

IMPLEMENTATION OF THE MIMICS PACKET SWITCH

by

JAMES R. RATLIFF

B.S., Kansas State University, 1974

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1980

Approved by:


Major Professor

SPEC
COLL
LD
2668
R4
1980
R38
C.2

TABLE OF CONTENTS

1. INTRODUCTION

1.1	Structure of This Paper.	2
1.2	Justification of Concurrent Pascal as Design/Implementation Language	4
1.3	Implementation Environment	8
1.4	Advantages of Heirarchical Structure	12

2. PACKET BUFFER MONITOR

2.1	Functional Specification	19
2.2	Data Structures.	26
2.3	Entry Point Descriptions	35

3. ROUTE TABLE MONITOR

3.1	Functional Specification	46
3.2	Data Structures.	48
3.3	Route Table Monitor Entry Points	50

4. WINDOW MONITOR

4.1	Introduction	54
4.2	Functional Specification	59
4.3	Data Structures.	60
4.4	Window Monitor Entry Points.	63

5. HALF-DUPLEX LINE CONTROL PROCESS

5.1	Background.	67
5.2	Functional Specification.	69
5.3	Extension to Other Protocols.	72
5.4	Data Structures	73
5.5	Protocol Implementation	75

6. CONCLUSION

6.1	Past Experience.	77
6.2	Appraisal of Concurrent Pascal	79
6.3	Enhancements and Improvements.	84
6.4	Conclusions.	88

REFERENCES.	90
---------------------	----

LIST OF FIGURES

1.1 MIMICS Protocol and Interface Mechanisms.	3
1.2 MIMICS Implementation Structure	10
1.3 Off-Loading of Packet Switch.	16
2.1 Packet Format	28
2.2 OUTBOUND Control Structure.	30
2.3 INBOUND Control Structure	33
2.4 Logical Packet Flow	36
4.1 Window with Unidirectional Data Flow.	56,57,58
5.1 Packet Format	71
5.2 Line Process Format	74

ACKNOWLEDGEMENTS

I wish to thank all those associated with me for the past many years for their tolerance in awaiting the completion of this historical report on the implementation of the MIMICS system. In particular, thanks to my wife and Dr. Wallentine for their understanding and patience. With all due propriety, I would also like to express my formal thanks for the routine help given me by my adviser and committee.

The MIMICS (Mini-Micro Computer System) is a general purpose network system developed at Kansas State University. The intent is to utilize progressively more sophisticated micro and mini computers to implement those functions necessary (or desired) to allow the distribution of processor load over multiple machines, and specifically to allow these multiple processors to share a common base of data. (An initial example now under development is the off-loading of a data-base management function from a mainframe onto a backend minicomputer, utilizing the MIMICS system as the distribution interface[14]).

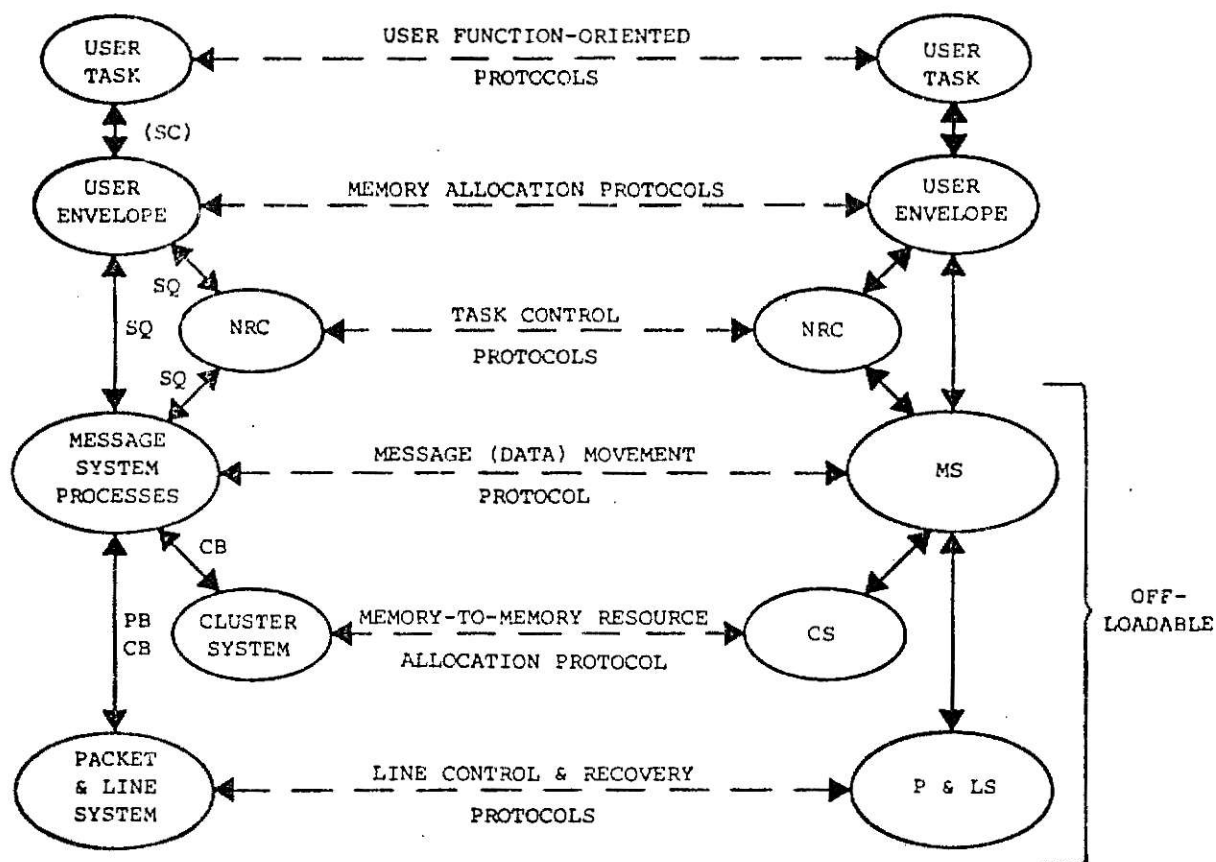
MIMICS is divided into three functionally distinct elements. The Message System (MS) accepts requests directly from the user via the local operating system. The Message System performs user -to -user coordination and flow control at the message level. The Packet System (PS) accepts messages which have been disassembled into 128 byte packets and insures their correct and timely flow to some remote machine, and the Cluster System (CS) manages high-speed transfers between tightly connected machines. The Cluster System performs high-speed transfers directly between memory spaces of the two requesting users, whereas the Packet System must utilize storage of packets in MIMICS memory while messages are disassembled and then reassembled at the destination.

The structure of the MIMICS system is such that it can

support very high-speed data transfers (on the order of 10 megabytes per second) between machines in close proximity to each other (referred to as a cluster) as well as lower speed (110 to 50000 baud) transfers over long distances using commercially available hardware. This is accomplished in a manner which is entirely transparent to the user except for the difference in time needed to get a response. For either type of transmission, enforcement of message level protocol and flow control is the responsibility of the Message System, the main component of the MIMICS System. After the Message System has confirmed the validity of a user's request and a connection to be used in fulfilling that request has been established, the respective Cluster System or Packet System is invoked to implement the actual data transfer. Figure 1.1 shows a graphic representation of the levels of flow control and coordination which take place between two Message Systems. (Note that the use of the term 'Message System' is used interchangeably in reference to the MIMICS system and to the MS part of the system.)

1.1 STRUCTURE OF THIS PAPER

Although this paper contains only the Packet Switch portion of the MIMICS System, the reader should be aware that many of the concepts (especially the choice of implementation language and the hierarchical nature of the



Legend:

SQ = SYSQ in Local Operating System (Assembler Code)

SC = Subroutine Call

NRC = Network Resource Control

PB = Packet Buffers

Figure 1.1

MIMICS Protocol and Interface Mechanisms

software) are common to the entirety of the software system.

The remainder of this chapter is devoted to justification of the choice of Concurrent Pascal as a combination design/implementation language, a description of the environment of this implementation, and an explanation of the design criteria achievable through the use of hierarchical components to implement the system.

After the introductory concepts are explained, the three components of the Packet Switch itself are presented. Chapter 2 contains a description of the Packet Buffer Monitor, Chapter 3 consists of a discussion of the Route Table, and Chapter 4 contains a discussion of the function and operation of the Logical Line Window Monitor. Chapter 5 contains a the line process used in testing between the Interdata 8/32 and 7/32 and an IBM 370-158. IBM 370-158.

1.2 JUSTIFICATION OF CONCURRENT PASCAL AS DESIGN/IMPLEMENTATION LANGUAGE

Although the title indicates that Concurrent Pascal (CPascal) is a design and implementation language, this section consists of a justification of CPascal as an implementation language. Once this is done, the perceptive reader should realize the advantages of having a single high-level language for both design and implementation.

The advantages of programming in a "high level"

language and the value of maintaining a single version of a software system (as opposed to the common dual version, one for design and another for efficient implementation) have been discussed so frequently that no reader of computer literature need be convinced in this paper of the advantages of having a single language for both design and implementation [1,3].

The point of contention has been the means for achieving efficiency from a robust high level language. The practice in the past has been to make one of two choices. The first choice was to take the high level design and hand translate this design into the assembly language of the target machine. This approach works quite well assuming the availability of a super programmer who can generate effective assembly language quickly; but loss of that one person leads to serious maintenance problems. Additionally, distribution of revisions and "patches" is complicated if hand translation is used. No argument can be made, however, about the relative efficiency of code generated by this method.

The second choice has been to implement the translation with the computer through the use of a macro language as the implementation (and possible design) language. Portability to a new machine then involves a single generation of the assembler code to implement the macro's of the language. Revisions and other maintenance are now based on the macro

language, and translation is performed by the machine. It is assumed that the macros themselves are quite static. This approach offers a good intermediate ground between efficiency of generated code and good software engineering practices.

Some problems do present themselves, however. Most important is the choice of a macro language. It would be desirable to select a language for which many portings had already been accomplished. Additionally, the nature of our network system is much like that of an operating system in that much of the activity is asynchronous, as opposed to a normal program which is largely sequential. A good macro language for our purposes would have to provide a great deal of support in synchronizing these asynchronous activities to insure relatively correct execution of code and to help prevent the occurrence of time dependent errors, since they are extremely hard to eliminate after the fact. No macro language was found which we felt adequately fulfilled these requirements; and so we must turn our search in a different direction.

In contrast to the other methods, the language Concurrent Pascal was designed expressly to meet the goals we have enumerated. The concept of monitors (and classes) allows a virtual certainty that shared data structures are shared properly among concurrently active processes. It can be guaranteed that, if a shared data structure (a monitor)

is corrupted, the fault is the result of an improperly written entry point and the error is of a sequential nature. The fault is not the combination of some complex sequence of time dependent events (usually involving disabling and enabling of interrupts) which the programmer had not anticipated, and which probably cannot be duplicated. Detection and correction of these time dependant errors can be a significant expenditure of resources in a large software system such as this.

Concurrent Pascal also contains a more than adequate complement of control structures, including "case" statements, "repeat-until" statements, "while" loops, and the common "if-then" sequence to allow the utilization of software engineering practices in the use of CPascal as a design language.

The final criteria is portability. The acceptance of Pascal as a programming language is increasing as evidenced by the number of Pascal compilers available. CPascal was initially brought upon a PDP 11/45 [2], and this system was then ported to an Interdata 8/32 in 4 (person) months [20]. Additionally, many other implementations of PASCAL systems already exist on other machines, thus enhancing portability of our system [1]. Thus, we see that the language is at least as easily ported as a macro language, and the general acceptance of the language may obviate the need for a porting effort.

It is the system structure of CPascal which allows for easy synchronization of asynchronous events, and the maintainability of a single language for designing and implementation which led to the choice. The compilers used in this implementation generate an instruction set designed to run on a virtual stack machine. This instruction set is then interpreted at execution time. Because of this, the execution times of programs executed on our system will not be as efficient as would be the case if the language of the host machine were generated. However, generation of code for a virtual machine allows the transportation of programs among machines without even the necessity of recompilation. Also, since the cost of processor capability continues to fall relative to the cost of porting and maintaining a large software system, processor costs become increasingly unimportant when measured against the costs of maintaining a large software system.

1.3 IMPLEMENTATION ENVIRONMENT

In this section we discuss the environment for this implementation of the Packet System (and indeed for the MIMICS system). As mentioned earlier, not many machines currently exist which use Pascal-like machine code. Therefore, most implementations consist of modifications to the compilers to generate assembly language from the Pascal

code, or of implementations of a Pascal interpreter to execute Pascal virtual instructions. The latter is the environment in which Pascal executes at Kansas State University.

David Neal [20] has ported the Pascal system developed by Per Brinch Hansen at Cal Tech on a PDP 11/45 onto an Interdata 8/32 system, running as a task under OS32MT2 [7]. The initial implementation at Cal Tech was a stand-alone system in which the kernel controlled the bare machine, and acted as the interface between the Pascal system and the real machine. In the KSU implementation, as shown in Figure 1.2, the kernel has been rewritten to act as the interface between the Pascal system and the 'virtual' machine (ie. the Operating System). In addition, the assembly language interpreter which implemented the virtual code from the compiler on the PDP system was hand translated into Interdata assembly language, and the entire system was operational in a matter of three months effort.

Additionally, since the purpose of the MIMICS system development was not to develop software at the synchronous line-driver level, it seemed appropriate to utilize the existing bisynchronous capabilities provided by Interdata. This choice was enhanced by the capability provided in OS32MT to allow the user two levels of access to the Interdata Telecommunications Access Method (ITAM) [8]. While the SVC 1 assumes the user desires to utilize IBM's

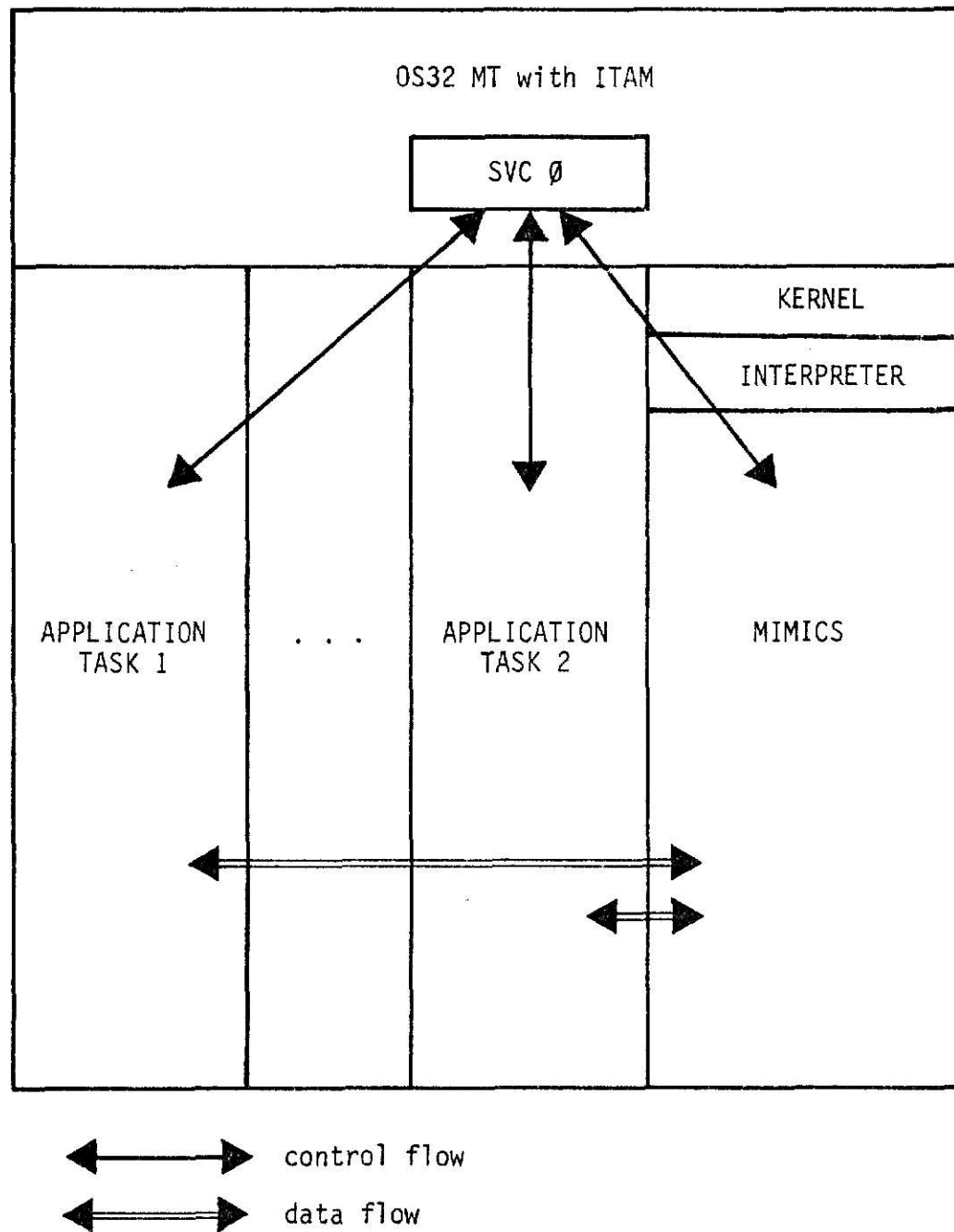


Figure 1.2
MIMICS Implementation Structure

Bisync communications protocol [17], the SVC 15 level allows the user to develop his own protocol, but utilize the common IBM synchronous communications format.

The SVC 15 level provided the capability to utilize a commonly available packet format, and yet develop a protocol which required less overhead than the standard Bisync. Standard Bisync provides capabilities for allowing multiple users to request the line via the sending of 'ENQ' packets, but requires considerable overhead in acknowledging the 'ENQ' packets. Since it was envisioned that a connection was to be point-to-point, we felt the overhead for sending and acknowledging ENQ's could be improved upon. Figures 5.1 and 5.2 are examples of a single communication utilizing standard Bisync, and a single communication utilizing our protocol.

Use of ITAM necessitated the addition of routines in the kernel to implement the interface between CPascal programs and the OS. This proved to be a fairly minor task. However, implementation and testing was not a minor task due to several factors, foremost of which was Interdata's lack of testing the QSA (Quad Synchronous Adapter). In developing ITAM they had access to a 7/32 utilizing a data set adaptor which is compatible with the Bell 201. Our environment consisted of an 8/32 and the aforementioned QSA. For example when testing in local loopback, the ITAM driver hung up in an infinite loop expecting a status which the 201

Data Set Adaptor would return, but which the QSA would never return. The solution to the problem entailed changing the instruction to always ignore the status.

The other 'major' problem was the communications environment. The data access arrangement (daa), which ties the telephone to the modem, had been installed for a Timeplex 1200 baud asynchronous modem. In an effort to save money, an attempt was made to attempt to make this work with the Rixon 2400 baud synchronous modem. The combination finally worked (off and on), but was restricted to being able to receive calls, not to originate them.

1.4 ADVANTAGES OF HIERARCHICAL STRUCTURE

In "The Architecture of Concurrent Programs", Per Brinch Hansen discusses some of the advantages of programming in CPascal as opposed to other more orthodox, block-structured languages. He likens building a program to building a pyramid of bricks, where we worry about the correct fit of each brick as it is put in place; but once in place we need not concern ourselves with that particular fit, since other bricks can only lie on top (constituting a well-defined interface) of the previous bricks and cannot disturb the fit of previous bricks. In CPascal the modules do not physically lie upon one another, but their interactions are limited (and compiler checkable) by the

explicit access rights just as the laws of physics limit a top brick from pushing in any direction other than down on a lower brick. This hierarchical ordering of system components has vital consequences for system design and testing.

A hierarchical concurrent program can be "tested" component by component "bottom up" irrespective of the design methodology (top down or iterative). And once this lower brick has been shown to work correctly (by proof, testing, or intuition), the compiler can guarantee that this component will continue to work correctly when new components are added by insuring that no new component can call this one in any way other than by the specified access rights. Notice that this bottom up testing starts with modules which rely on no other modules, and graduates up to modules which rely only on those modules already proven correct. Compare this to programming in assembly language, where the addition of any new module has the capability to destroy memory locations anywhere in some address space (probably the entire machine), including modules already assumed correct. Compare it also with block-structured languages in which the scope rules allow internal modules to have access to external variables. The common situation is program modification in which either an internal module relies on an external variable through poor design (or laziness in establishing calling formats), or changes in

declaration of internal variables allows incorrect access to an external variable, a condition which the compiler cannot catch but which would be caught in CPascal.

So much for the nice generalities of the desirable features of a hierarchical language such as Concurrent Pascal. The next sections contain a discussion of how these features affected the design goals of the MIMICS system. One of the primary goals of MIMICS was to allow the easy growth of total processor power through the addition of mini and micro computers in successively larger sizes until the total processor capability might well be greater than that of the original host. It was desired that this growing experience be as painless as possible, and allow the user as much flexibility in his choice of processor as possible. Thus the emphasis on portability, and on the hierarchical design. It was hoped that a user be afforded multiple combinations of host processor and MIMICS processor, some of which are listed below:

1. bringing up the MIMICS system in the host machine in its entirety, as is currently the case on our Interdata machine, either in an interpreted version or as compiled code in the machine language of the host machine;
2. estimating some percentage of free processor on the host machine, and offloading that portion of the MIMICS system which the host cannot handle. An obvious example would be the offloading of the packet switch portion onto a 'communications controller' to control the traffic on the synchronous lines, thus creating an environment much like the typical front end processor[24];
3. offloading all of the MIMICS functions onto the communications controller. Thus the only overhead on the host processor is the means of

- communication between the host and the CC;
- 4. acquisition of a communications controller with additional processor capability (a large mini), and using this machine as a communications controller for both machines as well as a second host;
- 5. modification of (4), in which both hosts have the Message System and Cluster System portions of MIMICS, but only one has the Packet System;
- 6. OTHER.

We will now attempt to walk through the example proposed in (2), in which we use the packet switch portion of MIMICS as a front-end processor for the synchronous line traffic.

What we wish to do is move all of the MIMICS structure from the Packet Buffer Monitor down through the physical line processes onto the newly acquired communications controller (see Fig. 1.3), leaving the Message and Cluster portions of the system running in the host. The hierarchical design capabilities of CPascal simplify this operation. The primary feature supported by CPascal is the delineation of those access rights by which any caller can access a given module; in this case the module in question is the Packet Buffer Monitor. If all of the capabilities and access rights provided by the Packet Buffer Monitor are present in both the host part of the Message system and in the communications controller part of the Message system, then no functional change has been made. All that need be done is to put the data structures and the code in the communications controller, and to insert a dummy Packet Buffer Monitor in the host machine with the same entry

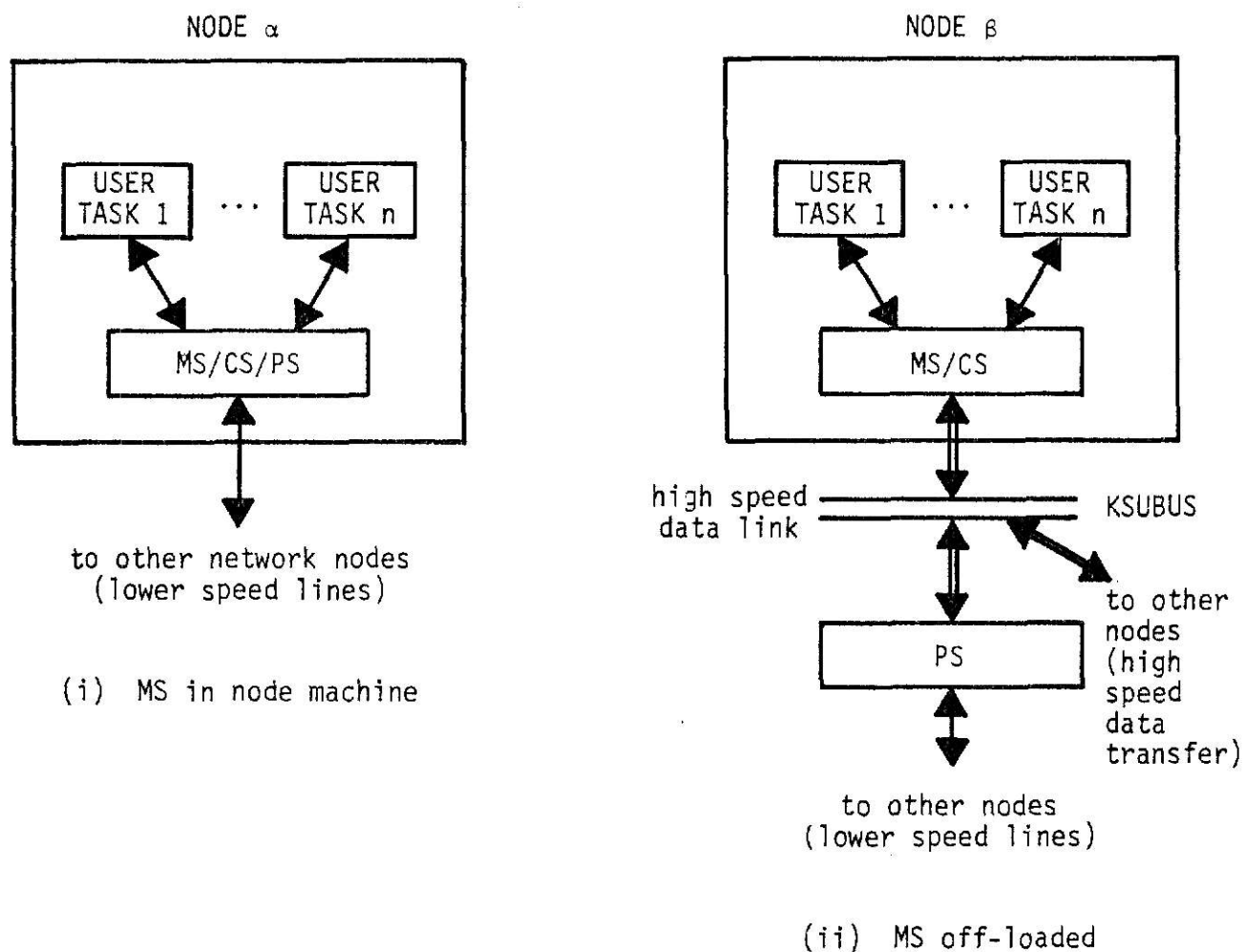


Figure 1.3
Off Loading of Packet Switch

points and access rights as the actual Packet Buffer Monitor. All this dummy monitor has to do is communicate with the communications controller, indicating which entry point was called, and what data was passed with that call. In the communications controller, communications with the host is accomplished via the addition of a process which controls the line to the host, decodes the transmission into entry point called and data for that call, and then calls the given entry point in the 'real' Packet Buffer Monitor. Since no functional changes have taken place to any of the bricks which made up the original pyramid, the possible side effects have been severely limited. Additionally, since it is undesirable to have a monitor controlling some real device directly (due to loss of concurrency and increasing likelihood of locking up the monitor itself) we insert another process between the dummy Packet Buffer Monitor and the path of communications to the communications controller. The configurations are shown in Figure 1.3. The important point to note is that the system is dependent upon showing that the processes communicating between the two machines never deadlock, and that no data is lost in transmission since no change was made to the functional bricks of the system.

Another example of the hierarchical modularity and its advantages involves development of a MIMICS system without any packet capabilities. This is a simple change. Since

portions of the Message System processes and Cluster processes assume that they have access to the Packet Buffer Monitor, we cannot merely delete those modules which make up the packet switch. That would involve going into the code of the entire system and dummifying out all calls to the Packet Buffer Monitor, and praying that this has no side effects that are not readily apparent. The easiest (but possibly not best) thing to do is to build a new Packet Buffer Monitor with the same access rights and entry points as the real thing, but with code changed to a BEGIN-END block with a single statement returning some error code. Hence, the access graph of the system is unchanged, and all we have done is supply a dummy Packet Buffer Monitor and delete all of the routines below the packet Buffer Monitor.

Up to now, we have discussed the hierarchy components as system types (Monitors, Classes, and Processes), but the reader should be aware that the same concepts are true within each of the system-type components. If a monitor manipulates a given data structure, then there should be some minimum set of operations which will be performed upon those structures. The code to perform these operations should be written as the basic bricks of the monitor, and the entry points of the monitor need only call upon these blocks in the desired order (ie. Get a free packet, Fill the packet, Link the packet to a logical line list).

2.1 FUNCTIONAL SPECIFICATION

The Packet Buffer Monitor consists of a common pool of data buffers, control structures for packets passing from the Message System to the line processes, control structures for packets passing from the line processes to the Message System, and routines to copy data and manipulate the control structures. It is also the function of the Packet Buffer Monitor to try to insure that packets are fairly allocated (no maverick process can tie up enough buffers to cause deterioration of service), and that packets going out across a telecommunications line are scheduled fairly according to their priority and length of time in the system.

Although it would appear that the Packet Buffer Monitor performs a similar function for both Inbound and Outbound packets, this is not actually the case. Both utilize a common pool from which they request buffers for the storage of packets, and both act as a buffer between a source and a sink, but the types of services required to provide robust buffering capabilities cause the actual structural differences to be significant. Thus we have a full-duplex buffering service with significant difference in the structure of each simplex half of the system.

In the following discussion, we will attempt to group items according to their functional half. We are, essentially, discussing a simplex Outbound Packet Buffer Monitor and a simplex Inbound Packet Buffer Monitor which

are combined to reduce the size requirements for buffers for each, since they can now share the common free pool.

OUTBOUND FUNCTION

Outbound packets (those originating in this host machine and destined for some other host whose only connection is across telecommunications lines) are of three types. DATA and COMMAND packets are originated by the message disassembly process(es) in the communications controller. FORWARD packets were originated in some remote machine, but have been routed through this communications controller because no direct path existed to the desired destination machine (or, in more sophisticated routing schemes, in an effort to increase the bandwidth of the logical communications path). FORWARD packets will, of course, be of either DATA or COMMAND type.

Several functions are provided by the Packet Buffer Monitor on behalf of the logical line processes, and each function typically results in a change in status of the affected buffer. In order to insure that no logical line is capable of requesting and receiving all of the buffer pool buffers, a level of flow control is provided based on the semi-static preallocation of a maximum number of buffers that a logical line is allowed to control at any time. Additionally, since transmission of a packet does not imply correct reception of the packet, the Packet Buffer Monitor

provides a holding function for packets until they have been correctly received. The line can delay the packet for some period of time (referred to as a Sack interval) and then later pass the packet to the line process for another attempt at transmission. Once the packet is acknowledged as correctly received at the destination, the Packet Buffer Monitor is told to release the buffer containing that packet, and the buffer is returned to the free pool.

The third major function of the Packet Buffer Monitor is the scheduling of packets based on their time-specific priority. Somewhat arbitrarily, the highest priority is given to Forwarding packets, to be discussed in depth after Command and Data packets. It is interesting to note that our three groups of packet types fit quite well with the network traffic in [3], that is low-delay, high-throughput, and real-time.

The next type of packet based on priority is the Command packet. Due to the effect which command packets can have on the overall traffic on the communications line, it is expedient to transmit them as quickly as possible, possibly saving considerable transmission of useless packets. Since it is desirable that these packets flow with as little delay as possible, we refer to them as low-delay traffic. Since these packets will flow quite quickly in a normal system, it is not necessary (and in fact not desirable) to allow a large number to be queued for

transmission. Therefore, the maximum allocation constant for the Command queue should be set quite small.

By contrast, data packets are said to be of a high throughput nature. The importance of a single packet is negligible (unless it happens to be the only one needed to complete a message); what is important is full utilization of the bandwidth of the logical communications path to allow the transmission of the entire complement of packets as rapidly as possible. Although data packets may be delayed by higher priority packets, it is important that enough packets be queued up awaiting transmission that no slot in the sequence of available packet slots is empty due to lack of buffer space. The Data queue should be allowed at least twice as many buffers as the Command queue, and perhaps a few more.

Now we return to the discussion of the Forwarding packets. Although it is obvious that Forward packets are of both data and command types, no effective scheduling would seem possible due to the possible wide disparity in the number of forwarding stations already encountered, and therefore the delay encountered by those packets. Somewhat arbitrarily, therefore, we have decided that forwarding packets are of a real-time [3] nature; that is, to insure data transmission and response rates throughout the network that are considered "acceptable", forwarding packets must have the highest priority. No distinction is made between

data and command packets, since this scheduling choice has already been made at the source Packet Buffer Monitor.

Some arguments might be made for saying that local command packets should be more important than forwarding data packets, but the need for some real-time response to a message combined with the unknown delay already encountered by any forwarded packet leads us to believe that we must forward packets as quickly as possible. However, to insure that no class of packet locks out the other classes, the following transmission ratios are maintained assuming that all queues have a ready packet in them when it is requested.

7 packets sent:	Forward queue => 4 packets.
	Command queue => 2 packets.
	Data queue => 1 packet.

Therefore, the local machine can make some headway, even in the presence of a heavy store-and-forward load. Very few store-and-forward buffers should be allocated, since they should flow quickly. We also wish to have a minimum number of packets outstanding in the Forward queues in the case that it becomes necessary to allow a Sack to ripple back through the route to the source machine.

INBOUND FUNCTION

In comparison with the Outbound control functions provided, the Inbound functions provided are quite simple. Incoming packets can be of only two types (COMMAND and DATA), and they are placed in one of four queues.

If some error has occurred, then the packet is placed on a queue (SPOOL0). The Spool0 queue is currently emptied merely to prevent the clogging of the Packet Buffer Monitor by garbage packets, but a later implementation might include a legitimate attempt to recover from the error, or to log some message as to the nature or frequency of the error. Some common errors might include mis-routing (the packet finally arrives at a machine which does not have a route to the destination); implying route table errors in the network. A more common error would be the reception of a packet before the proper synchronization has been done to allow the correct reception of the packet. In this case, a true spooler could be implemented which saved the packet and returned it to the Packet Buffer Monitor for later reception by the Message System.

The second receptacle (buffer) for incoming packets is labeled CLUSTER. A likely network configuration includes several host processors (mainframes or large minis) with multiple communications controllers (possibly one per host). In this configuration, it is likely that only one (certainly less than all) of the communications controllers will have

telecommunications capabilities. Those CC's with access to remote machines are now also responsible for forwarding data (consisting of Command and Data packets) to other machines within the cluster. The Cluster queue is used to facilitate this de-multiplexing of packets intended for a machine in this cluster but not actually controlled by this Communications Controller.

The other two queues, COMMAND and DATA, are used to retrieve command packets and data packets destined for hosts controlled by this Communications Controller.

As with the logical line processes, receipt of a packet from the Packet Buffer Monitor does not imply that the retrieving process is actually capable of processing that packet (primarily true of the Command queue). Therefore, the process can specify at retrieval time whether the Packet Buffer Monitor may release the buffer immediately (always true for data packets), or whether specific instructions should be awaited as to the action to be taken. The packet can later be either released, or delayed and scheduled for presentation to the retrieving process at a later time. This delay process is similar to the Sacking of a packet in the logical line.

AUDITING FUNCTION

The third function of the Packet Buffer Monitor is the auditing of it's data structures. We have discussed earlier

that Outbound packets may be "Sacked" and that Inbound packets may be "Delayed". The former case indicates that the destination machine was overloaded or otherwise unable to handle the packet at the time it was received, and the second case indicates that the retrieving process was unable to process the packet at the time of reception but expects to be able to process it (the packet) at some later time.

It is the function of the Audit entry point to check all data structures for packets that are either Sacked or Delayed, to calculate the length of time they have been in that state, and to prepare them for re-entry into their respective queue. Thus a Sacked packet is unsacked and prepared for transmission across the tele-communications line, and a Delayed packet is re-activated and scheduled for re-presentation to the retrieving process.

This function is not entirely straight-forward, since the re-activation of a packet may require the awakening of a process who is delayed on a queue awaiting the arrival of a packet. Continuing this waiting process forces termination of the audit process, possibly before auditing is complete. This is a restriction of Concurrent Pascal, to be corrected by additional monitor queue operations.

2.2 DATA STRUCTURES

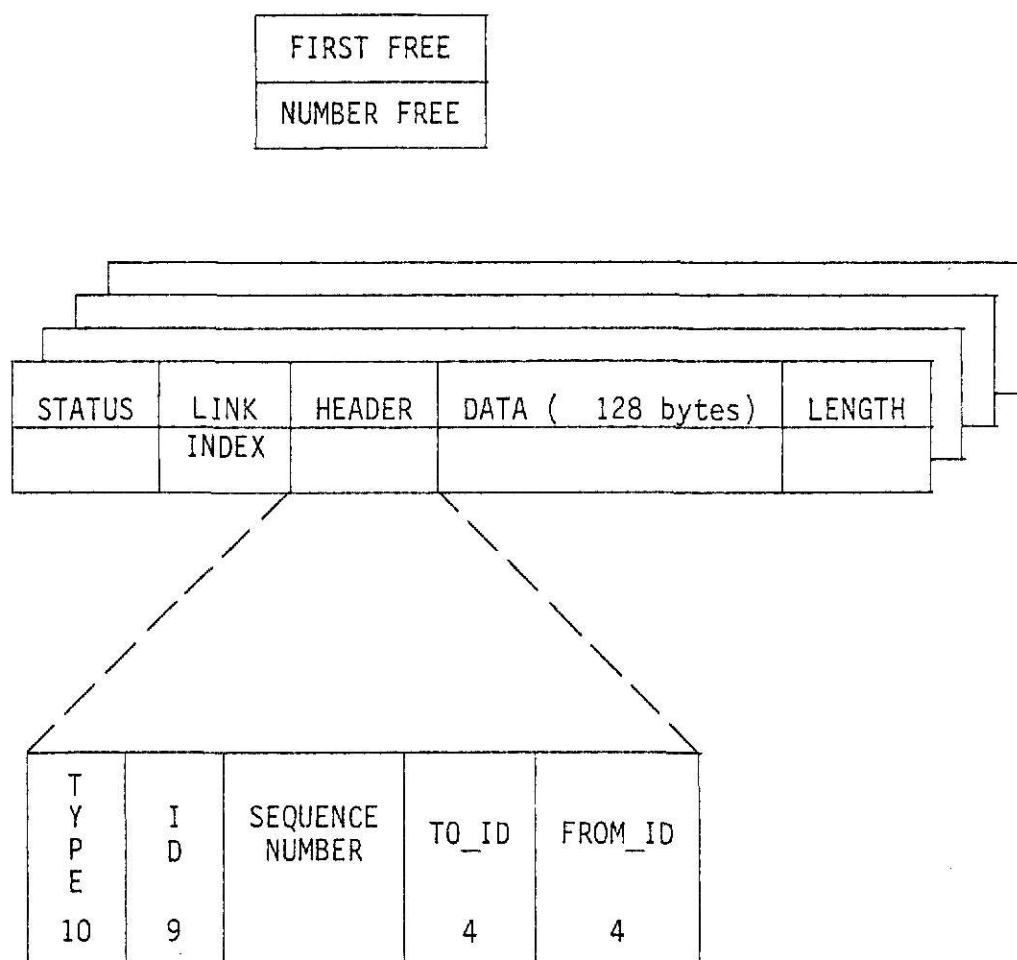
As we have mentioned before, the Packet Buffer Monitor

appears to be two separate entities, each half of which provides a different service, with each half sharing a common pool of free buffers. Indeed this is the case, and the three data structures which make up the backbone of the Packet Buffer Monitor will be discussed as individual entities. Just as the functional description of the Packet Buffer Monitor was subdivided according to the functions provided.

BUFFER POOL

The buffer pool is an array of records into which the actual data packets are copied. As shown in Figure 2.1, the data packets consist of a maximum of 144 bytes of data (including header bytes), and six bytes of local status and pointer information. In addition to the actual data space, each buffer pool record contains a link field containing the array index of the next buffer in whatever list that buffer is currently linked, and a field which is used to indicate the status of that packet. There is also a global free list integer containing the index of the first free buffer pool record, and a free count variable which indicates the number of buffer pool elements which are currently free (i.e. available to be filled with data).

In an effort to promote the orderly flow of data, the last 3 buffer pool records are reserved for INBOUND data packets. This is not an attempt at deadlock prevention, but



LEGEND

TYPE: Command or data packet.

ID:

SEQUENCE

NUMBER: For command packets, the sequence number of messages from this user.

For data packets, the sequence position of this packet within the message.

TO_ID: The fully qualified name of the destination port (Cluster-Machine-Task-Port).

FROM_ID: The fully qualified name of the originator.

Figure 2.1

Packet Format

rather a realization that is less time consuming to delay the movement of packets between the Message (or Cluster) System and the Packet Buffer Monitor than it is (time-consuming) to delay data flow between the line processes and the Packet Buffer Monitor. The former merely results in the delaying of process activity on this machine, and the latter causes the generation of a SACK message, a special response to that SACK, and later retransmission of those same packets (a very slow process relative to computer speeds). Initialization of the buffer pool consists of:

1. setting the free count equal to the number of buffer pool records;
2. setting the free list variable equal to 1, the first free element of the buffer pool array;
3. setting the status of each buffer to FREE;
4. setting all link fields to point to the next array element, and setting the link field of the last buffer to zero to indicate the end of the linked list.

OUTBOUND CONTROL STRUCTURES

The outbound logical line (shown in Figure 2.2) is the most complex of the data structures in the Packet Buffer Monitor, thus reflecting the variety of functions provided for OUTBOUND packets (and the requesting line processes). We will describe a single logical line, but the reader should be aware that the outbound logical line data structure is an array of individual logical lines, where the number of logical data paths emanating from this machine dictates the size of the array.

LOGICAL LINE DATA STRUCTURE

	FORWD	COMND	DATA
SENT			
SACKED			
OUTBOUND			
	NMAX____ NUSED__	NMAX____ NUSED__	NMAX____ NUSED__

- 1) LOCAL PROCESSES CANNOT AFFECT FORWARDING BUFFERS
- 2) FORWARDING PROCESSES CANNOT AFFECT COMMAND OR DATA BUFFERS
- 3) PACKETS WITH A SACKED ID WILL NOT BE ACCEPTED INTO PBM.

	1	2	3
ID SACKED			
TIME SACKED			
NUM SACKED			

Figure 2.2

OUTBOUND Control Structures

The logical line data structure is perhaps best described by listing its capabilities. At any given time, the logical line may contain three types of packets as indicated by their current status.

OUTBOUND-Queued for transmission.
SENT -Transmitted, but no positive response
(SACK or ACK) has been received in
response.
SACKED -Transmitted, but the destination machine
cannot accept packets for the given to
id. (remember that the destination id is
part of the data packet.)

Each of the three packet types are maintained in separate linked lists. The OUTBOUND and SEND lists contain a head and a tail pointer, while the SACK list is maintained as a stack.

Additionally, since we wish to utilize the buffer pool fairly, we need a value to indicate the maximum number of buffers which we can request for this logical line and class, and for convenience we need the number of buffers currently allocated. Thus we have a total of 7 variables.

Since packets may be of either FORWARD, COMMAND, or DATA types, we need these same seven variables for each of the three types. Thus, a given logical line can actually contain 9 distinct types of packets. Notice that this allows different upper limits on the number of buffers available for allocation to each class within the logical line as was mentioned as desirable in the functional description.

In addition to the structure for the packets

themselves, we also need to maintain information about those packets (or, more specifically, the machine identifiers) for which packets are sacked. Since no more than three identifiers can be Sacked before the entire logical line is considered Sacked, each logical line includes a 3 element array of information about Sacked destination identifiers. Included are the identifier, the time at which the first packet with this identifier was Sacked, and the number of packets currently in the Sack queue for that identifier. Initialization of each logical line consists of:

1. setting the maximum allocation field for each of the three classes of packets (FORWARD, COMMAND, and DATA);
2. setting all link fields and the current number allocated fields to zero; and
3. setting the three Sacked identifier fields to be all nulls, and setting the time sacked and the number sacked fields to zero.

INBOUND CONTROL STRUCTURES

Compared to the Outbound control structure, the Inbound control structure is similar but less complex (see Figure 2.3). There are four packet states associated with Inbound packets, but all but one of them occur mainly with command packets, and are not nearly as common as the three types of Outbound packets. Hence, no special data structures for each packet type are maintained. The four types are:

- INBOUND-Original status when the packet is received from the Line process and is awaiting retrieval by one of the four retrieval processes.
- SENT -Retrieved by a process who was unsure of

	FORWARD	COMMANDS	DATA	INVALID
HEAD*				
TAIL*				
DUM_ELEMS				
MAX_ELEMS				
TIME DELAYED				

*fields contain indexes into the buffer
pool array

Figure 2.3
Inbound Control Structures

his ability to process the packet's contents. Packet Buffer Monitor holds the buffer awaiting further instructions as to disposition.

DELAYED-Retrieving process has found that he could not process a buffer which is now marked as Sent. Disposition of packet is to mark as Delayed and await the Audit. The auditor will reschedule the packet and mark it as Old.

OLD -The packet is actually INBOUND, but it has been retrieved in the past, and this is at least the second time around.

Since the frequency of the last three types is quite insignificant from a volume point of view, but is necessary for the simplification of the Message System command receive function, the statuses are provided with the associated functions, but no special linkage structures are maintained. This can be compared to the Outbound packets, where every packet goes through a minimum cycle of FREE-> OUTBOUND-> SENT-> FREE, and could possible include several trips through the SACK structure.

An additional function provided by the Inbound structures is the delaying and continuing of the retrieval processes. Since we can be assured that there is no processing that can be done by a retrieval process if he requests a packet, and there is none available, we assume the responsibility of delaying that process on the queue for which he is requesting a packet until one becomes available, and then insuring that he is restarted (continued) at that time.

Since we keep only one list for each queue, changes in

a packet's state are reflected only in the status field of that packet. No linkage manipulation is performed. Return of a packet to a requesting process consists of sequentially scanning the list until a packet with a status of either INBOUND or OLD is found. Initialization of the INBOUND control structure consists of:

1. setting the maximum allocation field value for each of the four machine ports (SPOOL0, COMMAND, DATA, & CLUSTER); and
2. setting all other fields to zero.

2.3 ENTRY PROCEDURE DESCRIPTIONS

We now describe the entry points which perform the individual functions we have already discussed. Each entry point contains a description of calling parameters, function, and a pseudo algorithm. The order of appearance here is alphabetic, not functional. Figure 2.4 shows the relationship between the calling processes, the monitor entry points, and which data structure each entry point accesses.

Audit

One process in the MIMICS system is responsible only for calling every monitor in the system at specified intervals to allow that monitor the opportunity to clean up its data structures. Since a process may be delayed on a queue associated with the INBOUND control structures, it is

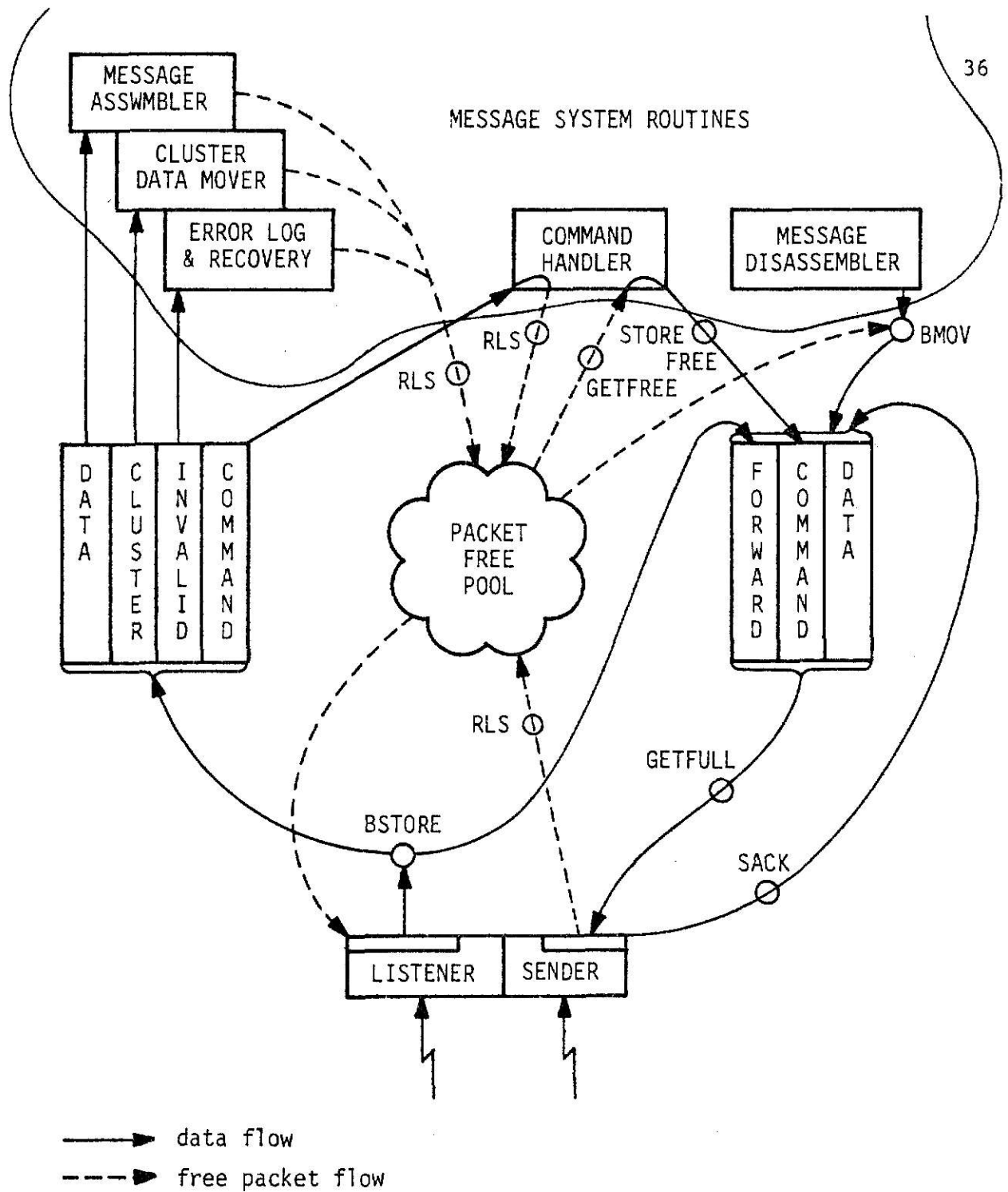


Figure 2.4

Logical Packet Flow
(with Packet Buffer Monitor calls)

possible to leave the monitor before completely checking the data structures (forced exit if we continue another process). Before this happens, we set the 'Repeat' parameter true before performing the operation. This acts as a flag for the Auditor process, and the call will be repeated immediately.

Auditing the OUTBOUND control structure involves the following. The time-sacked field for each sacked identifier is compared to the current time, and a check is made to see if the difference in time is greater than the sysgened 'sack-lapse'. If it is, then that identifier is available for re-activation, and all packets associated with that identifier (C.M.T.P) must be removed from the sacked list and placed on the outbound list. Additionally, the sacked identifier record must be reset.

On the INBOUND control structure, the audit function consists of locating packets which have been delayed, and determining if enough time has elapsed (by comparison of elapsed time since delay to the 'delay-lapse') to make the packet available for re-presentation to the proper process (Cluster, Spool0, Command, Data). The status of the packet must also be changed from DELAYED to OLD.

Additionally, since a message assembly process may be delayed awaiting reception of a packet for processing, the auditor must be responsible for continuing a delayed process if a packet is made available for that process. Normal

completion of the auditing procedure results in the assignment of a false value to the parameter 'Repeat', and the audit is complete, to be repeated at the expiration of the next AUDIT_INTERVAL.

Bmov

This procedure adds an OUTBOUND packet to the Packet Buffer Monitor data structure. 'ptr' is the packet, 'Classs' indicates whether the packet is to be placed in the Command or Data queue, 'Route index' indicates the entry in the route table (see chapter 3) from which the correct logical line for sending this packet may be obtained. As can be seen, this allows the physical line to change at run time, and in fact the logical line may change to allow for rerouting of packets of all of the physical lines which make up a logical connection fail.

Several checks are made before we accept the packet for transmission across the logical line. If the destination id is Sacked, then the packet is rejected in an attempt to reserve buffers for those packets which can move across the line. If less than 3 free packets are left in the buffer pool, then the packet is rejected, since those last three packets are reserved for Inbound packets. If the given Class has reached (or exceeded) it's maximum allocation then the packet is rejected.

If the packet is not rejected, then a free buffer is

obtained from the free pool data structure, linked to the tail of the Outbound data structure for the correct class and logical line, and the packet is copied into the buffer pool data space. Possible error codes are:

BUFFERS_FULL	-Less than 4 packets remain in the free pool structure.
CLASS_FULL	-The allocation limit for this class/logical line combination has been reached (or exceeded if limits are allowed to change dynamically).
SACKED_ID	-This source-destination combination has had previous packets sacked in the recent past; no further packets will be accepted for transmission until the 'sack-lapse' has expired.

Bstore

The BSTORE entry is the Inbound equivalent to the BMOV entry for Outbound packets. All packets received by a physical line process are given to the BSTORE entry for further processing. These packets should fall into two classes: packets destined for this machine, and packets which are destined to another machine for which we have a logical connection. This latter case is the store-and-forward packet. A possible third case occurs in the event that the packet is not destined for this cluster, but we have no route for the destination cluster. In this case, the packet is passed to the Spool0 process in this machine for any possible recovery or error diagnosis.

Decoding of the packet is based on the 'TO_ID' in the packet for determination of destination cluster and machine,

and the 'MSG_TYPE' in the ninth byte of the packet (see Fig. 5.1 for packet format) for determination of either data or command packet type.

If the cluster identifier in the to-id is this cluster, then we are sure that the packet will be placed in either the CLUSTER, DATA, or COMMAND queues. Further decoding is done on the machine identifier. If it is not this machine, then the packet goes on the Cluster queue for routing to the proper machine within this cluster. If it is destined for this machine, then we look at the message type to determine whether it goes on the Data or Command queue. Once we have determined the queue on which the packet is to go, it is a simple matter to request a packet from the free pool structure, link that packet to the end of the proper queue, and copy the data into the buffer pool packet.

Two possible return codes are possible, based on whether the packet was to be forwarded or was for the local machine. In either case, it is assumed that the caller will use the non OK return code to generate a sack packet (if this operation is supported). The return codes are:

LINE_SACKED	-Route table contains path to destination machine, but that
SACKED_ID	-This destination identifier has been sacked, and the time lapse necessary to retry the sending of packets has not yet expired.
LINE_FULL	-The allocation limit for this Inbound queue has been reached.

Copy

Copy is a two line routine designed to allow the Logical Line Window Monitor to get a copy of a packet for retransmission. It could also be used by any process who wished to look at the contents of a packet. The value of index indicates the packet in the buffer pool to be copied, and the indicated packet is copied into 'PTR'. No checking at all is done, a return code of OK is always generated, and no manipulation of linkage is needed.

Getfree

This entry point was provided to make the coding of the Message System somewhat easier, and is an example of the structured manner in which modules may be added to provide additional function within CPascal. The function of this entry is to ensure that the Message System has a buffer available for transmission of a command packet, before the code is entered which will actually build the packet. This entry forms a pair with the STOREFREE entry which is functionally equivalent to the BMOV entry; the extra time required for the double call was deemed justified by the savings in time in the Message System and the ease with which the Message System makes use of this entry.

The route index is used to determine which logical line the packet is to be allocated to, and it is assumed that it will be placed in the Command class. The same checking

which is done by BMOV is done to insure we are capable of accepting the packet, and then the free pool structure count is decremented by one. This effectively pre-allocates one packet for later use, and the logical line command count is incremented. No linkage manipulation is done until the packet is actually ready to be stored and the STOREFREE entry is called.

Several things should be noted about this test. First of all, due to the order of linkage and count manipulation, there will be one buffer more on the free list than is indicated by the count and one less in the command logical line than is indicated. By use of this mechanism, the Message System can effectively bypass the Sack mechanism of the Packet Buffer Monitor since we cannot know the destination id of the packet to come later, and we cannot reject a packet at STOREFREE time which we have already promised that we would accept. Finally, the mismatch between the counts and the actual number of linked packets causes no problems since searches on the data structure rely on a zero link to indicate the end of the list, and not on the count. The purpose of keeping the counts is designed to prevent the total number of Commands from exceeding its limits. Consider the case in which several GETFREE'S have been made followed by several BMOV'S before the matching STOREFREE'S come in. If we did not do the counts until the STOREFREE's then we would be guaranteed that we would exceed

the limit for the class. Possible error return codes are:

BUFFERS_FULL-Less than three free packets are left
in the free pool structure.

CLASS_FULL -The allocation for the Command class
of this logical line has been reached
or exceeded.

Getfull

The capabilities provided by this entry point are utilized by communications line processes or the Window Monitor (see Chapter 4) to obtain an Outbound packet for transmission across the telecommunications line. If an Outbound packet is available, then its index is returned in the variable 'INDEX' and the data from the packet is copied to 'PTR'. The return code is used to indicate why no packet was returned, or else OK. Since the Outbound structure keeps the packet until specifically told to release it, the 'index' provides the mapping between the calling process and the Packet Buffer Monitor.

Release

This is one of two entries which complement the Getfull entry. The line parameter indicates which logical line we are to reference, and index is the buffer pool index of the buffer which we are to release. This entry implies that the indexed packet was obtained by a previous call to Getfull, the packet was acknowledged by the receiving machine, and we are now free to get rid of it.

The internal procedure Remove_Old is called to remove the packet from the SENT list, and the packet is then added to the free pool. The only return code is if the index value indicates a packet which is not in the Sent list. The return code is ERROR and indicates a severe error in the calling process's code.

Retrieve

This entry point is utilized by message assembly processes to retrieve (hence the name) packets which have come into the machine from the telecommunications lines. If no packet is available, then the caller is delayed on a queue variable for the class from which he requested the packet. Implicitly, therefore, only one process per class should ever access the Packet Buffer Monitor. Failure to do this could result in a second process being delayed on the queue containing the first process, thus losing the first process.

The retrieve mode allows three different types of retrieval. A retrieve mode of 'RLSE' causes the return and immediate freeing of that buffer, and is intended to be used with data retrieval. A retrieve mode 'HOLD' is valid only for command buffers, and causes return of the packet without freeing the buffer. The buffer is marked as 'SENT', and the caller will later modify this status through another call. Retrieve mode of 'ACCESS' requests retrieval of a 'SENT'

packet and immediate release of the packet. Hence, packets which have been retrieved with mode 'HOLD' can be freed by another call with mode 'ACCESS', and may also be delayed or released via the RLS_DELAY entry point.

Release Delay

Used as discussed in the entry point Retrieve. The mode indicates whether the packet is to be delayed or released, and works only with packets which have a status of 'SENT'. No return is included, so no check of message system correctness is made. It is assumed that index correctly points to a packet with the proper status. If this is not the case, then the call is ignored, but no indication of this is returned.

Since only one field is retained for the time at which packets were delayed, delaying of a packet on a queue which already contains a delayed packet results in both having the same delay time, and both will be freed at the same time.

3.1 FUNCTIONAL SPECIFICATION

In the MIMICS system, communications between any two host machines occurs across a path referred to as a "logical line". It is important to notice that there is no binding between a logical line and any physical communications path. It is, in fact, likely that multiple physical paths might implement a logical path. It is also likely that the physical paths will change as the system runs and communications lines are connected and disconnected.

It is the Route Table Monitor which performs the mapping from a two character identifier (cluster and machine) into a logical line number. It is also the Route Table Monitor which determines an alternate logical path in the event that the current path is no longer available (either temporarily or permanently).

In deciding how the Packet Buffer Monitor might most effectively make use of the capabilities provided by a routing function, several implementation questions and possibilities occurred. From a design point of view, it seemed obvious that the Route Table Monitor was a separate entity which would be used primarily by the Packet Buffer Monitor in determining where a packet should be sent. This would, however, incur a nested monitor call from the Packet Buffer Monitor to the Route Table Monitor for almost every packet sent thru the packet switch, resulting in a significant degradation of packet throughput. To solve this

problem, we decided that the Route Table Monitor should reside internal to the structure of the Packet Buffer Monitor, eliminating the nested monitor call in the general case at the expense of increasing the load on the Packet Buffer Monitor for out of the ordinary calls on the Route Table Monitor. This seemed an equitable trade to make. In a further attempt to improve performance, we allow the message system processes to tell us which route table entry they are using to communicate with a given machine. This virtually eliminates the overhead associated with determining the route for Outbound packets. However, we cannot actually allow the message system processes to know the logical line, since we would lose the dynamic routing of the route table. That would imply that a logical line for a given connection is constant, and this is not the case. We desire to be able to dynamically route packets across a new logical line in the event that the old one becomes unavailable; or merely overloaded in sophisticated routing schemes. Therefore, we allow the message system processes to know the route table entry from which the logical line can be obtained, but not the logical line itself. This eliminates the search of the route table based on the packet identifier while maintaining the dynamic routing capabilities of of the Route Table Monitor.

3.2 DATA STRUCTURES

While the Packet Buffer Monitor has the most complex data structures in the packet switch, the Route Table Monitor has the simplest. The data structure is a single route table. In this route table are 5 elements for each route, where the number of routes is a sysgen constant.

The route-id is a two character field containing the destination cluster and machine identifier. Thus, this element contains the logical line routing information for packets whose destination cluster machine identifier matches this route id. Searching for a route involves searching the route table for a route id which matches the destination identifier in the packet.

The 'state' field identifies the current activity state of the logical connection. States of 'prim' and 'secdry' indicate that a logical connection exists, and that either the primary or the secondary logical line is to be utilized in implementing this logical connection. States of 'dormant' and 'unused' indicate the absence of a connection for somewhat obvious reasons, while 'down' implies that a connection did exist at some time in the past which is no longer implemented.

The next two fields are the indications of which logical line to use if the state is either primary or secondary. The first field, called 'prim_line', is the logical line number to be used for implementing a connection

across the primary logical path, while the second field is called 'sec_line' and serves the same purpose after it has been determined that the primary path is no longer viable. These values may be static from the time the system is compiled, modified at runtime, or a combination of both. Note that changing from primary data flow to secondary data flow is an dynamic operation, and is a primary design goal of the Route Table Monitor.

The final field is a twelve character identifier maintained purely for the purpose of allowing easy human comprehension of the real locations and machines associated with a cluster machine identifier. This field is not used for any purpose other than printing the contents of the route table. The goal was to minimize the overhead involved in sending packets, thus the representing of each unique machine as a two character identifier. This is not compatible with human thoughts, however, since we tend to think of an IBM 370-158 as exactly that, not as cluster 'Q' and machine 'Z' in that cluster.

The reader should note that, since every machine in the network should have access to every other machine in the network, the route table should have as many active elements as the number of machines in the network. Additionally, since the most secure implementation of MIMICS exists in a communications controller which is distinct from the host machine and the user programs, establishing a route table

with limited entries effectively limits the scope of the network which that host sees. The reader should also be aware, by now, that there is no correspondence between any connection path and the logical line used to implement that path, or between the connection path and the actual number of physical paths between machines which may be traversed in the transfer.

3.3. ROUTE TABLE MONITOR ENTRY POINTS

As was the case with the Packet Buffer Monitor, the order of presentation here is alphabetic. However, since many of the more sophisticated routing schemes require the presence of some Network Resource Scheduler or Command Processor, some of the procedures are not implemented in the current code. The algorithms presented seem straightforward, but should not be presumed to be debugged, as has been the case with all previous entry points. Similarly, since functions and requirements of an NRC are not clearly understood at this time, it is entirely conceivable that additional routines might be required at some time in the future.

Connect Route

This routine is utilized to fill in a new entry in the route table. The caller could follow one of two possible

schemes. In a network which attempts to maintain security at runtime, this entry should only be called by some NRC process who can verify the validity of the machine who is being added to the network. In our implementation, the route table is updated by the line process as soon as the physical connection is completed. Since this connection is assumed completed as soon as the two machines have exchanged their respective cluster machine identifiers, there is no attempt made at verification of whether or not the user is allowed to join the network. Another method is merely to establish the route table statically and not allow any runtime modification. Possible return codes are:

RT ERROR	-The route table entry already exists for the indicated cluster machine pair.
TABLE FULL	-The indicated cluster machine is not in the route table, but there is no more room for new entries. Recommend that table size be increased.
OK	-The requested operation has been performed.

Get Route

This routine is used to obtain the route table index and the logical line for a given cluster machine identifier. Although we stated earlier that to allow the message system processes access to the actual logical line in use would defeat the dynamic routing capabilities of the packet switch, we return both values in an effort to generalize the use of the entry point. When a physical line process first communicates with its counterpart on another machine, they

exchange cluster machine identifiers. The line process on this machine then takes that identifier, calls the GET_ROUTE entry, and uses the logical line information returned to determine which logical line he is to obtain packets from. Thus, we see that the route table is, in a sense, used in the logical line to physical line mapping. Also, the message system processes responsible for generating packets need to know which element of the route table is used to determine the logical line. Therefore, this entry point returns the logical line for use by the line processes and the route index for use by the message system processes. The only two possible return codes are:

NO ROUTE	-The requested cluster machine identifier does not have an entry in the route table.
OK	-This is a self-explanatory return code.

Set Status

This routine would be used to update the status of the route table elements. It is currently done implicitly by the Connect Route entry, which changes the status of the connection from dormant to primary. However, there is currently no entry point or routine which determines that a connection needs to change from the primary line to the secondary line (for example), or to determine that the connection has been broken and cannot be recovered. Possible arguments might include a logical line to be used in implementing a route, such that the state might remain

'primary', but the logical line used might be changed to a new value.

Changing the logical line over which packets are transferred is entirely transparent to all parties if the logical connection is in a quiescent state (no packets queued for transmission). If this is not the case when reconfiguration becomes necessary, then the recovery across the new line is far from trivial. This is the problem. At the sending end of the physical line, we know which packets have already been sent and which were merely awaiting transmission. Of those sent, however, we have no easy of determining which have actually been correctly received, independent of which the other end of the line says have been received.

4.1 INTRODUCTION

The implementation of a logical line window monitor is, perhaps, the only concept which is entirely original to this project. The notion of utilizing a window for controlling full duplex flow of traffic is widely accepted; and an example of this use is well documented in a paper by Gregor V. Bochman[6]. However, the typical window is only utilized for controlling flow over a single communications link.

The unique characteristic of our window monitor is the existence of a single pair of windows at each end of the logical line, as opposed to the typical implementation of a pair of windows per line process. If the window is in the line process, then all return status about a window is limited to that physical line. Since our window is defined to cover the logical line, return status may come across any of the possible physical lines which make up the logical line. This allows a number of interesting capabilities.

No longer do communications have to be either half-duplex or full-duplex. The communication across the bandwidth of the logical line is full-duplex, entirely independent of the various line protocols utilized to implement the logical line. If the system programmer decides that traffic is quite heavy in only one direction, then the logical line might efficiently be implemented with several simplex lines in the direction of heavy traffic with

only a single line in the reverse direction. This return line would effectively notify the originating window monitor of which packets have been correctly received over all of the other lines (see Figure 4.1).

Lines can be dynamically added to and deleted from the logical line window monitor to allow adjustment to the current flow of traffic. All that is required is the addition of a mechanism to detect levels of traffic which require an additional line, or when traffic levels have declined such that some lines may be dropped. The window monitor allows the addition and deletion of lines dynamically. Since no particular window information is tied to a given line, the line may go away at will. The window monitor will continue to sequence packets properly (and will retransmit any packet lost when the lines disconnected).

Another important capability concerns throughput under conditions of a failing line (or highly error prone line). Since the windows in other protocols are maintained in the line processes, any return status must also come across that line. If the line dies, then detection and recovery must rely upon several successive timeouts before any recovery algorithms can be initiated. If we view the situation where several lines are transmitting packets which make up a message, a single packet can lead to failure (or slowness) in reception of the entire message. Since a given packet is not bound to a line in our implementation, retransmission

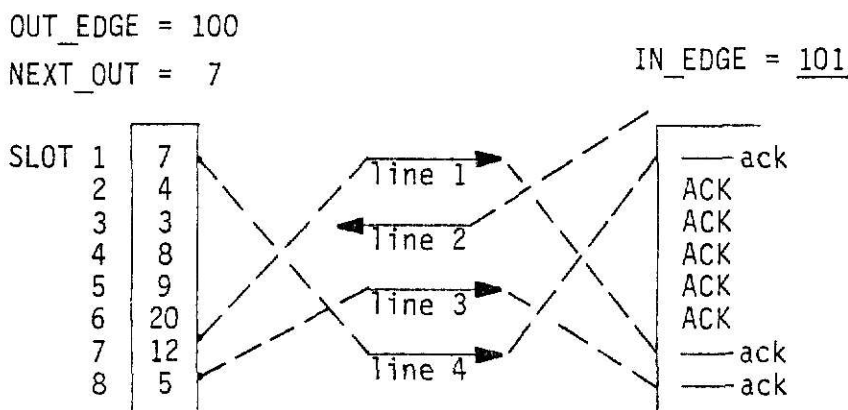
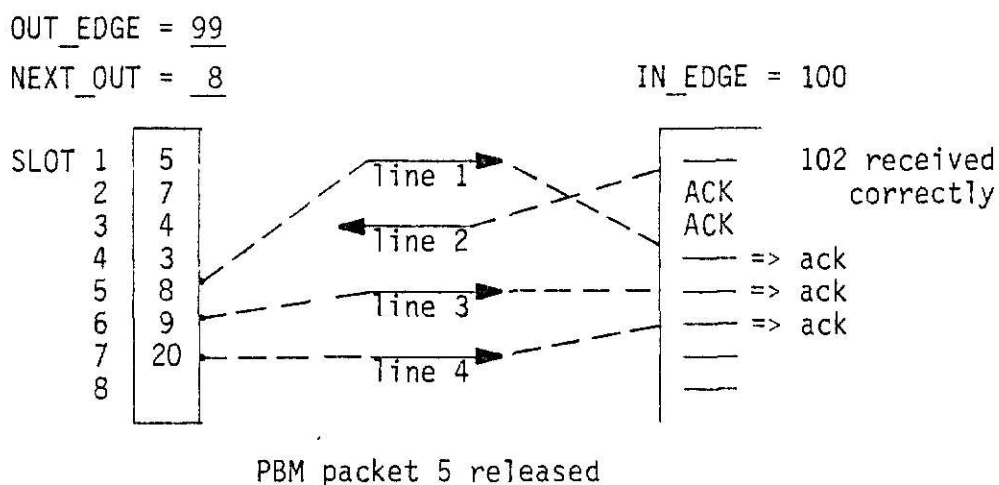
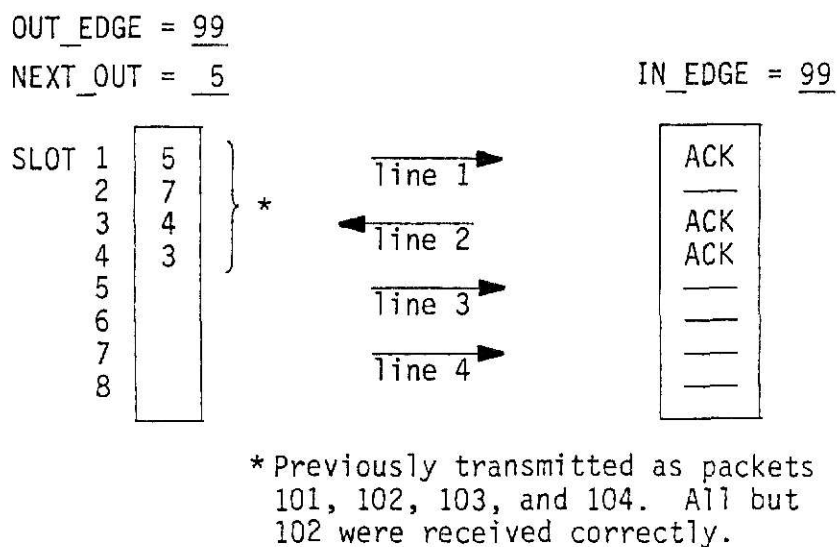


Figure 4.1

Window with Unidirectional Data Flow

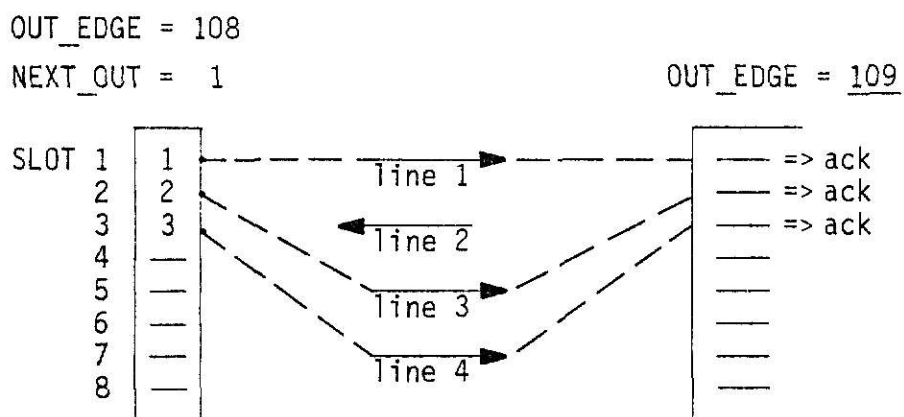
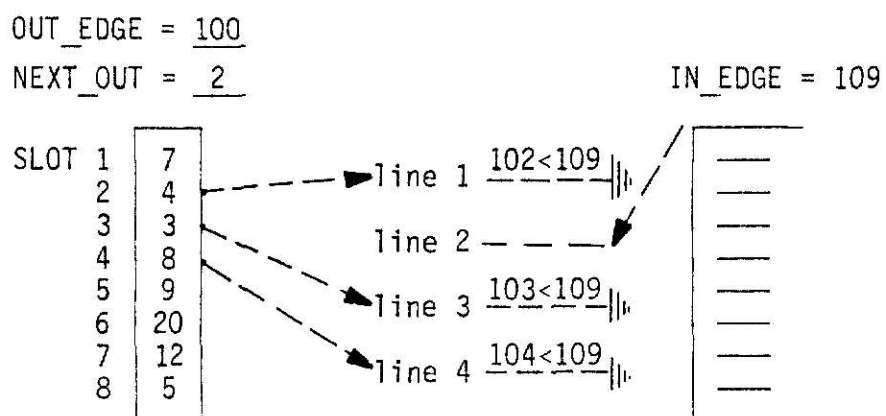


Figure 4.1 (Continued)

SEND WINDOW

OUT_EDGE _____
 NEXT_OUT _____
 OUT_WINDOW
 OUT_WINDOW

--	--	--	--

 1 2 ... n

OUT_EDGE: The highest numbered packet sent by us and acknowledged as received

NEXT_OUT: The out-window index of the next packet to be transmitted. The sequence number of this packet will be OUT_EDGE+NEXT_OUT

OUT_WINDOW: An array (of window size) of addresses of packets (or PBM array indexes) to be sent.

RECEIVE WINDOW

IN_EDGE: The highest numbered packet for which we have generated an ACKnowledgement message.

OUT_WINDOW: An array of tri-state flags indicating the status of corresponding packets. States are:

ACK: Packet received correctly

SACK: Packet received, but cannot be accepted

NULL: Nothing received

Legend for Continuation of 4.1

will almost certainly not occur on the same line (since this error prone line will still be awaiting time-out). Therefore, even though one line is unreliable, messages do not get hung up awaiting one packet out of many. The packet will be retransmitted automatically as soon as the window fills up with unacknowledged packets.

4.2 FUNCTIONAL SPECIFICATION

The Window Monitor performs the functions of determining the sequence number associated with each packet which is transmitted, returning status to the sender about those packets which have been received, and attempting to promote throughput by sending and receiving packets within the range of the window before earlier sequenced packets have been acknowledged, and accepting packets which arrive out of sequence. To accomplish these functions, three bytes of information are appended to every packet which is transmitted. The first two bytes are used to return information to the sender about packets received up to this point. The first byte is the control byte (RCNTRL), and the second is the sequence number associated with that control item (RSEQ). The possible control bytes and their meanings are:

ACK	-All of the packets with sequence numbers up to and including RSEQ have been accurately received. This information will allow the sender to release all of these packets.
-----	---

- NAK -The packet with the sequence number which accompanies this control item has been accurately received, as have all earlier sequence numbers. However, an error has been encountered and retransmission of all outstanding packets is requested. Again, the sender can release packets.
- SACK -All packets with this sequence number and earlier have been received, but this packet cannot be accepted due to lack of buffers, or other reasons. The sender can release all previous buffers, but must return this packet to the buffer pool for later retransmission (when retransmitted, the packet will have a different sequence number). The sender must positively respond to receipt of the Sack control item.
- SACKRSP-This is the positive response to the Sack control item. The originator of the Sackrsp is telling the receiver that he has accurately handled the Sack item, and that the given sequence number can be skipped.

The other header byte is the sequence number of the data packet associated with the line packet. If there is no data, then the sequence number has no meaning. Notice that a data packet can accompany any control transmission, so during periods of high traffic, virtually no overhead is incurred in exchanging control information.

4.3 DATA STRUCTURES

There are two data structures which actually implement the two windows which are necessary to maintain the window of packets sent and the window of packets received. These are referred to as the send window and the receive window respectively. Also, since the header of the next packet to be sent is built based on information received, a dummy

header entry is maintained for modification as a result of incoming return status, and then prefixed to the next outgoing packet. As soon as that packet goes out, then the new dummy header is built with a standard ACK format. This interaction between two different entry points is perhaps difficult to understand.

The reader should also understand that the window may be of any size, even though the sequence numbers range from 0 through 255. It was also decided to represent the window as offsets from a base number, since this base sequence number is often needed for ACK prefixes to packets.

Receive Window Structure

The purpose of the receive window is to maintain the status of all sequence numbers represented by the slots in the window, where the sequence number can be found by adding the left edge of the window to the displacement of that slot in the window. Initially, the left edge of the window is 255, and the first slot would imply a sequence number of 0 (sequence numbers are transmitted as a byte). If packet 0 is received, then slot 1 is marked as received. If packet 3 is received, then slot 4 is marked as received. If packet 10 is received (and the window size is 8), then the packet is ignored. If the window slot for a packet is already marked, then the packet is ignored.

After the window slot is marked, an attempt is made to

update the window by starting at slot 1 and updating the left edge by one for every slot which is marked as received. If slots 2, 3, & 4 were marked, and the left edge were 10, then reception of packet 11 would mark slot 1 and the left edge would be updated to 14. The slots themselves must then be adjusted to reflect the status of the new sequence number with which they are associated.

A special case exists if a packet is received with a valid sequence number, but which the Packet Buffer Monitor will not accept. If a packet of this type is received, then the slot is marked as 'SACKED' and cannot be updated until a 'SACKRSP' is received for the specifies sequence number. When the window is updated, and the first slot in the window has a 'SACK' marking, then the control character on the next transmitted packet is forced to be 'SACKED'. This 'SACK' marking is updated to 'ACK' upon reception of a special sack-response message, and the window can then be updated. An attempt will be made at a later time to retransmit the same packet, but with a new sequence number. Care must be taken with the sack mechanism to insure that sacked packets are recognized as such by both window monitors, or the probability of receiving the same packet twice with different sequence numbers or of not receiving a packet at all is great.

Send Window Structure

The send window is used to associate a sequence number with a given packet. The contents of a send window slot is the address (or Packet Buffer index) of a packet. The left edge of the send window and the window slot can be added to determine the send-sequence number of this packet. If the slot is empty, then a new packet is requested of the Packet Buffer Monitor and it's address is stored in the slot. If the slot is flagged, then the packet associated with that slot was sacked and a SACKRSP has not yet been generated. One is in order.

Retransmission of packets occurs under several circumstances. If the window is full, and the last packet was from the right edge (highest sequence number), then the next packet will come from the left edge of the window. It is assumed that a packet has probably been lost. If a packet is received with a RCNTRL character of 'NAK', then retransmission occurs after the RSEQ character is used to update the left edge. If the next slot is empty & there are no additional packets are queued in the Packet Buffer Monitor, then retransmission occurs from the left edge.

4.4 WINDOW MONITOR ENTRY POINTS

Since the Window Monitor responds to requests for packets to send and requests to store packets which have been received, it would be expected that there would be two

major entry points to implement these functions. In addition to these two entry points, there are two additional entry points used by cooperating line processes to allow the window monitor to determine the number of line processes currently utilizing the capabilities of the window monitor for this logical line. A fifth entry is implemented, but not really utilized. This MAIL entry point was intended to allow a command processor to communicate with the line processes for the purpose of rerouting.

Connect

This routine merely increments the counter maintained for each logical line, indicating the number of line processes utilizing the capabilities of the window monitor has been increased by one. Obviously, this should be one of the first calls made by a cooperating line process.

Disconnect

This entry point is the opposite of the connect entry point. At the current time no action is taken when the number of connected processes goes to zero, but this is the point at which rerouting logic could be invoked. Rerouting would only take place if the number of connected processes became zero and there were packets in the send window which had not been acknowledged.

Take

This entry point is utilized by the line processes when they request a packet to be transmitted over their physical line. The take procedure builds the 3 character header discussed previously based on information deposited by the GIVE entry point (see next section). It selects a packet to be transmitted based on the current status of the send window, and determines the sequence number of the packet based on the slot in the window.

Give

This entry point is the complement of the "take" procedure. Line Packets which the line processes have accumulated are "given" to the window monitor for processing. The window monitor uses the information accumulated in the header to update its local data structures (primarily the out window) and to determine whether or not the data in the packet should be passed on to the Packet Buffer Monitor (insures that this is not a retransmitted packet).

It is assumed that the take entry has set the RCNTRL character for the next packet to go out to ACK. Therefore, this entry changes that variable only if something other than an ACK is to be transmitted. This would occur only if a line process called the entry point with no packet, but with a Boolean set indicating it had detected a line error.

In this case the window monitor should indicate to its counterpart at the other end of the logical line that retransmission should occur (NAK). The second occurrence of changing of the RCNTRL variable would occur if the received packet would not be accepted by the Packet Buffer Monitor. The appropriate response in this case would be a SACK. The third case is the receipt of a SACK packet. The window monitor must then respond with a SACKRSP packet.

Notice that all of the instances delineated above are uncommon (hopefully) and that the normal action is to update the out window based on the information in the header and to determine whether or not the packet received can be passed along to the Packet Buffer Monitor (insure that the sequence number has not already been used).

5.1 BACKGROUND

The half duplex line process is that entity of the Packet Buffer system which actually insures that data packets are accurately transmitted from one machine to another. Although this particular line process uses half-duplex synchronous hardware facilities, any other protocol or hardware communications could be utilized with no changes in any of the higher levels of the software. This heirarchical isolation of function was one of the primary factors in the choice of CPascal as the implementation language.

The line process was probably the easiest module to write, and the protocol was one of the earliest defined parts of the entire Message System (largely due to the bottom up nature of the design). Although the protocol developed has several advantages over IBM's Bisync protocol [17], the main area of effort was the actual implementation of synchronous capabilities; i.e. use of Interdata's ITAM [8] subsystem.

The problems came in two forms. First was the problem of adding the desired interface from the kernel of the PASCAL system to the SVC 15 level of ITAM. The second area of frustration was in the actual use of ITAM with our hardware configuration. I would estimate that allowing the PASCAL system the capability to utilize ITAM occupied at least two months of very steady work.

Areas of frustration and work were as follows. Almost total lack of comments in the kernel. Without David Neal, the task would have been even more complex. His tutelage was (and is) appreciated. Addition of a completely different IO scheme to accomodate the SVC 15 call; the existing IO was accomplished via SVC 1 calls [20]. However, these were minor problems compared to the problems encountered when attempting to use ITAM.

Probably the most frustrating experience was in getting help from Interdata concerning our configuration. The personnel involved were very cooperative, but had never had an 8/32 processor and had never tested ITAM on the Quad Synchronous Adaptor. The device driver was advertised as supporting either the QSA or the 201 dataset adaptor (a single line adaptor for synchronous communications). Unfortunately, we had the QSA with an 8/32, and no factual information was available for that particular combination. To this day, in fact, executing the ITAM driver in a local loop-back test mode will cause the machine to hang in an infinite loop with interrupts disabled (I have long since lost the necessary patch to circumvent this).

In summary, this enumeration of the problems encountered seems relatively miniscule; at the time the learning experience was quite frustrating. It was, however, better than an experience with a similar package for the Data General Nova, since the ITAM package did work. Also on

the plus side for Interdata is the design concept of the ITAM subsystem itself. Two views of the system are provided: the SVC 1 level is for the average user who desires to have the fact that telecommunications facilities are being used hidden from him; the SVC 15 level is provided to give more sophisticated users predefined line control capabilities with a user provided protocol [8].

5.2 FUNCTIONAL SPECIFICATION

Due to the wide usage of the IBM Standard Bisync protocol and the availability of synchronous hardware, it was decided to use a synchronous protocol. However, the structure of Bisync seemed not to fit our needs. In particular, the inconvenience of having to bid for the line before every transmission via the ENQ message and respond with an ACK seemed to induce unnecessary overhead.

It was felt that we could reduce the overhead in two ways. If the half duplex line was used in such a way that each end alternated as sender and then receiver, then the necessity to bid for the line could be deleted. Obviously, this means that all communications are point-to-point, but that was an original specification of the message system design anyway. The other inefficiency was the requirement of an ACK in response to every packet transmission. We felt that we could piggy-back this information on a data packet

(if one were available) and thus eliminate unnecessary response packets (see Fig. 5.1 for packet format). If no data is available, then the protocol degenerates to the Bisync case whereby we have transmission of a data packet followed by transmission of an ACK packet. The real benefit lies in periods of high traffic (which is when we most want good throughput). There are no extra response packets transmitted, since each data packet also has the response information for the previously received packet piggy-backed at the front.

Since we have specified that either end of the line is either in a send or receive state, and that they alternate, we must turn the line around even if we have no data to send. This allows the other end the opportunity to transmit any available data. It should be obvious that lack of data to send results in the idling of control packets to continually turn the line around. It also means that a given packet may have to wait some period of time before the line is turned around and serious data transmission can begin. A line process whose turn it is to send may delay before turning the line around only if both there is no data to send and the last packet received had no data. If either condition is not true, then the line must be turned around immediately, either with data or with a control packet to allow the other end to send data.

It seemed an obvious decision to sacrifice the delay

packet header																
line header			WINDOW HEADER		MSG SEQ #		TO ID		FROM ID		data		line trailer			
Ø	D	S	A	1	1	24	A	B	T	1	B	T	1	from 0...128 bytes	D	E
	L	T	C	0	4										L	T
	E	X	K	0	7										E	X

Assume this packet is from machine BB and is being transmitted to machine CA, who will then forward the packet to machine AB.

LINE HEADER:

- Ø: From 1 to 6 SYN characters necessary to allow hardware to synchronize.
- DLE-STX: Line control character marking the beginning of the data bytes.

WINDOW HEADER:

- ACK-100: Machine BB has accurately received all packets through sequence number 100 from machine CA.
- 147: The sequence number of this packet is 147.

PACKET HEADER:

- 24: The message sequence number indicates where in the message this particular packet fits.
- A.B.T.1: This packet is destined for task T.1 on cluster/machine AB.
- B.B.T.1: This packet was originated by task T.1 on machine BB.

PACKET HEADER: From 1 to 128 bytes of information which actually make up the packet.

LINE TRAILER: This two-character sequence indicates the end of the transmitted packet.

Figure 5.1: Packet Format

and additional packets incurred in a low traffic environment for the efficiency of transmission in a bi-directional high traffic environment. In a uni-directional high traffic environment, the MIMICS protocol is the functional equivalent of Bisync minus the ENQ message and its response. In a bi-directional high traffic environment, the MIMICS protocol is markedly better than Bisync. In a (bi-directional?) no traffic environment, the MIMICS protocol results in the transmission of untold numbers of idle packets where the frequency is determined by the allowed idle delay, but this would seem to be of little consequence. The lack of traffic probably indicates lack of processor load on the connected machines (although not necessarily) and implies that there is plenty of processor capacity available for what little is required for sending idle packets. The MIMICS synchronous protocol seems like a good, load-sensitive protocol; that is the efficiency improves as the load increases.

5.3 EXTENSION TO OTHER PROTOCOLS

It should be obvious that implementation of other protocols would be quite straightforward using either a different synchronous protocol, or using a different protocol and a different line discipline (SDLC, HDLC, DDCMP). Implementation of this protocol using a different line

discipline would be very transparent to the line process, since all changes would exist in the driver portion of the ITAM subsystem. The only synchronous dependencies in the line process involve the insertion of DLE-STX and DLE-ETX sequences for transmission and removal of these same sequences after reception. These two sequences mark the beginning and the end of the packet respectively, and are localized to the procedure READ and WRITE within the line process. In an application environment, these internal procedures would be declared as CPascal classes. The code for changing line disciplines would still be very localized.

5.4 DATA STRUCTURES

The only data structures in the half duplex line process are a buffer for manipulating line packets before they are sent and various error flags and counters and an Action variable. The most important to understand of these, is the variable called 'Action'. The line process has only 9 states, and the diagram in Figure 5.2 indicates the possible state transitions. The Action variable is used to emulate this state transition.

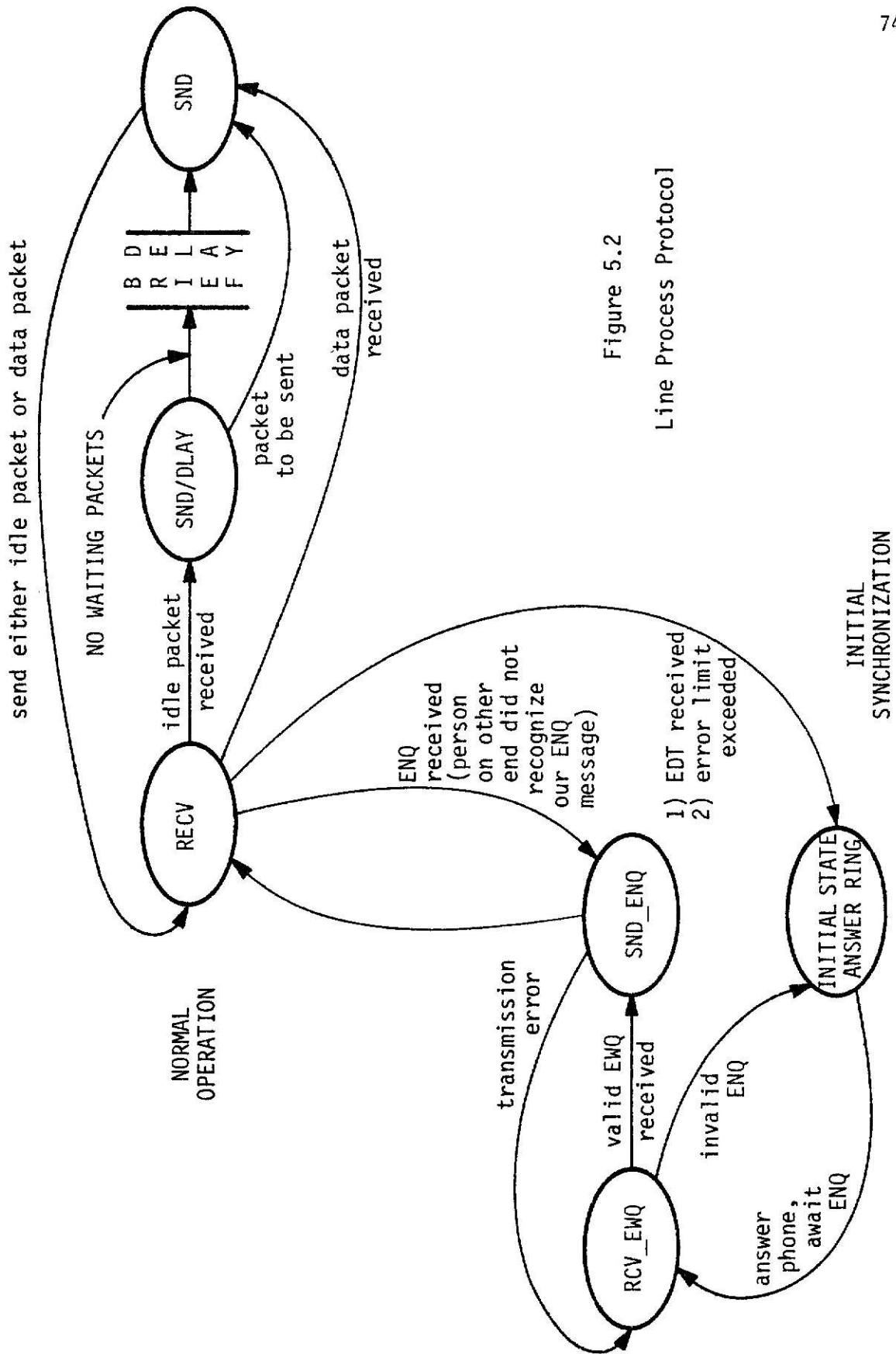


Figure 5.2
Line Process Protocol

5. PROTOCOL IMPLEMENTATION

As depicted in Figure 5.2, the half duplex line protocol serves two main functions. The first is establishing the connection of the two machines, and the second is the transmission of data across the line.

The establishment of the connection protocol requires that both communicating processes receive the other process's ENQ identification, and that that identification contain a machine and cluster which is valid when looked up in the route table. The process will continue to cycle in the sending of ENQ's and the receiving of ENQ's until a given process has both received an ENQ and is no longer receiving ENQ's from its partner. If a process has sent an ENQ and has received a valid ENQ message, then the receive state is entered. Receipt of an ENQ at this time will force reversion to the connection sequence.

Once in the normal mode of operation, reversion to the initial sequence can be caused by either receipt of an EOT (end of transmission) character, errors on the line, or receipt of an ENQ. The latter indicates that, although status on this end is ok, something has happened at the other end of the line, and that process needs to re-establish synchronization.

Within the normal mode of operation, there are two paths commonly taken in the transmission of data. If there is no information to be sent, and the last packet received

was an idle packet, then the sending process is allowed to delay before sending a packet. After delaying, a check should be made for information to be transferred, and if none is available then an idle packet is sent to turn the line around. Thus, the line need never be contended for, since the state always has one process in a receive state and the other in one of the two sending states. This leads to overhead in the idle state while sending idle messages, but much less overhead (than with Bisync) when the data flow is heavier. As mentioned earlier, we felt this trade-off to be well worth while, and especially if one assumes that data load is indicative of processor load (or machine load) in general. If the network is being used very little, then quite possibly the machine is not currently busy, and the overhead of sending idle packets is quite easily absorbed.

6.1 PAST EXPERIENCE

This project was almost an exact follow on to an earlier project which attempted to link an IBM 370 to a Data General Nova. The IBM part of this previous project was the masters work for Sheldon Fox, and the Nova part was attempted by Richard McBride and myself. It is significant to note that, although the IBM half of the connection functioned when tested with itself, the processor to processor link never became operational. This was largely due to two factors, and the operational nature of the IBM link was a perfect example of one of the laws expressed in the introduction.

The implementation on the IBM was the result of an effort by a very accomplished assembly language programmer. It was his abilities which enabled the design to be implemented, and was his leaving which made the system difficult to maintain. This is a viewpoint expressed in the introduction when discussing methods of implementing high-level designs in an processor efficient manner. The success and maintainability of the system is directly tied to the ability of the assembler programmer (and teams of programmers often just get in each others way if they attempt to jointly code individual modules).

The failure of the Nova implementation was a corollary to this. The project had been started by Lee Allen, and then taken over by two others. The only documentation available was the comments in the margin of the assembler

listings, and they were more cryptic than helpful. Again, maintainability of assembler programs is extremely difficult. The other major shortcoming of the previous implementation effort presented itself when testing of the system began. It was soon discovered that concepts of the connection protocol differed on each end of the communications line. Since no hard documentation seemed evident (and certainly was not current), development of the protocol was of a trial and error nature. But most important, the only documentation at the end of the project would have been the assembler programs and the report that Mr. Fox had written. Unfortunately, documentation prepared after the project is done is the result of recall, and this is often less than complete. Contrast this with the situation where the documentation can be taken directly from the program listings (with some modicum of support from the original functional design documents).

A third shortcoming of the previous project was the inability to obtain synchronous software for the Nova end of the project. This did not lead directly to the failure of the Nova end of the connection, but was an additional frustration in addition to the documentation problems listed above. Compared to this previous attempt, the current MIMICS implementation was a rousing success. It currently runs on the Interdata 8/32 and the Interdata 7/32 owned by the Dept. of Computer Science, the ITEL AS/5 owned by the

KSU Computing Center, and a limited subset has been implemented on the PDP 11/70 by Cullinane Corp. for the Army. Considerably more successful than the previous attempts.

As a sidelight, this previous implementation was the reason for the half-duplex line protocol being so well defined very early in the project. This had been a major effort in the previous effort, and considerable credit should be given to those involved in that project. Also, the concept of the window monitor (a new concept in this implementation) was a favorable factor in Cullinane's decision to implement our design.

6.2 APPRAISAL OF CPASCAL

As outlined in the introduction, the availability of CPascal as a language which provided both an up-to-date design as well as an implementation vehicle seems justified purely on the basis of maintainability. However, it would be remiss not to address other aspects of the language problem. One of the primary reasons for using assembler languages is their efficiency. The efficiency of Concurrent Pascal in our implementation was even poorer than our expectations, and a constant source of amazement. As stated earlier, the compilers used generated an intermediate language which was then interpreted at run time. We were

not expecting this environment to provide efficient execution, but were not prepared for the apparent slowness of the system. Current work with a new set of compilers has shown improvement of 4-5 times faster execution with compilers which generate the target (native) language of the host machine (with minor impacts on the portability of the system). No exact timings are given, since it was not possible to completely load the MIMICS message system to measure the processor load. It was obvious from watching the wait light on the two Interdata machines, however, that the message system was occupying a significant amount of processor time. When generating trace to a disc file, a figure of .5 seconds per packet seems an accurate guess.

However, it was not merely the implementation of the language which led to the poor performance figures. Most testing was done with a very light load (no more than two users at each end), and thus led to a lot of waiting in the message system. If one recalls the line process, the idle state allows a two second wait before turning the line around to see if anything is available. Since we were testing with light loads, we were always encountering these time delays; and the line process is only presented as an example. In general, many things were driven by the expiration of time intervals rather than the occurrence of specific events. This made it extremely difficult to instrument the system, as well as making it difficult to

measure the processor required for a given load. The time-out driven nature of the system may also have contributed to the performance already mentioned. It has not been decided whether the time-out driven nature of the implementation was a design deficiency on the part of those of us at Kansas State, a feature of the Concurrent language which only allows a given process to await the occurrence of a single item (necessitating a time driven auditor process to determine if a process has been waiting too long), or a failure on our part to adapt our assembly language thinking to a new means of performing concurrent programming.

A few other shortcomings of the language were primarily implementation restrictions. Among these are the need to have strings with an even number of characters, and the restriction that characters could only assume the values from 0 through 127. This made it extremely difficult to place a sequence number of 255 in a byte of the message (there is no byte type). This character restriction was eventually relaxed to allow full 8 bit values, but the notion of a "set" of characters was then invalid, since sets can only contain 127 elements. Also, availability of an "else" mechanism in a case statement would have proved handy, and has in fact been implemented in more recent compiler revisions.

A final factor which led to poor performance and cumbersome programming was the inability to pass a

capability to access a packet without expressly copying the entire packet. From the efficiency side, this meant reception of a packet incurred a copy from the Line Process to the Window Monitor, a copy from the Window Monitor to the Packet Buffer Monitor. When requested by the Message System processes, further copies occurred from the Packet Buffer Monitor to the calling process, from the calling process to the "route table monitor(or others in the message system)", and so on ad-nauseam. This physical memory copying certainly led to efficiency degradation as well as cumbersomeness when programming. Some research has been done with a concept of Managers intended to alleviate some of the above listed problems.

In spite of the seemingly endless list of criticisms, my impression of the language is quite favorable. Almost everyone can program comfortably in a language if they have been using it for any length of time, but for those of us using Concurrent Pascal it was a new language. Even so, it took a very short time to become comfortable with the language, and there were very few nuances (if any) to the language which caused problems. Particularly, the concept of Monitors allowed the coding of routines which governed sharing of data structures in a straight forward manner. Anyone who has coded program where multiple entities share data structures can appreciate the simplicity of writing the sequential steps to modify the data structure without

worrying about when to make the program uninterruptible, when to enable interrupts, and how to minimize the duration of the disarmament. The monitor in Concurrent Pascal accomplishes all of this in the kernel, and does not require the disabling of real interrupts at all.

The final feature of Concurrent Pascal was the modularity which was introduced. This was partly a side effect of the monitor concept, and partly a conscious design effort to retain each functional entity as a separate program entity. Of course, you need a non-sequential programming language to support this goal of functional program integrity.

As a summary of my opinion of Concurrent Pascal as a language for design and implementation of programs to implement concurrent operations is an unqualified success purely on the basis of ease of design and maintainability of the programs. The efficiency of the programs generated remains to be seen, but as mentioned in Chapter 1, processor efficiency may not be of real importance. The language is extremely well suited to applications where several concurrent operations take place. The programmer no longer needs to force this concurrency into a sequential program; this leads to very modular programs; and this is a desirable result of good software engineering. Personnel at Kansas State University have found that Pascal is a very easy and robust language for programming (not really even missing

better I/O capabilities), and prefer it to the more common COBOL, FORTRAN, and PL/1.

6.3 ENHANCEMENTS and IMPROVEMENTS

Enhancements and improvements to the MIMICS system seem to fall into two categories. The first category includes those things to be done to understand exactly what capabilities are implemented relative to those capabilities desired, what the machine requirements are to implement these capabilities, and why the machine requirements are what they are. The second category includes extension of the MIMICS concepts to make the entire system more usable.

In the category of usability, valid comparison can be made to the days of the first disc operating systems. The capability of disc storage was there, but the user had to be very involved with the actual capability (even to the point of knowing track addresses) in order to use the facility. Later, file management was added, and the user no longer needed to be so intimately involved with the actual machinations of the disc itself. Disc storage became much more usable as a result. The MIMICS system is much like these early disc systems. The capability to communicate among heterogeneous processors is definitely available, but the user is still required to remain intimately involved with the detailed operation of the system. What is needed

now is an extension to the MIMICS capabilities just as the file managers were an extension to the disc capabilities. It is unclear exactly what capabilities should be provided by this network manager (the name used at Kansas State is Network Resource Controller "NRC") is unclear, but it should certainly attempt to hide the topology of the network from the user, and relieve the user of knowing the exact name of the processor and task with which communications is desired just as the file manager relieved the user task from remembering sector addresses. Names of tasks should be meaningful to the user and the NRC should maintain a catalog of those names which have been declared and a mapping from catalogued names to Cluster Machine Task Port names. It is also likely that all of the NRC's which make up the network would need to pass bookkeeping information back and the forth, but the nature of this information is unclear.

Another area needing additional research to make the system more usable is reconfiguration of the 'logical' line. As mentioned in Chapters 4 & 5, physical lines can readily be added to or deleted from the logical line, but no capability is provided for rerouting traffic over a different logical line if that becomes necessary. Recognition of this need resulted in the designation of primary and secondary routes in the route table, but this only reroutes arriving packets. The problem of how to reroute packets already in the Packet Buffer Monitor or even

lower in the Window Monitor is much more complicated. Since sequence numbers are generated by the respective Window Monitors, rerouting of packets which have already been sent once and guaranteeing that they have not already been received is not easy. Rerouting of packets in the Packet Buffer Monitor is easier, assuming they have not been accessed by the Window Monitor. This is probably a fairly major shortcoming in a network of any size, since a logical line will often consist of only one physical line, and the failure of that physical link seems probable. The same capabilities are required in order to bypass a failed processor which had been performing a store and forward function, or in order to attempt to dynamically equalize loads by bypassing a loaded machine.

Another area of investigation is determining exactly what and how the MIMICS system operates, and especially how Concurrent Pascal has impacted these answers. Several areas of concern have already been voiced as concerns response times, although these have largely been the result of timing response in a distributed data base test. In many instances, the implementation allows the system to await some timeout awaiting some intervening occurrence rather than being driven by the event occurrence itself. Work needs to be done to determine whether this method of implementation (which differs significantly from the typical assembler programming scheme), is a result of the language.

If it is, then is it a desirable result, one we can live with, or is it the root of all the performance problems.

Another aspect of interest is the size of the resulting system. The CPascal version requires approximately 70 K, and the requirements for the ITEL version are similar with the addition of considerable operator control, diagnostics, and statistics incorporated into the system. Thought should be given to whether or not the processor and memory requirements are justifiable in light of the capabilities provided, or whether certain functional capabilities ought to be either deleted or only conditionally included. It is not known what the processor or memory requirements are for a limited subset

It is probably significant to notice that much of this discussion is a revisitation of the design criteria. It was decided that the priority of maintainability of a system was more important than either processor requirements or memory requirements. What is being suggested is merely an evaluation of whether or not these priorities are valid. In the opinion of this author, they are.

A final area for extension is the instrumentation of the MIMICS system itself, thus allowing it to be easily used for further research. This would include capabilities to gather statistics about its internal data structures which might aid researchers in distributed processing areas (particular distributed data bases). Statistics such as packet queue

lengths, transmission delays, and traffic levels might aid researchers in determining the validity of current simulation. These same statistics might also prove handy for system users to tune their system for better performance, a capability often missing from software systems.

6.4 CONCLUSIONS

My reaction to the entire project in retrospect is very positive. in spite of some of the problems involved in getting the packet switch and ITAM running. The only disturbing point in the process has been my negligence in finishing this report.

Through the course of the design and implementation of the MIMICS Packet Switch, I have learned a considerable amount of information about all of the following, and I feel few Master's projects offer this breadth of exposure:

- Communications line protocols and particularly IBM's Bisync, as well as knowledge of exactly what the computer must do to control the communications line.
- Operating system internals, particularly the implementation of SVC's and the means by which the modern OS controls the state of a task while the task awaits some system service.
- Interdata's Telecommunications Access Method, since it didn't work as the documentation led me to believe. For those who have not had the opportunity, following the execution of a non-trivial device driver by single stepping the processor is always educational.
- Exposure to the problems inherent in writing a program consisting of several interrelated

concurrent processing activities. Utilization of Concurrent pascal provided an excellent comparison to earlier problems in a sequential language (assembler).

- Perhaps most important. an exposure to the programming language Pascal. According to many, this is the language of the future and exposure in this depth is bound to be beneficial.
- Opportunity to work with Robert Young, an educational experience in and of itself.

1. Neal, D. and Wallentine, V.E. Experience in Porting Concurrent PASCAL. KSU Department of Computer Science, Technical Report CS 77-9, June 1977.
2. Brinch Hansen, P. The SOLO Operating System. Software Practice and Experience, Vol. 6, No. 2, April - June 1976, pp. 141-206.
3. Kimbleton, Stephen R. and G. Michael Schneider "Computer Communication Networks: Approaches, Objectives, and Performance Considerations". Computing Surveys, Vol. 7, No. 3, Sept. 1976.
4. Wallentine, V.E., Hankley, W.J., Anderson, G.G., Calhoun, M.A. and Maryanski, F.J. Progress Report in Functionally Distributed Computer Systems Development: Software and Systems Structure. KSU Department of Computer Science, Technical Report CS 77-4, January 1977.
5. Brinch Hansen, P. Network: A Multiprocessor Program. IEEE Computer Software and Applications Conference. Chicago, Illinois, November 1977.
6. Rehme, E. and Wallentine, V.E. MIMICS (Asynchronous) Control Line Protocol. KSU Department of Computer Science, Technical Report CS 77-15, December 1977.
7. INTERDATA, INC. OS-32/MT Program Reference Manual. Pub. No. B29-390R02.
8. INTERDATA, INC. Interdata Telecommunications Access Method (ITAM/32) Reference Manual. Pub. No. 29-541R01.
9. Campbell, J.H., Wright, C.T. and Wallentine, V.E. A Pedagogical Operating System. ACM SIGPLAN Symposium on Pedagogic Languages with Small Computers, January 1972.
10. Maryanski, F.J., Fisher, P.S., Calhoun, M.A., Sernovitz, L. and Wallentine, V.E. A Mini-computer Based Distributed Data Base System. Proceedings NBS-IEEE Trends and Applications Conference--Mini-and Micro-Computer Systems, Washington, D.C., June 1976.
11. Brinch Hansen, P. "The Architecture of Concurrent Programs". Prentice-Hall, Englewood Cliffs, N.J., 1977.
12. Silberschantz et al. Extending Concurrent PASCAL to Allow Dynamic Resource Management. IEEE Transactions on Software Engineering, Vol SE-3, No. 3, May 1977.

13. Cullinane, J., R. Goldman, T. Meurer, and R. Navarawa "Commercial data management processor study". Cullinane Corp., Wellesley, Mass. Dec. 1975.
14. Housh, R.D. "An implementation of a distributed data base management system", M.S. Report, Computer Science Dept., Kansas State University, Manhattan, Kans., 1978.
15. Thompson, K. and Ritchie, D.M. The UNIX Time-sharing System. CACM, Vol. 17, No. 7, July 1974, pp. 365-375.
16. Honeywell Information Systems, Inc. File Management Supervisor. Pub. No. DB54, 1972.
17. International Business Machines, Inc. General Information - Binary Synchronous Communications. Order No. GA27-3004-1.
18. Mounday, D. Porting CPASCAL to the NCR 8250. Kansas State University, Computer Science Department, Master's Report, 1978.
19. Thompson, K. and Ritchie, D.M. UNIX Programmer's Manual. Bell Telephone Laboratories, Inc., May 1975.
20. Neal, D.N., Anderson, G.G., Ratliff, J. and Wallentine, V.E. KSU Implementation of Concurrent PASCAL - A Reference Manual. KSU Department of Computer Science, Technical Report CS 76-16, June 1976.
21. Cottel, D. and Zaun, J. Concurrent PASCAL (For Interdata 7/16 Computer). Naval Underseas Laboratory, December 1976.
22. Kessels, J.L.W. An Alternative to Event Queues for Synchronization in Monitors. CACM, Vol. 20, No. 7, July 1977, pp. 500-503.
23. Forsdick, A.C., Schantz, R.E. and Thomas, R.H. Operating Systems for Computer Networks. Computer, Vol. 11, No. 1, January 1978, pp. 48-57.

IMPLEMENTATION OF MIMICS PACKET SWITCH

by

JAMES R. RATLIFF

B.S., Kansas State University, 1975

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas
1978

ABSTRACT

This report describes an implementation of the MIMICS packet switch developed at Kansas State University. MIMICS is a network of heterogeneous Mini and MICro computer systems. It is intended that the system be adaptable to a growing environment via offloading of function from the host to a mini or micro processor, and via distribution of network functions among mini and micro processors. This report discusses both the design, and the choice of an implementation language, since the latter significantly affects the portability and distributibility of the software system. Although this report is directly concerned with the Packet Switch portion of the system, many of the remarks are equally valid with the Message System and Cluster Data Movement sections of the entire system.