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

TABLE OF CONTENTS

LIST OF FIGURES

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]).

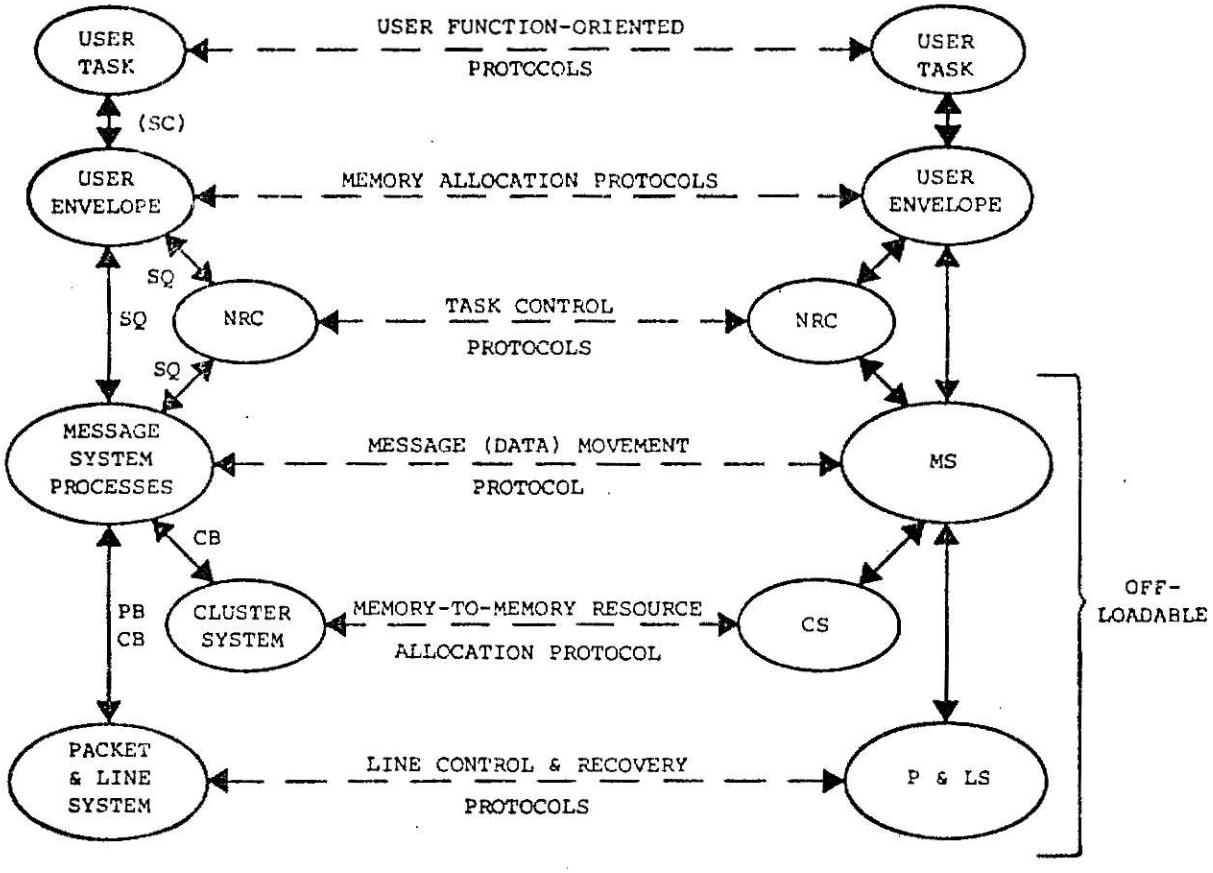
MIMICS is divided into three functionally distinct elements. The Message System (MS) accepts requests directly from the user via the local operating system. The Message System performs user -to -user coordination and flow control at the message level. The Packet System (PS) accepts messages which have 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 langauge 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 programmmer 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 up on 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 inplementation 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 if 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 bisyncronous 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

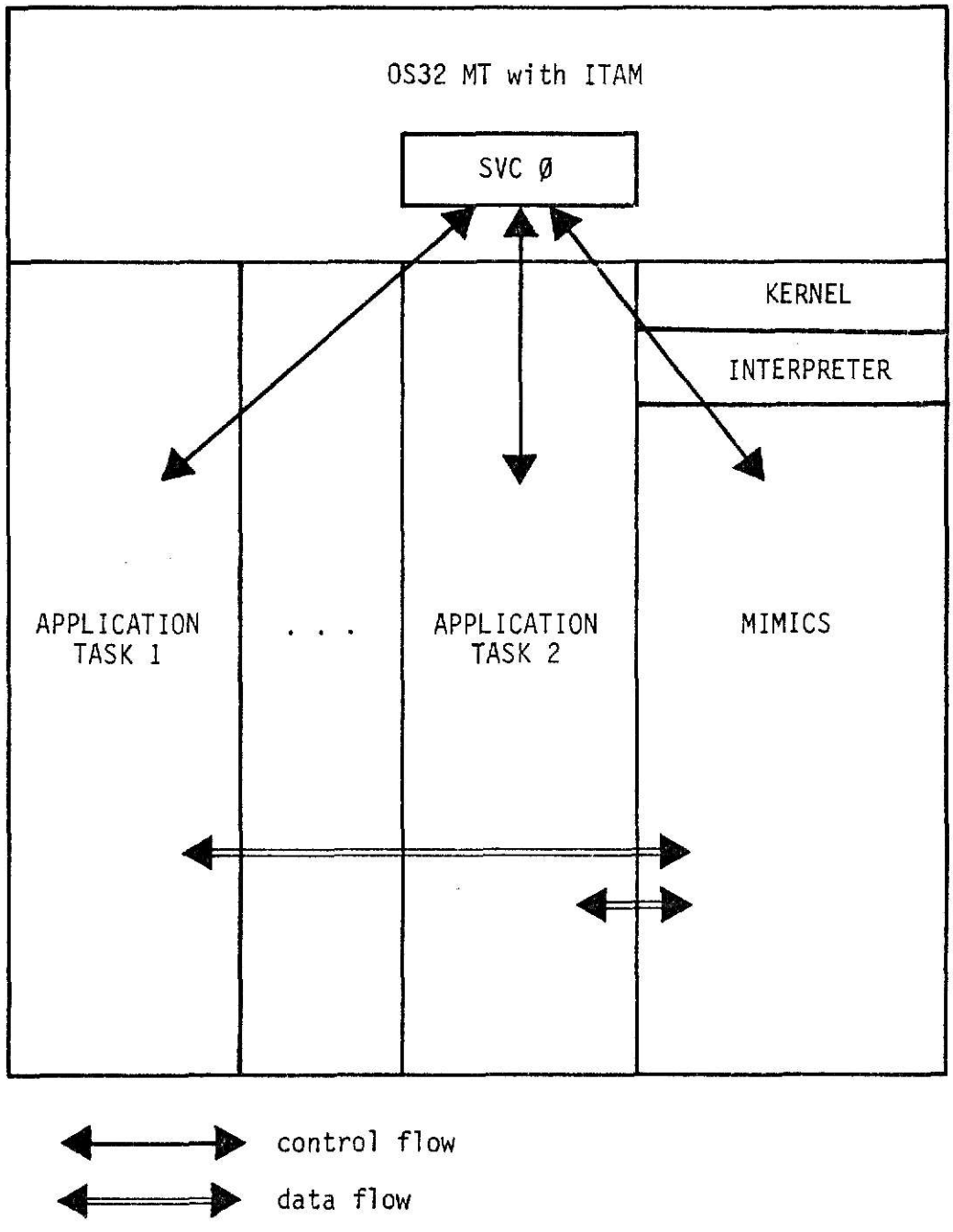


Figure 1.2

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

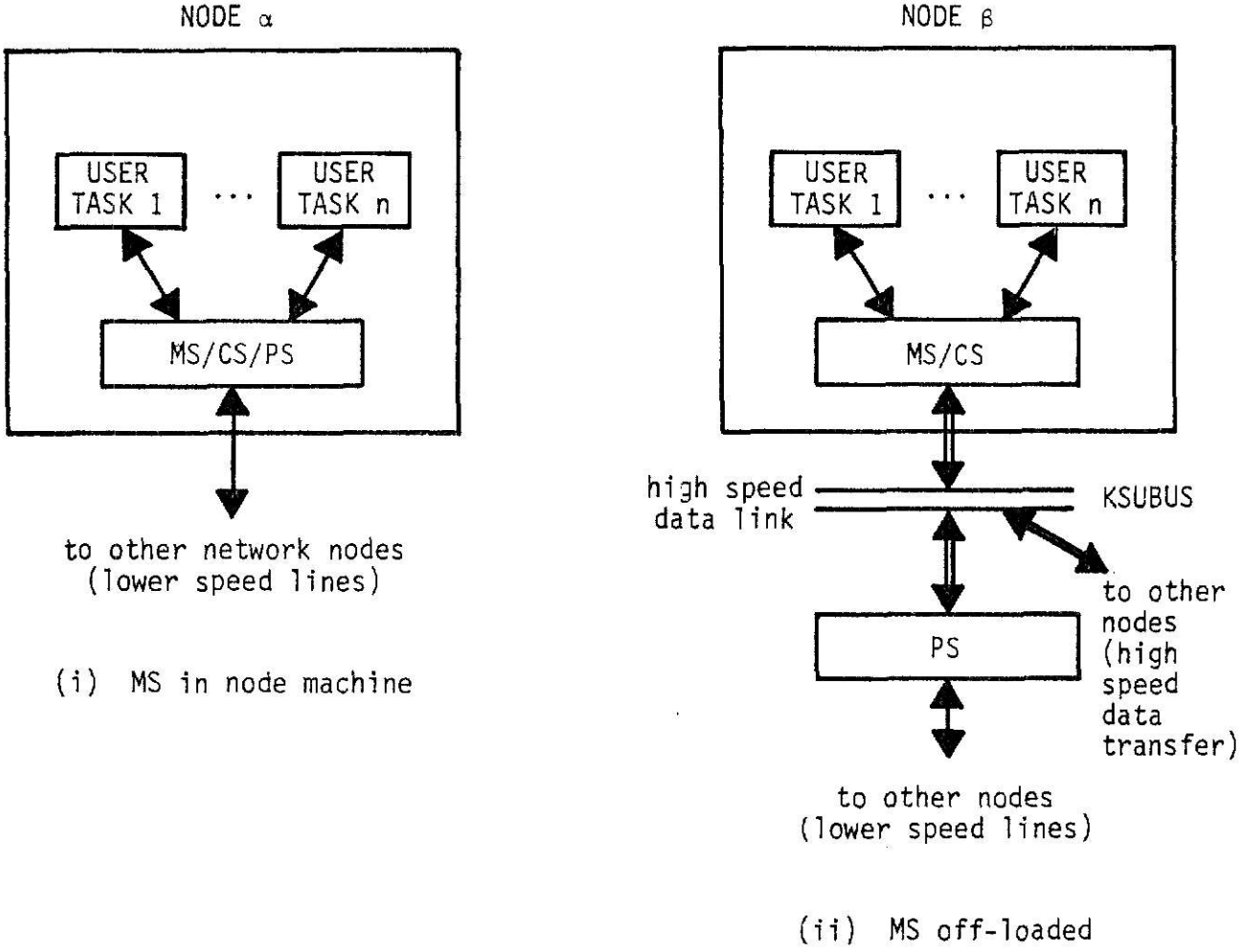


Figure 1.3

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 communciations to the communcations 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 dummying 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