisms purely internal to the ARPANET, which itself has many similarities to internet and TCP mechanisms in internet hosts. The ARPANET IMPs use retransmission, ACKS, flow control/windowing, fragmentation and reassembly, out-of-order sequencing, and other mechanisms which create a serial bytestream service based on a datagram network, much as TCP does.

The issues to be discussed in these notes are at least partially applicable to the internet mechanisms, including TCP in hosts, as well as IP in gateways, since those mechanisms are functionally similar in the services they are intended to implement. We propose no solutions here, such as buffer mechanisms for TCP implementations, but rather intend to explore the issues which motivated the IMP implementation in the ARPANET, to help TCP and internet implementors in their similar tasks of creating an Internet.

Anyone interested in seeing how the issues raised in this discussion can be applied to the ARPANET will want to see Chapter 7 of BBN Report No. 4088, as well as Chapter 1.5 of BBN Report No. 4473, which are not included in this excerpt. Copies of those reports are available from the author.

2 Overview

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.

3 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.

4 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.

4.1 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).

4.2 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 rules 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.

4.3 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.

4.4 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 brg the question of fairness, and in doing so lead to congestion control or buffer management schemes which are seriously deficient.

5 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 message 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 PARTICULAR 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.