*This blank page was inserted to preserve pagination.*

# INTERACTIVE DEBUGGING
# IN A DISTRIBUTED
# COMPUTATIONAL ENVIRONMENT

ROBERT DAVID SCHIFFENBAUER

September, 1981

© Robert David Schiffenbauer 1981

The author hereby grants to M.I.T. permission to reproduce and to distribute copies of this thesis document in whole or in part.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

*This empty page was substituted for a blank page in the original document.*

## ACKNOWLEDGEMENTS

I wish to thank my thesis advisor, Professor David Reed, for reading and commenting on early drafts of this thesis, as well as for a number of suggestions and ideas that improved the final product.

I wish to thank Bob Baldwin for contributing so much time and effort towards improving this research at a time when he was busily involved in his own thesis work. Our conversations were always informative and enlightening and his enthusiasm for the project never faltered. His help improved this work in many, many ways.

I would also like to thank Brad Myers and Andy Mendelsohn for their aid in dissecting various portions of the Alto/Mesa system, as well as for various illuminating discussions. I thank Dan Theriault for his Mesa implementation of a linked list data abstraction, of which my debugging facility made extensive use. Thanks to officemates Karen Sollins and Stephen Kent for all their help. Thanks to Professor Jerry Saltzer and Dr. David Clark for initially introducing me to the Computer Systems Research Group at M.I.T.

Finally, I would like to thank my parents and my brother, Joel, for all their help and encouragement.

*This empty page was substituted for a blank page in the original document.*

# INTERACTIVE DEBUGGING

# IN A DISTRIBUTED COMPUTATIONAL ENVIRONMENT

by

ROBERT DAVID SCHIFFENBAUER

Submitted to the Department of Electrical Engineering and Computer Science on August 7, 1981 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science

## ABSTRACT

This thesis describes an implementation of a facility for interactively debugging distributed programs. These distributed programs consist of groups of cooperating processes concurrently executing on an arbitrarily extensive network of processors. The facility allows the user to monitor and control, at his leisure, the interprocess communications that occur through message passing while execution of the distributed program proceeds. It presents the user with the ability to simulate transmission errors and delays, to alter and create packets, and to precisely control the pattern of such communications. The facility serves as a tool for the detection of lurking bugs, those errors, peculiar to parallel processing, which may or may not appear during the course of any particular execution.

The facility possesses a high degree of transparency towards the program being debugged. That is, it has a minimal effect on the events that define the execution of that program. Transparency is a desirable property for any debugger to possess. To achieve such transparency, the processes of the distributed program are made to execute in a logical time environment, reading logical, rather than physical, clocks.

We show that the facility obeys a clock condition, with which any logical time system must comply in order to be correct. We also show that the facility actually simulates the program it is being used to debug. Finally, we show that the facility simulates a particular computation of the program that is likely to occur. The notion of probable simulation is defined, and our debugging facility is shown to achieve it.

Key Words: distributed systems, debugging, monitoring, reproducibility, lurking bugs

*This empty page was substituted for a blank page in the original document.*

CONTENTS

Acknowledgements ..... 2

Abstract..... 3

Table of Contents..... 4

Chapter One. Introduction..... 6

1.1 Distributed Systems..... 7
1.2 Distributed Programs..... 10
1.3 Debugging, Monitoring and Transparency..... 13
1.4 Previous Work..... 18
1.5 Hardware Environment for this Project..... 22
1.6 Software Environment for this Project..... 24
1.7 The Internet Protocol..... 28
1.8 Plan of Thesis..... 32
1.9 Some Definitions..... 33

Chapter Two. Issues in the Design of a Debugging Facility..... 34

2.1 Use of the Debugging Facility..... 36
2.2 Practical Considerations: Transparency and Artificially Induced Communication Delays..... 45
2.3 Theoretical Basis: Causality and Systems of Logical Clocks..... 57
2.4 The Uncertainty Principle of Program Debugging.. 69

Chapter Three. Implementation of the Debugging Facility..... 70

3.1 Overview of the Facility..... 71
3.1.1 The Central Site..... 73
3.1.2 The Nub..... 75
3.2 Routing and Timestamping of Application Packets. 78
3.3 Nub - Central Site Interactions..... 87
3.3.1 Initialization Packets..... 87
3.3.2 Handler-Creation Packets..... 87
3.3.3 Receive-Request and Maybe-Receive-Request Packets..... 89
3.3.4 Conditional-Execute Packets..... 90
3.3.5 Give-Me-Now Packets..... 90
3.3.6 Cannot-Be-Satisfied Packets..... 91
3.3.7 Clock-Update Packets..... 92

3.3.8 Package-Destroyed Packets..... 93
3.3.9 Enter-Debugger Packets..... 94
3.3.10 Ack Packets..... 95
3.4 Low Level Mechanisms..... 96
3.4.1 Initialization..... 96
3.4.2 Application Packet Selection Algorithm.. 101
3.4.3 Node Suspension and Logical Clock Maintenance..... 109
3.4.4 Deadlocks..... 114
3.4.5 Termination..... 118
3.5 User Interface..... 120
3.5.1 Monitoring..... 120
3.5.2 Debugging (User Commands)..... 121
3.5.2.1 The Send Command..... 122
3.5.2.2 The Withhold Command..... 122
3.5.2.3 The Replace and Retrieve Commands..... 123
3.5.2.4 The Delay Command..... 124
3.5.2.5 The Display Command..... 125
3.5.2.6 The Create Command..... 126
3.5.2.7 The Call Debugger Command..... 126
3.5.2.8 The Quit Command..... 126

Chapter Four. Correctness and Usefulness of the Debugging Facility..... 127

4.1 Maintenance of the Clock Condition..... 129
4.2 Proof of Simulation..... 133
4.3 Probable Simulation..... 146
4.4 Probable Simulation vs. Transparency..... 157

Chapter Five. Related Ideas and Suggestions for Further Research..... 158

5.1 Fragmentation..... 159
5.2 Bottlenecking..... 161
5.3 Order of Event Reporting..... 163
5.4 The Multi-Application Problem..... 165
5.5 Controlling Monitor Entries..... 169
5.6 Future User Interface..... 175
5.6.1 Multistepping and Slow Stepping..... 175
5.6.2 Graphical and Analogical Display of Data 177
5.6.3 Dynamic Display of Events..... 181
5.7 Towards an Integrated Debugging System for Distributed Computational Environments..... 184

References..... 186

# Chapter One

## Introduction

Computer programmers are human. They make mistakes. It is a rare program that is coded correctly the very first time it is attempted. As long as there are program "bugs", debugging tools will be needed to detect and correct them. This thesis explores the unique problems encountered in designing and implementing one such debugging tool for programs executing in a distributed computational environment.

## 1.1 Distributed Systems

Distributed computing occurs when two or more computer processors join in a cooperative venture to get a particular job done. It has been characterized (Metcalfe76) as including an entire range of computational organizations: multiprocessing, local networking, and remote networking. The properties of these systems differ in degree, rather than in kind.

In multiprocessing, processors are usually small, in proximity physically, and lack an ability to function autonomously. Programs are executed in a highly parallel fashion by meting out independent tasks to each processor for concurrent handling. Interprocessor communication is swift and frequent since, in general, all processors share the same memory.

In remote networking, processors are often large, powerful, centralized, highly autonomous computers in their own right. They may be designed and built independently, and connected together in a network as an afterthought. Remote networks may extend over many miles (for example, the Arpanet is nationwide). In these systems, interprocessor requests are usually for certain kinds of services that cannot be performed by the requesting node or for bulk information transfer. The notion of highly parallel task execution is not indigenous to this arrangement.

Local networking lies somewhere in between multiprocessing and remote networking, although it is much closer to the

latter. Local networks may extend anywhere from several yards to a few miles. Often, too, the processing power of local network processors is intermediate to those of multiprocessing or remote networking systems. Local network processors may, at times, be highly autonomous and, at other times, be highly cooperative.

In general, there are no strict dividing lines separating remote networking, local networking, and multiprocessing. Systems are often assigned to one category or another, as discussed above, on the basis of imprecise properties such as distance between processors (Metcalfe76) or degree of autonomy (Svobodova79 - this report, incidentally, provides an excellent introduction to many of the issues and problems involved in distributed computing). Thus, we say that a system is a remote network when its processors are separated by <u>about</u> ten kilometers or more, or we say that a system is not of the multiprocessing type because its processors are <u>highly</u> autonomous.

For our purposes, it is useful to classify these three systems by another method (which is no less hazy than those mentioned above). To us, the key characteristic of a distributed system is that it is impossible to appraise <u>simultaneously</u> all processors (hence, the different segments of program code executing on these various processors) of the occurrence of some particular system event. We distinguish the three system types by a value, $\Delta t$, representing the average time interval between the informing of the first processor and

the informing of the last processor of the occurrence of the event. In multiprocessing systems this value is quite small. In remote networks, this value is often quite large. In local networks, of course, the value of $\Delta t$ is intermediate to these two.

We are interested in those systems for which $\Delta t$ is significant in comparison to the time it takes to execute instructions on any processor (a system may contain processors which operate at varying speeds). Another way of saying this is that we will be concerned with systems where $\Delta t$ is significant when compared to the time interval between successive events on any processor. Multiprocessor organizations typically do not possess this characteristic. Local and remote networks do. Thus, in this thesis, we are interested mainly in the latter two types.

The utility of this particular outlook towards distributed systems will be made clear presently.

## 1.2 Distributed Programs

Many problems suitable for solution by computer are capable of being broken down into a number of smaller subtasks which can be processed independently. These types of problems lend themselves to handling by some distributed system organization. The programmer codes his solution as a set of processes, assigning each one of these to some processor in the system. A process is, "a set of events with an a priori total ordering." (Lamport78). That is, a process is a chronological sequence of events. (An event may be, for example, the execution of a single machine instruction.) A distributed system can execute a set of independent processes in parallel. Thus, distributed programming implies parallel processing.

However, the converse is not necessarily true. Parallel processing may be simulated on a single processor through some kind of interleaving mechanism whereby the processor now executes in the context of process A, now changes state to execute in the context of process B, now process C, and later, perhaps, back to process A again.

Our model of a distributed program is one which combines both genuine and pseudo parallel processing. A distributed program is considered to consist of a set of processes, partitioned into non-intersecting subsets with varying cardinality. Each such subset is assigned to a single processor in the distributed system. That processor performs

pseudo parallel processing on this subset of processes via an interleaving mechanism. Genuine parallel processing occurs between the processes residing at distinct processors (see figure 1.1). Usually, some mechanism exists to allow the various processes to communicate in a cooperative fashion.

In light of the classification discussed in the previous section, we say that all processes at the same processor can simultaneously be appraised of the occurrence of some system event. Processes residing at distinct processors, however, cannot be so appraised. In this thesis, we are interested in those systems in which $\Delta t$ is not insignificant in comparison to the time needed to perform any two consecutive events within any process belonging to the distributed program being executed.

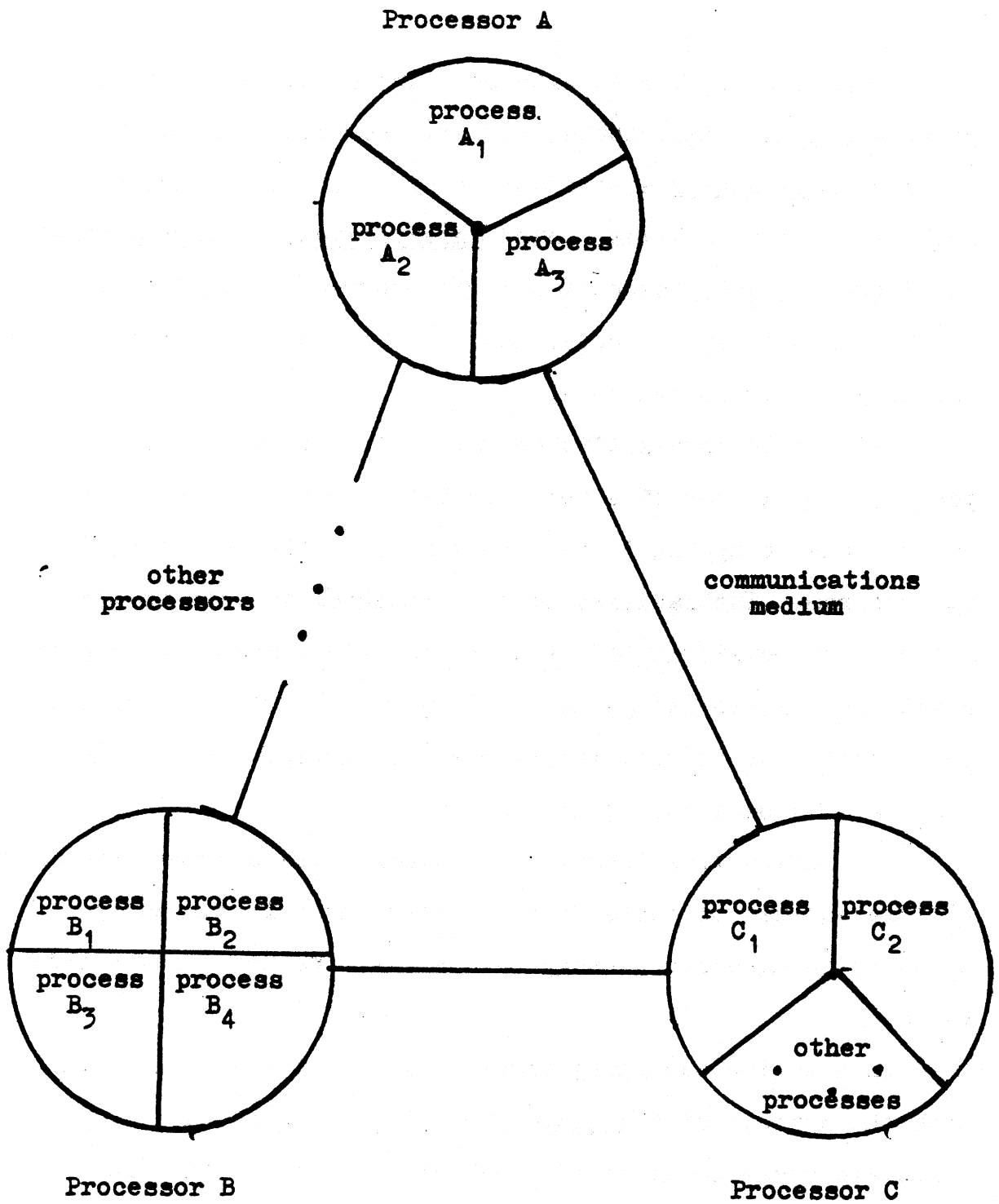

Processor A
process A1
process A2
process A3
other processors
communications medium
process B1
process B2
process B3
process B4
process C1
process C2
other processes
Processor B
Processor C

Figure 1.1

## 1.3 Debugging, Monitoring and Transparency

We will be concerned with one particular aspect of programming in a distributed system. We will examine the problem of debugging distributed programs. More precisely, we analyze the difficulties behind interactively debugging (that is, debugging while execution is in progress) such distributed programs as discussed above, and will propose a practical facility to accomplish this.

Related to interactive debugging is the concept of monitoring. A user of a debugger has no basis on which to perform his debugging if he cannot monitor the behavior of his program. The facility to be introduced in this thesis allows user monitoring of certain specific classes of program events as well as debugging of those events. These events are those having to do with interprocess communications. This will be discussed more fully in chapter two.

The interactive debugging of distributed programs requires a different set of tools from those employed in currently existing debuggers or debugging systems. We now see why this is so.

Interactive debugging almost universally depends heavily upon the concept of breakpointing. It is a somewhat fortunate characteristic of computers that they are able to perform their various functions at speeds far in excess of the speeds at which humans can keep track of what they are doing. When interactively debugging, the human user must be aware

of what has already been accomplished in order to make decisions about what is to happen next. This is done by allowing the computer to execute for a period of time and then suspending execution at a designated point in the program (the breakpoint) to allow the user to "catch up". The breakpoint concludes when the user has examined the state of the machine, has, perhaps, made various alterations in this state, and has allowed execution to recommence. Theoretically, it is the job of the debugger to insure that the state observed by the program being debugged upon execution restart is identical to the state observed at the breakpoint (with the exception, of course, of changes caused by the user). Then the fact that a breakpoint occurred will be invisible to the executing program. The debugger has made the breakpoint <u>transparent</u>.

Transparency is an extremely desirable property for a debugger to possess. We define <u>transparency</u> as being achieved by a debugger just when the events that constitute the program being debugged are identical in the presence or in the absence of the debugger (aside from user initiated alterations performed when the debugger is present). This means that the debugger, itself, does not affect the program being debugged. A lack of transparency implies that the program being debugged is not quite the one that the program writer had in mind. A lack of debugger transparency affects the behavior of the program being debugged. The less transparent the debugger, the more this behavior is affected.

Total transparency is a theoretical concept. In practice,

no debugger is completely transparent to the program it is being used to debug. A debugger only possesses a higher or lower degree of transparency towards that program.

Now, in a non-distributed system, it is relatively easy for a debugger to maintain a high degree of transparency (i.e. to accomplish highly transparent breakpointing) towards a non-distributed program. One reason for this is that it is easy to suspend simultaneously all processes making up that program.

Simultaneous suspension of all processes means that the entire program is halted at a definite instant in time, when the machine is in a definite state. It is not difficult to save this state and to restore it when all processes recommence execution simultaneously at some later instant in time. Then the processes making up this non-distributed program are unaware that any debugger induced execution break has occurred.

In a distributed system, however, such simultaneous suspension is not possible. This is because all processes cannot be appraised simultaneously of the occurrence of any system event. For example, suppose the user stipulates that a breakpoint is to occur just before the execution of statement X in process Y residing at processor Z. When this occurs, processor Z sends messages to all other processes commanding them to suspend execution. It is not possible for all processors to simultaneously receive such commands. $\Delta t$ is always greater than zero.

It is also not possible for the user to inform all

processors before execution begins that they must all suspend themselves at some future time V (even if this capability were possible, it is not clear that it would be at all useful). Completely accurate synchronization of the time of day clocks existing at each processor can never be achieved (Lamport78). Thus, each processor will read time V at a slightly different real time than any other processor. Again, $\Delta t$ will be greater than zero.

In those systems where $\Delta t$ is significant, the fact that simultaneous breakpointing is impossible to achieve means that it is very difficult to maintain a high degree of debugger transparency. The greater the $\Delta t$ value, the harder it is to maintain such transparency.

To see this, consider the program consisting of two processes, A and B, residing at distinct processors. Consider the interval, $\Delta t$, between the time that A receives a command to suspend and the time that B receives a command to suspend. This $\Delta t$ value is considered to be larger than the time it takes to execute two consecutive instructions in process B. In this interval, A is suspended while B continues to execute. If B was to receive some kind of communication from A during this interval had A not been suspended, transparency would be lost. The suspension of A by the debugger would prevent B from receiving its communication. Obviously, the greater $\Delta t$ is, the greater the probability that B was to receive a communication from A during the interval, hence, the greater the probability that the debugger would prevent this communication from taking

place leading to a loss of debugger transparency. Notice that if $\Delta t$ was not at all significant, then B would not have a chance to execute any instructions during the interval. In distributed systems with such a $\Delta t$ (highly integrated multiprocessing organizations), completely simultaneous breakpointing is nearly achievable. As a result, little transparency is lost because of this problem.

Currently existing debuggers have not been able to provide interactive service via breakpointing for distributed systems in which $\Delta t$ is significant, because they have not been able to solve this transparency problem. For these systems, a method is needed which does not depend on the simultaneous appraisal of events, the concept on which breakpointing is based. In this thesis, we present an interactive debugging facility for such systems. This facility maintains a high degree of transparency towards the distributed program being debugged. It in no way depends upon the concept of simultaneous appraisal.

## 1.4 Previous Work

A good introduction to many of the issues involved in program debugging and monitoring can be found in Model (Model79). Brief descriptions of some debugger implementations may be found in Myers (Myers80). The reader is referred to the bibliographies of those two works for in depth information on particular subjects in this field.

The earliest debuggers were suited for single process programs. As programming languages with parallel processing capabilities have come into vogue, and as computational systems have grown in complexity, tools for monitoring and debugging concurrently executing processes have arisen. COPILOT (Swinehart74) was capable of displaying information about many processes simultaneously while permitting the user to interactively issue debugging commands. DLISP (Teitelman77) is a graphics package which uses multiple windows (designated display screen areas) to facilitate the simultaneous reporting of information about various concurrent processes. Model's system possesses the multiple display capabilities of DLISP and COPILOT as well as the ability to create a history tape of the program's execution, which may be played back later at the user's leisure. It should be noted that these three facilities are tailored to uniprocessor or multiprocessor systems, or, in general, to systems in which $\Delta t$ is insignificant.

An attempt to extend a debugging tool to a local network,

the Ethernet (Metcalfe76), where $\Delta t$ is significant, may be found in the Metric system. Metric consists of three portions. "There is a probe in the user's object system, an accountant that collects information from the probe, and an analyst that processes the information and presents it in an intelligible format. Measurement events are those data that the probe transmits to the accountant, and which are subsequently processed by the analyst." (McDaniel77) The object system probe exists at each processor on which the program to be debugged is executing. The accountant and analyst reside on processors distinct from any of these. Metric is itself a distributed program.

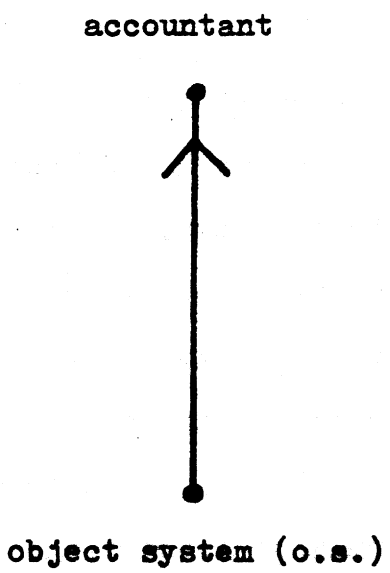
There are three arrangements of Metric (see figure 1.2). We mention these briefly in order of increasing complexity.

The Line - This consists of a single probe and a single accountant.

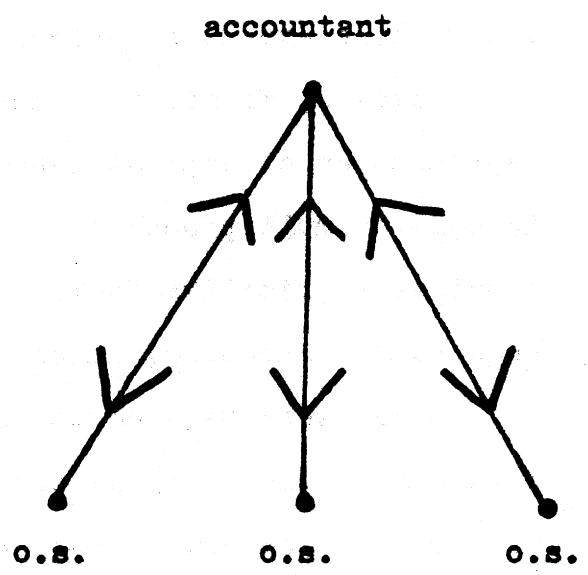
The Tree - This consists of an arbitrary number of probes simultaneously transmitting event data to a single accountant. We will see that the debugging facility we propose in this thesis is closest to this type of structure.

The Network - This consists of an arbitrary number of probes simultaneously transmitting event data to an arbitrary number of accountants, the latter perhaps operating in a cooperative fashion. We will have reason to refer back to this structure in chapter five.

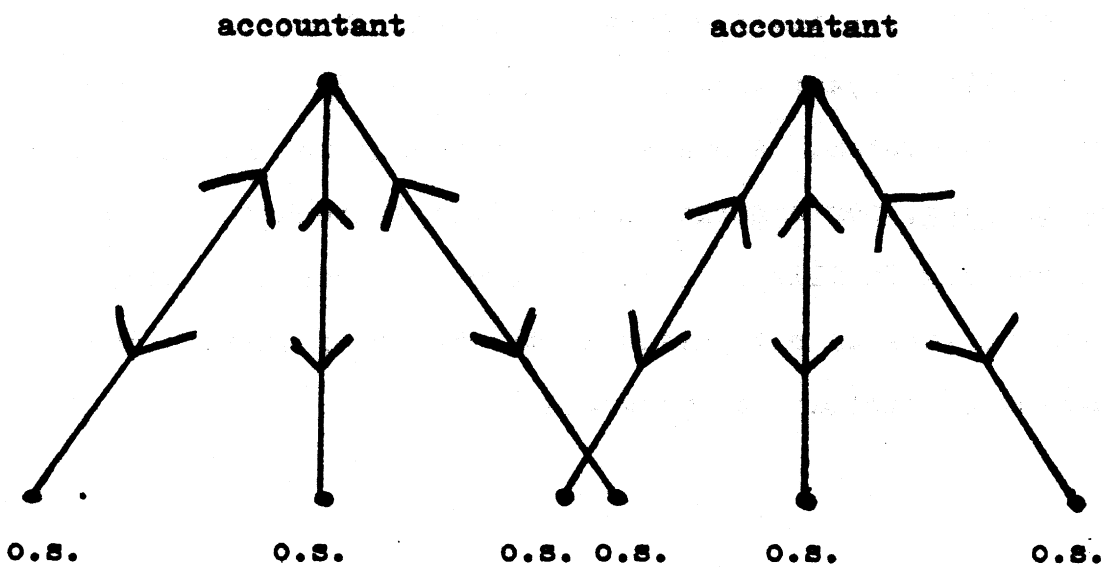
We must emphasize that Metric is not an interactive debugging facility. Thus, the fact that it operates in a system with a significant $\Delta t$ value is not really of any great import. Metric, like Model's facility, collects event reports on a history log. The user examines this log after the program




The Line




The Tree




The Network

Figure 1.2
(from (McDaniel77), fig. 3)

to be debugged has ceased execution. The user does not debug while execution proceeds.

However, Metric is important to us because it does represent an attempt at debugging programs that do not execute in uni- or multiprocessor environments. Furthermore, Metric provides a primitive facility for the detection of lurking bugs (to be discussed in chapter two) in distributed programs.

Other work related to ours, which are not strictly debugging tools, include the Virtual Machine Emulator (Canon80) and research by Bryant (Bryant77). The Emulator adopted the expedient of having programs execute in a virtual time environment, reading virtual clocks which do not "tick" in real time. Our debugging facility also makes use of a virtual time environment. We point out that our use of this concept is different from that used in the Emulator. However, we received some inspiration from that project's approach.

Finally, Bryant treats the subject of simulation in a distributed system. A number of the techniques he employed (timestamping, for example) are similar to techniques used in our debugging facility. Bryant, however, does not attempt to extend his work in simulation to the realm of interactive debugging of distributed programs.

## 1.5 Hardware Environment for this Project

Our debugging facility is implemented on the Ethernet network, a local network with a significant $\Delta t$ value. There is no reason why the same facility could not be implemented on a remote network. In other words, there is no maximum $\Delta t$ value beyond which the facility will cease to operate. However, as we move from networks with small $\Delta t$ values to those with large $\Delta t$ values, we often find a decrease in the number and importance of truly distributed applications implemented upon them. Therefore, the need for our debugging facility on many large $\Delta t$ networks may not be very great.

". . . Ethernet uses tapped coaxial cables to carry variable length digital data packets among, for example, personal minicomputers, printing facilities, large file storage devices, magnetic tape backup stations, larger central computers, and longer-haul communication equipment." (Metcalfe76) Interprocess communication between processes residing at distinct processors occurs through explicit message passing. The Ethernet hardware does not guarantee the errorless delivery of such messages. Messages arrive at their destinations only with high probability. If the program requires a probability greater than the Ethernet can provide, it must implement, in software, some packet transmission protocol to mask hardware packet loss. See Pouzin and Zimmermann (Pouzin78) for an introduction to packet transmission protocols.

Each processor on the network is an Alto desktop personal computer (Xerox79a). This is a minicomputer containing 64K 16-bit memory words, one or two 2.5 Mbyte removable cartridge disks, a sophisticated 875 line display screen, and an interface to the Ethernet. Each Alto is capable of operating in a stand alone mode, or in cooperation with various other machines on the network.

## 1.6 Software Environment for this Project

The software environment for our debugging facility is provided by the Alto/Mesa system (Xerox79b). The programming language used in this system, Mesa, is a Pascal-like language which permits concurrent execution of multiple processes (Mitchell79). Our facility has been implemented in Mesa. In this section, we discuss some of the important concurrency features of Mesa and the Alto/Mesa system (see Lampson and Redell (Lampson80) for more detail). We do this in order for the reader to be able to appreciate some of the implementation details in later chapters.

Mesa allows the creation of a new process to be accomplished via the FORK call. A previously existing process may fork any number of new processes to execute in parallel via the interleaving mechanism of the Alto processor. Forked processes are deleted via a JOIN statement. The JOIN statement permits the joining process to retrieve whatever results have been computed by the joined process. The system then destroys the joined process. Forked processes which do not compute explicit results may be <u>detached</u>. Detached processes are never joined.

Thus a usual paradigm is for some process to fork another, execute some code independently of the forked process, and attempt to rejoin the forked process at some later time to retrieve its results.

This mechanism may at times be too restrictive. Processes often need to interact in a more highly sophisticated manner

than the fork-join apparatus allows. Thus Mesa possesses a monitor mechanism (Hoare74) which allows processes to have synchronized access to shared data in memory through explicit procedure calls. Synchronization is achieved through mutual exclusion by the use of a monitor lock which must be acquired before the process may enter that monitor and access its protected data. When a process acquires the monitor lock it effectively shuts out all other processes from that monitor. The process may then access the data without worrying about concurrent access by some other process. Many interprocess timing difficulties are solved in this fashion. When the process is finished with the monitor data it releases the monitor lock, and any other process may then acquire it. It is obvious that only processes residing at the same processor (Alto) may interact through the monitor mechanism since such interaction is achieved through shared memory.

Implicit in the monitor mechanism is the notion of a monitor invariant. The invariant is "an assertion defining what constitutes a 'good state' of the data for that particular monitor." (Mitchell79). This invariant must be true whenever a process acquires the monitor lock and is about to access the state of the monitor data. A process inside a monitor can make the invariant false, if it pleases, but must restore the invariant before it relinquishes the lock. Thus, when a process acquires a monitor lock it may see any of a range of states, all of which satisfy the monitor invariant.

At times, a process may enter a monitor and find that,

although the invariant is satisfied, the state is such that it cannot proceed. It must WAIT for some other process to enter the monitor and satisfy whatever condition it requires. The process waits on a condition variable and releases the monitor lock (after, of course, restoring the invariant) until some later time when the condition is satisfied. Eventually, perhaps, another process will come along to satisfy the condition being awaited. This new process will NOTIFY the waiting process that the condition has been satisfied. The latter may attempt to reacquire the monitor lock at some future time and continue execution from the point where it left off. If a notify occurs on a condition on which no process is currently waiting, that notify is simply discarded.

Occasionally, it happens that a process decides it has been waiting too long to receive a notify. A timeout value is assigned to each condition variable specifying the maximum amount of time that a process should wait on it before it "wakes up" of its own accord. Processes may time out when some failure occurs in the communications mechanism or simply when no other process has been able to satisfy the condition in a reasonable amount of time. Timeouts may be disabled for a particular condition variable. In that case, a process waiting on that condition will never wake up by itself. To resume execution, it must be notified by some other process.

Processes acquire the processor for execution by first joining a ready list (this "join", of course, has nothing at all to do with the "join" discussed above). The ready list is

a linked list of process state blocks (PSBs) which represent various important information about each process. When a PSB reaches the front of the ready list, the process it represents is eligible for execution by the processor. Generally, PSBs join and exit the ready list in a first in - first out order. However, certain processes may be assigned a higher priority than others. High priority processes have their PSBs placed on the ready list ahead of the PSBs of all low priority processes. In fact, a high priority process will preempt a low priority process that is currently in execution. After the high priority process has relinquished the processor, the preempted process is able to reacquire it right away without having to go back to the end of the ready list.

Each PSB contains a priority field indicating the priority of the process it represents. It also contains a timeout field indicating the time at which the process it represents will timeout (based on a hardware timeout clock) if it is currently waiting on a condition variable. If this field is zero, and the process is currently waiting on a condition variable, then that condition variable has had its timeout disabled.

Processes control the processor until they conclude their execution, until they are forced to wait, until they attempt to enter a locked monitor, or until they are interrupted. There is no attempt by the processor to implement a fair scheduling policy among the various processes. Occasionally, a process that has been executing too long will voluntarily yield the processor to other processes of equal priority.

## 1.7 The Internet Protocol

Our debugging facility is implemented on top of the Internet Protocol (ISI80). This protocol allows interprocess communication to take place via explicit packet transmission. These packets, or datagrams, may be received by the Internet Protocol from higher level protocols (TCP, for example) and are, in turn, handed down to the hardware for actual transmission over the Ethernet. The Internet Protocol merely provides for datagram transmission across the network. "There are no mechanisms to promote data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols." (ISI80)

A datagram receives an internet header in order to facilitate its transmission. This header includes a number of fields worth mentioning here:

> Source Address - The 32 bit internet address of the processor at which the datagram was created. Some process at that site was responsible for creating this datagram.
>
> Destination Address - The 32 bit internet address of the processor to which the datagram is to be sent. Some process at that site will accept this datagram.
>
> Identification - A 16 bit value assigned by the sending process that distinguishes this datagram from any other created at that site.
>
> Protocol - An 8 bit value indicating the "type" of the datagram. This field is used to determine what process the datagram should be routed to at the destination processor.

The particular implementation of the Internet Protocol which we have used was implemented by Robert W. Baldwin at MIT.

We briefly discuss how this implementation is used by processes to transmit and receive packets over the Ethernet. We describe this here because these ideas will prove necessary for a full understanding of the implementation of the debugging facility to be described in chapter three.

In order for any process at a processor to make use of the Internet Protocol, some process residing there had to have issued a create-internet-package command. This initializes various parameters necessary for communication. After this, any process may assemble a packet for transmission by interfacing with various internet procedures. When the packet is to actually be transmitted, the process calls the internet Send procedure. At this point, the packet is made ready for Ethernet transmission and the Internet Protocol hands it off to the hardware for this purpose. Any process may send a packet of any protocol type at any time after the internet package has been initialized at the processor where it resides.

Processes are somewhat more limited in their ability to receive packets. A particular process may only receive packets of one particular protocol type at a time. It specifies the protocol value of packets it is willing to receive by creating a handler for that protocol. Handler creation simply means that the internet package has been informed that this process is now willing to accept packets of the specified protocol type (and no other). A process that is done accepting packets issues a destroy-handler command. A process that desires to receive packets of a different

protocol type from the one it is currently receiving must issue a destroy-handler command first, and then may issue a create-handler command for the new protocol value. At any time, only a single process at a particular processor may accept packets with a given protocol type.

Packets arriving at their destination processor are handed by hardware mechanisms to the Ethernet Driver existing there. This is a high priority process. The Ethernet Driver, in turn, hands control of the packet to the Main Dispatcher, yet another high priority process. The Main Dispatcher interacts with the internet package to notify the appropriate program process (based on the packet's protocol field) of the arrival of the packet.

The process that desires the packet must issue a special request in order to obtain it. There are two possible ways to issue this request. The process can call a maybe-receive procedure, which attempts to acquire a valid packet and immediately returns if none is present. The process can also call a receive procedure, which attempts to acquire a valid packet and will wait on a condition variable if none is present. Should a packet arrive before the process times out, it will be so notified by the Main Dispatcher and it will be able to acquire a packet. If a timeout occurs before a packet arrives, then the process may simply reissue its receive command and recommence waiting on the same condition.

Thus, we see that a call of maybe-receive is satisfied by any packet that arrives strictly before the call. However, a

call of receive may be satisfied by any packet arriving before the call or by any packet arriving in the interval between the time the process begins to wait on the condition variable and the time it times out. This is a crucial point, and one which must be understood in order to appreciate the implementation described in chapter three.

We add that if the condition variable had a timeout of zero (no waiting is done - this is different from having a zero value in the timeout field of the PSB, which would imply that the condition has been disabled) then the receive and maybe-receive calls are identical.

The efficiency of our debugging facility heavily depends on the length of the timeout interval of this condition variable (see section 3.4.2). As this interval is increased, the facility will function more slowly. Indeed, if the interval goes to infinity (i.e. the timeout is disabled) the facility will cease to function at all. The timeout of this particular condition variable must under no circumstances be disabled if use of the debugging facility is intended. Since this condition variable is embedded in the internet code, there is usually no reason for the programmer to tamper with this value.

When no further interprocess communications need to be performed by any of the processes residing at a processor, some process there is free to call a destroy-internet-package procedure.

## 1.8 Plan of Thesis

Chapter two discusses how and why our debugging facility will be used. It introduces the notion of a lurking-bug and how the facility may be employed to detect these. It discusses the issue of transparency introduced in this chapter and shows both theoretically and practically how debugger transparency may be maintained while interactive debugging of distributed programs proceeds.

Chapter three provides a detailed description of the debugging facility we have implemented. Those who have read this far may skip to it directly, if they wish, as, for the most part, it may be understood independently from the rest of the thesis.

Chapter four proves that the debugging facility is correct and useful. That is, it proves that the debugging facility may be validly used to debug a distributed program and that the program being debugged is the intended one. However, we see that the facility is not quite totally transparent towards the latter.

Chapter five discusses some ideas that we have not implemented for various reasons. We suggest a number of topics for future research and thought.

## 1.9 Some Definitions

We have repeatedly used a few terms in this thesis that we felt were naturally understood. However, this may not be the case. Thus, we define them here:

node - A node is a processor connected in a network. Since we wish to emphasize that the program to be debugged resides on several interconnected processors, we refer to them as nodes throughout the rest of this thesis.

application - The application is the program to be debugged. Both it and the debugger are distributed across the network.

user - The user is the person who employs the debugging facility to debug an application. The user may or may not be identical to the person who actually programmed the application (the programmer).

# Chapter Two

## Issues in the Design of a Debugging Facility

This chapter provides a detailed introduction to the problems involved in debugging an application that is distributed across a computer network. The concept of transparency, alluded to in the first chapter, has been important in guiding our research. We motivate the design presented in chapter three by explaining how it helps achieve a high degree of transparency during interactive debugging of distributed applications.

Related to transparency is the notion of providing the user with precise control over events occurring during the debugging session. In the following discussion, we indicate how transparency implies that interprocess communications (the "events" with which we will be concerned) are controlled solely by the user and are unaffected by the existence of the debugging facility. In chapter three, we delve more fully into the mechanisms provided by the facility for such precise control (i.e. the ability to duplicate communications, to delay communications for specified lengths of time, to prevent communications from taking place, and, most importantly, to create any pattern of interprocess communications that may be desired).

This chapter deals with the theoretical as well as the practical. It is our desire to describe not merely a particular scheme that works only for the Alto/Mesa/Internet

environment, but to present these ideas as a theoretically reasonable model for future designers of debugging facilities for any distributed system.

## 2.1 Use of the Debugging Facility

We stated in chapter one that the facility herein described has use both as a monitor and as a debugger of distributed applications. We now assert that, as a debugger, by far the most interesting use is in the detection of lurking bugs (Van Horn66), defined below, in programs consisting of sets of processes executing in parallel. It is generally acknowledged (Myers80) that the detection and elimination of lurking bugs is one of the most difficult and frustrating of all debugging related tasks. Yet, up to now, the tools available to aid the programmer in this have been scant. Our debugging facility does not guarantee detection of all lurking bugs. It does, however, provide a tool for the skillful user, which transcends previous debuggers in providing help in this important area. The concept of a lurking bug will now be made precise.

An important feature of parallel processing is that of nondeterminacy of computation. It is unusual for even a moderately sized program consisting of two or more processes executing in parallel to proceed in the same way during distinct executions. This is because such executions are performed in an arbitrarily timed (Van Horn66) manner. By arbitrarily timed, we mean that the order in which processes acquire the processor for execution is not well defined. In a distributed environment, furthermore, the timing relationships between processes executing on separate processors (e.g. which processes execute before or after others, which processes

execute in parallel) are also not well defined. Stochastic events are constantly at work in a system making it impossible to predict, a priori, the timing relationships among the various processes. For example, the results of a particular execution might be affected "because of slight variations in the speeds of autonomous processing units, because of replacement of one system component by another of different speed, because of variations in the duration of i/o activity, or, perhaps most significantly, because of the scheduling strategy of a multiprogrammed system." (Van Horn66)

Nondeterminacy means that it is impossible to predict the next computation state of the machine based on the current state, as is possible when analyzing a single process computation. Since it cannot be foretold which process (or group of processes in a distributed system) will be the next to begin execution from the current state, it cannot, in consequence, be foretold how the state will change; what memory and register locations will be affected and in what way. Nondeterminacy is a given, however. The very nature of parallel processing implies that interprocess timing relationships may be very loose and may vary from execution to execution. It is the burden of the programmer to insure that his application is robust (functions "correctly") for any possible sequence in which the processes may be executed.

Now it is possible that not only will certain machine states arise during a particular computation that may or may not be seen again during the lifetime of the program (i.e. until

it is scrapped or replaced), but certain errors of this fleeting type may be detected too. Those errors that arise during particular computations, for which it is impossible to predict their recurrence in ensuing computations, and which may never have manifested themselves before, and may never manifest themselves again, but which are there, are called lurking bugs. Lurking bugs become apparent during a particular computation because the order in which processes have executed has shown up a logical flaw in the program. A different execution ordering during another computation, may be sufficient to mask this flaw.

We present an example (Van Horn66). Consider an application consisting of three processes. Process A writes a value to a memory cell which is then read and output to a file by process B. Process C contains an error in its coding. It accidently puts an incorrect value into the same cell that process A is writing and process B is reading (it was, say, supposed to affect an entirely different cell). Now consider the following two process execution sequences (on a uniprocessor machine) for two possible computations:

i) A B C A B C A B C . . .
ii) A C B A C B A C B . . .

In computation i), process C never affects the memory cell in question until process B has already read it and written it to the file. Thus the affect of process C is invisible in the final output. In computation ii), however, process C always changes the value in the cell before process B is able to read

it. In this case all values written to the file are incorrect. The lurking bug, an error in the coding of process C, has become manifest due to the particular ordering of process executions in computation ii). (The reader may easily imagine certain execution sequences intermediate to the completely correct computation in i) and the completely incorrect computation in ii); for example, executions that yield some correct values in the output file and some incorrect values. The reader may also imagine certain questionable computations. Is, for example, A B B C A B B C . . . "correct" or not? We return to this problem in chapter four.)

This is a simple example, but it should be easy to see how in large programs consisting of many dozens of cooperating processes, it is difficult, if not impossible, to feel assured that all lurking bugs have been eliminated in a program that appears to work correctly.

We digress, for a moment, to point out that even users of languages without parallel processing capabilities (such as Fortran, Algol60, etc.) are not immune to the problems of nondeterminacy and lurking bugs. In today's computational environments, no process is an island. Any application must coexist with various operating system processes; schedulers, i/o routines, other user applications, and the like. Yet the Fortran programmer who believes his application to be determinate, because, for a given set of inputs, he can trace step by step through his listing predicting subsequent states from earlier states until the final results have been determined,

is safe in his naivete. This is because the designers and implementors of the system being used have taken the burden of worrying about lurking bugs on themselves. They have caused user-system and user-user process interactions to be of the simplest type so that the order in which system and user programs execute is of minor consequence. All programmers, however, should be aware of these problems. As networking grows and as languages which directly incorporate parallel processing become more prevalent, the onus of ensuring correctness in the face of nondeterminacy is no longer solely on the shoulders of the systems programmer. Tools for the detection and analysis of lurking bugs will become increasingly important to both systems and applications programmers.

We have been careful so far to refer to the system herein described as a "facility" or a "tool" for debugging, not actually as a debugger itself. It allows the user to monitor and influence directly only the interprocess communications during a particular computation, not the sequential instructions that define process events (as discussed in the first chapter). By use of this facility, bugs can be detected, be they lurking or otherwise, in an indirect fashion, based on how these bugs manifest themselves as errors in the communication streams. In conjunction with conventional debuggers, which can be used to monitor and influence process events themselves, this facility provides a powerful debugging system for distributed computations.

It is assumed that the debugging facility will be used in

a number of ways. We don't wish to overstate its use as a detector of lurking bugs. Most users will employ it simply to check whether interprocess communications proceed in a reasonable fashion. They will execute a handful of computations, permuting the order in which packets are sent and received, varying transmission times for particular packets, losing packets, etc, until they are reasonably certain that their application functions correctly under most conditions.

A second, slightly more sophisticated, mode of use would be to monitor and influence communications up through a certain point in the computation. The user might then choose to monitor or debug directly any one of the nodes involved in the computation. He may employ a remote debugging facility to examine another node directly from the node at which he is situated. (In the Alto/Ethernet environment there exists a remote debugger called Teleswat (Xerox79c) which allows any node on the network to attempt to debug any other, with the consent of the latter. This is achieved by passing messages between the two sites.) He may also physically go to the site he wishes to examine and make use of a conventional debugger existing there. Debugging (by either means) can proceed up through the next internode interaction involving that site. This can be done for all nodes involved in the computation. The user may alternate between using the debugging facility to monitor communications and debugging sites individually, remotely or otherwise.

Finally, the facility may be used to detect lurking bugs.

No claim is made that all lurking bugs will, or even can, be detected since it is usually impossible to test all possible process execution sequences for correctness. For any untested execution sequence there may exist undetected lurking bugs. However, we hypothesize (with fairly strong feelings of justification) that it is often the case that the user has a general "feeling" for his program that tells him which particular execution sequences are more likely to house lurking bugs than others. The facility provides a tool to allow the re-creation of those execution sequences which are of particular interest, via manipulation of the communication streams. The user chooses for examination a small subset of the myriad of possible computations.

As an example, the user may formulate a set of computations that causes all the code in every process to be executed at least once. In communications software, a great deal of code is often written to handle unusual conditions (for example, extremely long packet transmission delays due to hardware problems). Since these conditions rarely occur, this software is left untested. The debugging facility allows these conditions to be simulated, creating a set of test cases in which all program code is executed. If these yield satisfactory results, the user may presume (perhaps justifiably, perhaps not) that his code is free of lurking bugs.

This example hints at how the debugging facility is used to create different execution sequences. By delaying a packet, for instance, the user may delay the execution of the receiving

process, thereby changing the order of processor acquisition by processes at the receiving node. The user then determines whether his program functions correctly for this particular execution sequence which he has just produced.

Debugging in this fashion may be likened to a chess game. During any move, the player has dozens of avenues to explore, and the deeper he searches the more rapidly the number of alternatives increases. However, the vast majority of such moves are tactically silly or meaningless. The player does not get bogged down in analysis because he is able to immediately dismiss these possibilities and concentrate on the handful of interesting moves. Like the chess player, the user of this debugging facility is able to eliminate all those possible computations that he feels are not necessary to explore. He is given a tool which allows him to concentrate only on the meaningful alternatives. He possesses precise control over the interprocess communications occurring during the execution of the program.

To continue the analogy, moreover, a single session with the debugging facility can be likened to the chess player's top-down exploration of a particular avenue of attack. By a session, we mean the interactive use of the facility to monitor and influence the application through the course of a single computation. Just as the chess player mentally decides on a move to bring the game to a particular (usually more advantageous) state, and then extrapolates his next move based on this state and his opponent's reply, and so on, so the user

employs the facility to create various execution sequences to bring his program to a particular state, and then decides on his next "move" based on that state. This pattern continues until the computation concludes.

We don't wish to carry this analogy too far, however. The chess player possesses the luxury of backtracking when his extrapolations lead to a poor position; the user does not. Backtracking would require the inclusion of state recovery mechanisms which are well beyond the scope of this thesis. The addition of these mechanisms would, however, make for an extremely powerful debugging facility, and this is a worthwhile avenue for future exploration.[1] Currently, the effects of backtracking are achieved by the clumsy method of restarting the computation from the beginning, bringing it back up to the last state that the user was satisfied with, and proceeding on new paths from that point. The ability to accomplish this implies that the user possesses the precise control mentioned at the outset of this chapter. However, we shall see in chapter four that stochastic processes may work to prevent precise control by destroying the complete transparency of the debugging facility. Stochastic processes can reduce a completely transparent debugging tool to one that is only more or less transparent.

1 This is currently being investigated as a Ph.D. thesis topic at M.I.T. by Wayne Gramlich.

## 2.2 Practical Considerations: Transparency and Artificially Induced Communication Delays

The debugging facility is a program that enables the user to be aware of any message packet transmitted by any process within the application being debugged. The facility possesses code that intercepts any such packet before it is sent to its destination process and reroutes it to a central debugging facility receiving area.

This central area is responsible for reporting the existence of the packet, as well as various other pertinent information, to the user of the facility. The user, then, is free to make decisions about whether this packet is to actually be transmitted to its original destination process, whether its transmission is to be delayed for a specified amount of time, whether another packet is to be transmitted in place of the one in question, etc. The implementation of the debugging facility is described in much greater detail in the following chapter.

Thus, the facility provides the user with the capability to examine and make decisions about packets after they are transmitted from the source process and before they are received by the destination process. The destination process does not receive its packet until

the user has given explicit permission for it to do so. It is therefore obvious that interprocess communications will be slowed down by many orders of magnitude. The central problem to be addressed, then, is how to maintain the execution of processes at computer speeds in the face of interprocess communications that proceed at severely retarded, and quite arbitrary, speeds. The user should be able to make decisions about packets at his leisure, yet the computation of the application must remain coherent.

More than mere "coherence" is required, however. What is desired is the complete transparency of the debugging facility towards the application program. It makes no sense to attempt to debug a program when its behavior has been rendered unrecognizable by the debugger itself. Just as a thermometer ought not to affect the temperature of a liquid which is being measured, so the debugging facility ought not to affect the application which is being debugged.

How is execution affected by arbitrary communication delays? Let us pretend that we have an application in the midst of execution with process I on the ready list of one of the participating nodes at time t. At time t + 1, a communication packet arrives

for process Q, which is duly placed on the ready list at time t + 2. At time t + 3, process J gets placed on the ready list. Finally, when process Q executes it notifies a process L (time t + 4) and when process J executes it notifies a process M (time t + 5). Thus the order in which the processes acquire the processor is: I, Q, J, L, M.

Now suppose that the packet that should have arrived at time t + 1 is, in fact, delayed until time t + 10 (because the user has been examining it). Then not only will process I execute ahead of process Q, but so will process J. This reordering of the execution sequence has no effect unless processes Q and J directly communicate, say, through a monitor, during their executions. (Strictly speaking, this is not quite correct: if processes Q and J communicate even indirectly during their executions, then there may be an effect. Indirect communication between Q and J implies the existence of some process X such that there are communication paths from both Q to X and J to X. A communication path exists from processes $M_m$ to $M_n$, denoted $M_m \rightarrow M_{m+1} \rightarrow M_{m+2} \rightarrow \ldots \rightarrow M_n$, if for every q, $m \leq q < n$, a packet stream is open between $M_q$ and $M_{q+1}$, or a monitor exists that is accessible to both $M_q$ and $M_{q+1}$. This definition is similar to the path concept found in Bryant77.) Suppose that they do. Then, in the first case, process Q enters the monitor before process J. In the second case, the entry order is reversed. The consequence of this is that both Q and J see different states of the monitor data than they would have had the packet's arrival not been delayed by

the user. It is then possible that the actions performed by both Q and J will be different from what would have been had the packet not been delayed.

The fact that the processes will see different monitor states than they would have is a consequence of the semantics of the monitor construct. Upon entry to a monitor, a process may see, not a particular state, but any one of a range of states that satisfy the monitor invariant. As far as program correctness goes, as long as each entering process sees some state that satisfies the invariant, the order of process entries makes no difference. Monitors, then, are designed to take into account the inherent nondeterminacy of parallel processing.

Yet we wish to draw a distinction between program correctness and the maintenance of debugger transparency. The reader must realize that the scenario described above violates the principle of transparency of the debugger facility. The facility has made its presence known to the application by causing various states to arise that would not have arisen had it not been present.

Thus one effect of delaying the message lies in the states that processes Q and J will see and the actions they will take based on these states. Nor is this effect limited to only processes Q and J. The order in which Q and J execute will determine the order in which L and M, the processes Q and J notify, execute. If L and M communicate via a monitor, then the same problems apply to them as apply to Q and J. Thus it

is not difficult to see that a single debugger facility induced change in the execution may propagate rapidly, perhaps vastly altering events right through to the conclusion of the execution.

Nor are these effects limited simply to differences in the values of data seen by processes. Suppose that process I, above, is in charge of making sure that the communication stream between process Q and the process sending the packet is functioning correctly and terminating the connection if it is not. It may be that I and Q share a monitor whereby Q, upon receiving its packet "leaves word" for I that the stream is functioning normally. I periodically waits, wakes up, and checks this monitor. If I makes z consecutive checks without finding that Q has received its packet, it aborts the entire connection, destroying any related tables it may have set up for bookkeeping purposes. When packets arrive on time, I and Q alternate in execution (ignoring other processes at the node): I Q I Q I Q . . . When packets habitually arrive late due to the affects of the debugger facility, the execution might be I I I . . . I Q I I I . . . Q I . . . The risk of I destroying a connection that ought not be destroyed is apparent.

This, then, is the real danger introduced by lack of transparency on the part of the debugger facility: the destruction of communication streams (and consequently the disintegration of the computation) by processes which presume communication failures because their real-time expectations (that is, their insistence that certain events must take place within x seconds) have not been met.

How do we combat all of these problems? One way to mask arbitrary communication delays due to the debugger facility is to slow down the executions of the processes themselves to maintain synchronization with the slowed down communications. This is achieved by process suspension, that is, artificially delaying a process which is ready to execute from acquiring the processor. Furthermore, when a process that is supposed to receive a packet has its execution delayed because the packet has been delayed, we prevent the execution of processes that should not execute until after this one, by suspending them. In the example discussed earlier, if the packet for process Q is delayed, in turn delaying the execution of that process, then process J should be artificially delayed, or suspended, until such time as process Q receives its packet and executes. Then the problem of J entering a monitor before process Q and seeing a state it would not have seen, and the problem of J notifying M before Q can notify L thus altering the sequence in which M and L execute, become nonexistent.

We state that for a given node, the problem of maintaining transparency is solved by ensuring that the order in which processes are placed on the ready list, hence the order in which processes execute, is the same with the debugging facility present as it would have been had the application been executing without it. So far transparency has been discussed only in an intuitive manner, and we ask the reader to accept this above assertion intuitively, for the moment. We postpone a more concrete discussion of transparency and a

more detailed explanation of this statement until the next section.

At any rate, in our example, when process Q cannot execute because its packet has been delayed, we must make sure that no other processes execute in the interim. This is easily accomplished by having a debugging-facility-created process seize the processor and loop until Q's packet arrives, at which time the processor can be relinquished and Q can execute. This mechanism is referred to as node suspension, since its effect is to prevent any activity from taking place while Q's packet is being awaited. At the time of relinquishment, it is the job of the looping process to restore the state encountered when the processor was seized. Thus, node suspension is rendered invisible to the processes of the application being debugged.

Of course, there is nothing new about this procedure. Conventional debuggers have always used it to allow breakpointing. The user has always been able to specify an instruction at which he wishes his application to be suspended, to examine and alter the state of the computation at his leisure, and to recommence execution when he desires. Theoretically, a debugger guarantees that breakpointing is transparent by restoring the state at the time the breakpoint occurred when execution restarts.

But now we are dealing with distributed systems, where it is impossible to suspend the computation by seizing the processor, because there is more than one processor. If we

suspend processes at one node and allow other nodes to continue executing, then communications may break down because the delay time of packets originating from the suspended node will prove intolerable, and will appear to the unsuspended processes as stream failures. These processes would close the streams and the computation would disintegrate.

Suspending one of the nodes involved in the computation for x seconds causes this node to execute x seconds "behind" all of the other nodes in the computation. This means that other nodes will see (through the communications streams) all events at this node occurring x seconds later than they would have had the node not been suspended. The transparency of the debugger facility would again be lost. Just as transparency was lost when packets were delayed for user examination, it is now lost because packets from this node have been delayed due to node suspension.

One might attempt to solve this by suspending all nodes simultaneously whenever any of the nodes needs to be suspended creating a kind of internode breakpoint. Then, relatively speaking, no node will be perceived as having lost x seconds because all nodes will have lost the identical amount of time. Conventional debuggers achieve breakpointing by stopping all processes at the same point in time. This is easy to do when only one processor is present. It is, however, impossible to achieve in a distributed system since one cannot guarantee (due to unpredictable loss or delay) the simultaneous receipt by all nodes of "suspension command" packets. Nor would it

do, as an escape from the necessity of simultaneous receipt, to include in each packet the time at which the node should suspend itself (so that each node will suspend at some time, x, in the future). This is because it is impossible to maintain the perfect synchronization of the clocks at each node, and, more importantly, it is impossible to guarantee that transparency will not already be lost before time x is reached. Thus, we cast about for a solution which is independent of the concept of simultaneous events; independent of the notion that suspension of all nodes must occur at a single point in time.

We just now stated that a node will not notice that another has been suspended until it examines its communication ports. Herein lies our salvation, for as long as there is no communication between the suspended and unsuspended nodes, the latter cannot possibly notice a loss of transparency. Suspension need not be done until such time as one of the processes at the unsuspended node requests a packet. Then this node is suspended until it can receive its packet from the original, suspended node, which, in turn, proceeds when the user is through examining the original delayed packet and allows it to be sent. Thus, to render debugger facility induced communication delays invisible, the execution proceeds with various nodes alternately in states of execution and suspension. Node suspension occurs whenever a process on that node requests a packet. It may last for an arbitrary interval of real time. It concludes either when the requested packet

arrives or when it is finally determined that no packet is available to satisfy the request.

Now the processes of the application are no longer executing in real time. Node suspension has caused execution to slow down the same amount of time for each process on the same node, but, since the length of suspension of one node is unrelated to that of another, different amounts of time for processes residing on separate nodes. Each node now is executing in a logical time, reading its own logical clock that is unrelated to the logical clock of any other node.

The consequence of this is that the timing relationships that would have existed between process executions on different nodes are changed. They are not the same as they would have been had all nodes been executing in real time. Hence, there is again a danger that transparency will be lost. For example, suppose process A at node a communicates with process B at node b and process C at node c. Furthermore, suppose that, due to node suspension, node c is executing behind node b in logical time. Then it is possible that message $m_B$, from process B, will reach process A before message $m_C$, from process C, when, had execution been proceeding normally in real time (without the debugger facility) the order of receipt would have been reversed. This is one possible effect when a node has been caused to execute more slowly than it would have.

Furthermore, the fact that a node executes behind another in logical time implies that the latter is executing ahead of the first (of course). This leads to yet another set

of problems. Suppose process C, above, is expecting a packet from process B. It is possible that process C will receive the packet too early, earlier than it would have had execution been proceeding normally in real time. It is interesting to note that a solution which takes into account the effects of packets arriving too late must also consider the effects of packets arriving too early.

All of these problems, which are due to the alteration of internode timing relationships by the debugging facility, are solved by a mechanism which causes any process to see all external events (those due to other processes) in the same relative time and order as it would have seen them had the debugging facility not been present. This is accomplished by assigning a timestamp to all external events of which a process is aware (in other words, assigning a timestamp to each packet in the communication stream; a process cannot be aware of an external event unless that event is reported to it via the communication stream). Timestamping was first used (Johnson75) to order a set of events when the danger of a different, incorrect, ordering being perceived arose. However, the mechanism was used to solve an entirely different problem than is examined here. We defer until chapter three a description of the method by which timestamps are formulated and assigned.

To summarize this section, then, we have stated a need to maintain transparency in the face of artificially induced communication delays. We suspend the process which is expecting the delayed packet in order to render the delay invisible.

Then, to make sure that other processes at the same node do not notice monitor states that they should not because of this suspension, we suspend the entire node. This ensures that the ordering of events at the node is unaffected by the debugging facility, hence transparency is maintained at that node. Finally, to keep the order in which all external events are perceived invariant, we assign timestamps to these external events. This preserves each process' perception of internode timing relationships. Preserving the order in which events occur at a specific node, and maintaining the order and timing of external events as seen by each node is, we postulated, both necessary and sufficient to maintain transparency towards the application. The debugging facility, as a result, only affects the application in ways dictated by the user. The user possesses precise control over the events in the system. The measuring tool, itself, does not affect that which it was assigned to measure.

## 2.3 Theoretical Basis: Causality and Systems of Logical Clocks

We now wish to examine the issues discussed in the last section from a more theoretical perspective. Our reason for doing this is to show how a debugging facility ought to work for any process system, not just for the Alto/Mesa environment in which it has been implemented. Before we can do this, however, we need to precisely define a term we have used somewhat loosely thus far.

A computation, c, (Van Horn66) is defined to be a single execution of the processes making up an application. It is represented by a run, R, (Van Horn66) which is, in turn, defined as the ordered pair $\langle S_R, T_R \rangle$ where $S_R$ is an initial computation state (the state of the machine when the computation commences) and $T_R$ is a (possibly empty) transition sequence $T_0, T_1, T_2, \ldots, T_n$ where each $T_i$ is the set of processes in execution during the time interval $[i, i + 1)$. The number of elements in each set, $T_i$, is limited by the number of processors involved in the execution. The transition sequence, $T_R$, is a generalization of the turns-history concept (Jaffe79). A turns history is merely a sequence of process names, indicating the order in which processes execute on a single processor.

Because of nondeterminacy of execution, the run of a computation performed at time t may differ from the run of a computation performed at time t' even though the executing application is the same in both cases. Also, a run specifies

all the interprocess timing relationships among the processes of the application. That is, by looking at the run, one may determine which processes executed before or after others, and which processes executed in parallel. For any time, t, the identity of processes executing at that time may be determined.

Lamport (Lamport78) has devised a useful way to represent sets of computations pictorially (see figure 2.1). In this diagram, each vertical line represents the execution of a distinct process involved in the application. The dots on each vertical line represent the sequence of events that define that process. The wavy arrows represent any form of interprocess communication. Lamport defines these as representing the transmission of a packet by a process (the tail of each wavy arrow) and the receipt of that packet by another process (the head of each wavy arrow). Since, in our system, interprocess communication is achieved either by the explicit transmission of packets or through monitor interactions, we extend this definition. The wavy arrows will also represent the release of a monitor lock by one process (the tail of each wavy arrow) and the acquisition by the next process of that same monitor lock (the head of each wavy arrow). The vertical direction represents the passage of physical time. That is, the events at the lower part of the diagram occur (in real time) before those that are higher. The intersection of a dotted line and a process arrow represents the instant when the clock for that process reads time t. Since all process clocks run in real time (assuming they are well synchronized) it is reasonable

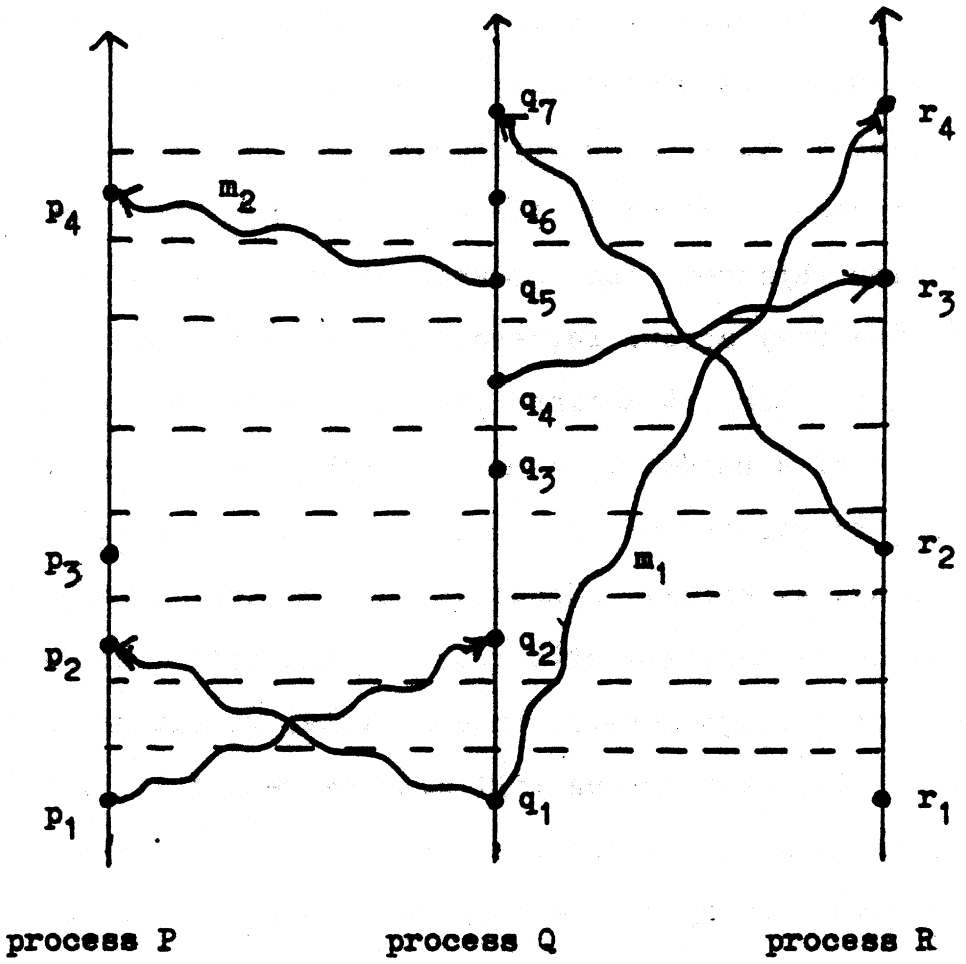



**Figure 2.1**

(from [Lamport78], fig. 3)

that these dotted lines are horizontal.

Lamport defines what it means for an event to "happen before" another in this system.

> Definition. The relation "->" on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b, then a -> b. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then a -> b. (3) If a -> b and b -> c then a -> c. Two distinct events a and b are said to be concurrent if $a \not\rightarrow b$ and $b \not\rightarrow a$.
>
> . . . It is easy to see that a -> b means that one can go from a to b in the diagram by moving forward in time along process and message lines. For example, we have $p_1 \rightarrow r_4$ in Figure [2.1].
>
> Another way of viewing the definition is to say that a -> b means that it is possible for event a to causally affect event b. Two events are concurrent if neither can causally affect the other. For example, events $p_3$ and $q_3$ of Figure [2.1] are concurrent. (Lamport78)

Thus we see that a diagram such as this can be used to show both "happened before" and "concurrent" relationships, existing among the events in the system. It represents a set of computations, rather than a particular computation, in that there may be more than one run that yields the "happened before" and "concurrent" relations depicted. That is, it is possible that there are many sets of interprocess timing relationships that yield the same causal dependences as shown in the diagram.

For example, if arrow $m_1$ represents a monitor entry, then any computation with a run which has process Q entering the monitor immediately followed by process R may be included in the set of computations depicted by the diagram. The other timing relationships in the diagram may serve to narrow down the set of represented computations somewhat further.

Suppose we decided to see what would happen if one of the communication arrows in the figure (arrow $m_2$) was lengthened (as in figure 2.2) so that the head of the arrow intersected with the process line at a higher point, later in real time. It ought to be clear that the causal relationships defined by the original diagram have been lost. Whereas before it was possible for $q_5$ to causally affect $p_4$ ($q_5 \rightarrow p_4$), now it is true that $q_5$ and $p_4$ are concurrent. Therefore, the new diagram represents a new set of causal relationships distinct from that of figure 2.1. (In fact, we point out that the lengthening of the arrow may mean that event $p_4$ will not occur at all, or will be replaced by event $z_1$, as in figure 2.3. Then, certainly, the relations represented in the original figure have been lost.)

We would like, however, to maintain the same causal relationships as shown in the original diagram. We do not mind changing the run (changing the interprocess timing relationships to create a new set of computations) as long as it is possible to retain the original "happened before" and "concurrent" event relations. That this is possible we already know, because it was stated above that more than one computation may define the same set of causal relations. We search for a new computation to maintain these in the face of the lengthening of one of the communication arrows.

It is clear that in order to compensate for the stretching of the arrow, the vertical process line, P, must also be stretched so that $q_5$ can once again be seen as "happening before"

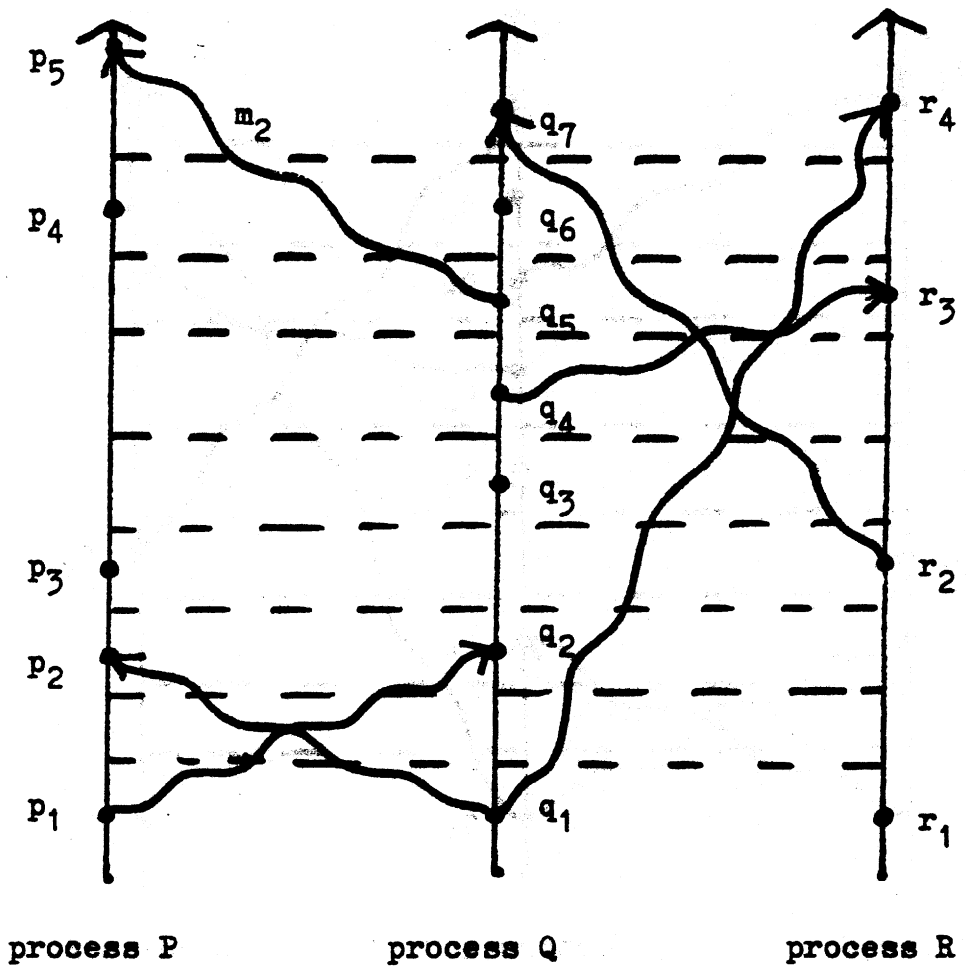



Figure 2.2

(based on (Lamport78), fig. 3)

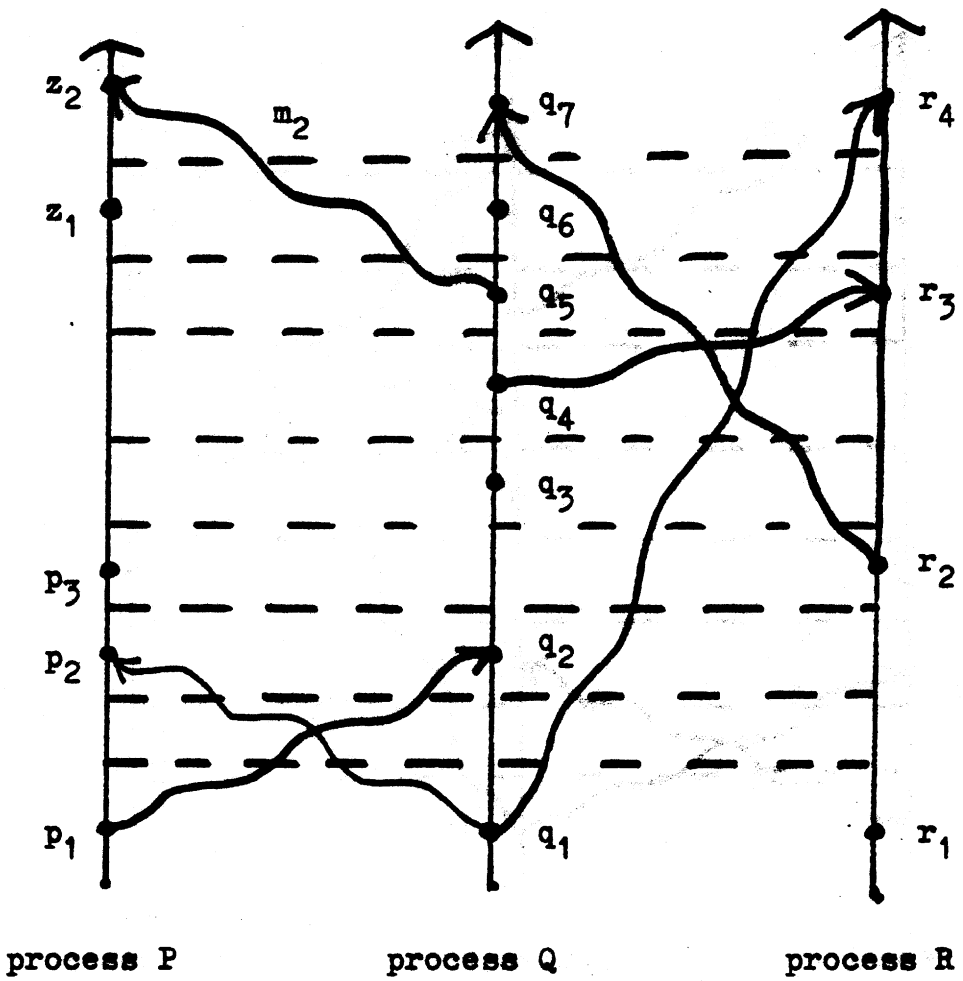



Figure 2.3

(based on (Lamport78), fig. 3)

$p_4$ (see figure 2.4). Notice, however, that this will cause the dashed line representing physical time to be bent away from the horizontal. This implies that processes Q and P will read the value x (the time represented by that dashed line) on their respective clocks at totally different real times. This is not possible in a system of well synchronized physical time clocks. Here is the crux of the matter. A new set of computations can be found to restore the original causal relationships, however none of these computations are executable in real, physical time. That is, an abstract mechanism, a logical time clock (as opposed to a physical time clock) must be introduced into the system. Furthermore, there must be a private logical clock for each process, since various alterations of the communication arrows may rapidly cause all processes to be executing in their own unique logical times. The new set of "logical time" computations may be depicted as in figure 2.5. These logical time computations and the original set of real time computations in figure 2.1 both yield the identical set of causal relationships.

Now we state the central point of this thesis. As Lamport has pointed out, ". . . <u>there is no way to decide which of these pictures [figures 2.1 and 2.5] is a better representation. . .</u>" of the particular set of causal relationships. Practically, this means that it is possible to <u>simulate</u> the effects of the real time computations using one of the logical time computations. Causality can be maintained in the face of alterations in the

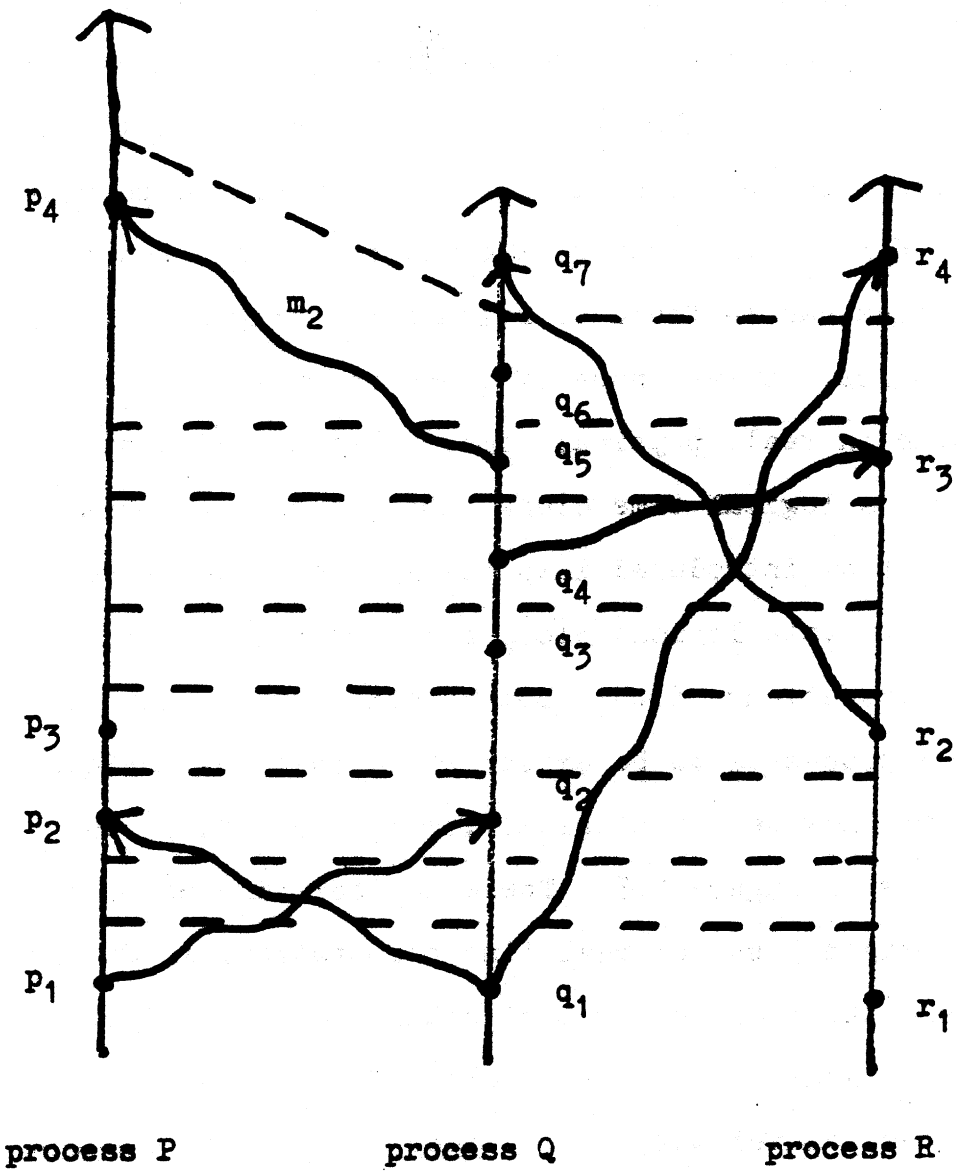



Figure 2.4

(based on (Lamport78), fig. 3)

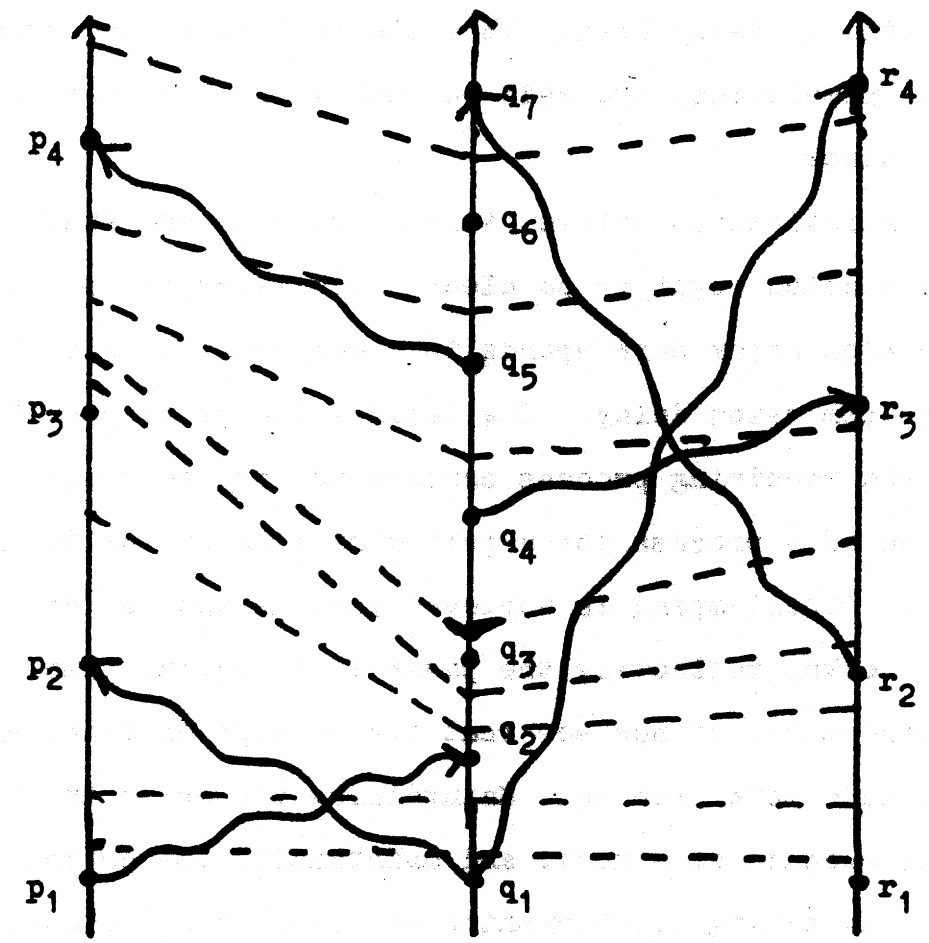



**Figure 2.5**

(from (Lamport78), fig. 2)

lengths of the communications arrows.

The use of logical time is an attempt to have each process "believe" that it is executing in real time. That is, the process perceives that all events, both internal and external, are occurring at the same time whether real or logical time is being used. This occurs because processes, under the simulation, are made to read logical rather than physical clocks.

The relationship between this discussion and that of the previous section ought to be clear. The extension of a communication arrow corresponds to a debugger facility induced packet transmission delay. The lengthening of the vertical line of the receiving process corresponds to the artificial suspension of a process for a period of time by the debugger facility. Timestamping is achieved through the use of logical clocks, reflecting the passage of logical time.

Furthermore, we now see that the concept of transparency has been made more precise. Maintaining the order in which processes execute at a node, and maintaining the correct sequence and timing relationships of all external events perceived by any process is another way of stating that the causal relationships between events of the application have been maintained. Transparency, then, is achieved by maintaining these relationships in the face of artificial communication delays caused by the presence of the debugging facility.

We conclude this section by pointing out that the

identical solution to the transparency problem, discussed in the previous section on a practical level, has now been motivated on a theoretical plane.

## 2.4 The Uncertainty Principle of Program Debugging

It is the job of a debugger to maintain causality relations while providing the user with the tools necessary to detect bugs, lurking or otherwise, in his computation. Only if the debugging tool is reasonably transparent is it useful. We have shown, both in a theoretical and practical fashion, how such transparency might be maintained. After describing a Mesa implementation of a debugging facility, we return to the problem of transparency in chapter four. An important question which we have not yet answered precisely is, "What computations are we maintaining the causal relationships of?" In other words, if we are maintaining transparency, what are we maintaining transparency towards? An analysis of this will show that, as previously stated, complete transparency is an unattainable ideal. Stochastic processes reduce our debugging facility to possessing merely a high degree of transparency towards the application being debugged. The tool must affect that which it is measuring.

# Chapter Three

## Implementation of the Debugging Facility

This chapter describes, in detail, the implementation of a debugging facility for distributed applications. The hardware environment for this project was the Ethernet network of Alto minicomputers, as described in chapter one. The software environment was provided by the Alto/Mesa programming system, also discussed in the first chapter.

application node
application node
application processes
nub
application processes
nub
communications medium
central site
nub
application processes
nub
application processes
application node
application node

Figure 3.1

so each nub sends certain information reports to the central site. However, the comparison ends at this point. The central site is no "passive engine" as Metric's accountant has been described to be. We shall see that, as well as merely collecting information, the central site acts on the information by issuing commands or replies to the nubs. The central site actively controls, to a large extent, the events which take place at any node participating in the debugging session.

### 3.1.1 The Central Site

Before the debugging session commences, the user designates a node, distinct from any node on which application processes are executing, from which to monitor and debug his application. This node is the central debugger site, and the user causes the central debugger site code to begin executing here. (The stipulation that the central site must be physically distinct from any node on which application processes are executing is partially a consequence of the small memory size of each Alto. The central debugger site code uses up much of this memory, leaving little room for any application processes to reside. Furthermore, the central site makes extensive use of the Alto screen for reporting information and receiving user commands. Any application process also requiring use of the screen would interfere with user monitoring and debugging.)

The central site provides four essential functions. First, it provides servicing for all nub initiated requests and handling for all nub initiated reports. These nub requests, reports, and central site responses are transmitted in the form of overhead packets of which the user and the application program are never made aware. Overhead packets are distinct from the packets that are spawned by application processes during the course of their executions. The latter are termed application packets.

Second, the central site may issue commands to each nub on its own initiative. The nub is required to obey each command so issued. In this relationship, the central site is clearly master, the nub is clearly slave.

Third, the central site acts as a temporary repository for application packets. In this implementation, the secondary storage of the Alto, a disk (or, occasionally, a pair of disks), is used to cache these packets. Packets that are so cached may take up disk space indefinitely, or may be released by the central site on order of the user in an effort to create more free space. We add that overhead packets are never cached in this fashion. (This is another reason why the central site code must execute at a physically distinct node. In order for the debugging facility to function reasonably well, there must be a certain minimum amount of disk space for caching arbitrary size application packets for arbitrary lengths of time. The presence of such disk space at an application node cannot be guaranteed. Thus a separate node is required.)

Finally, the central site provides the user with an interface to the system with which he is able to monitor and control the proceedings. The information flow is bidirectional. The central site reports to the user various events occurring in the system and various data values. This allows the user to monitor his application. The central site accepts from the user various commands which must be obeyed. This allows the user to debug his application.

### 3.1.2 The Nub

Before the debugging session commences, the user must bind in a version of the nub with all application code to reside at a particular node. An identical nub version must be bound, in this fashion, at each node participating in the session. This binding is done at the time the application code is configured (that is, at the time the various application modules at a particular node are linked together to form an executable program - this is done after each individual module has been compiled). Thus the executing program at each node is a combination of application processes and nub processes.

The nub performs a number of duties. It acts on behalf of the application processes executing at the node on which it resides, forming a kind of liaison between these and the central site. As mentioned, it issues requests to the central site whenever some application process requires it and issues status reports to the central site as necessary. Furthermore,

it processes the replies to these requests and reports.

The nub is also responsible for the correct maintenance of a designated memory location which is incremented at periodic intervals by the Alto hardware. This counter constitutes a logical clock, of the type discussed in chapter two. We note that there is only one such logical clock at each node, regardless of the number of application processes residing there.

Related to this is the concept of timestamping, as introduced in chapter two. This function is also performed by the nub. All application packets are timestamped based on values read off logical clocks. Actually, the timestamping mechanism involves the cooperation of two separate nub versions, the one residing at the node from which the packet emanated, and the one residing at the node where the packet is received.

Also related to this is the mechanism of node suspension. The need for node suspension was motivated in the preceding chapter. It is the job of the nub to make sure that node suspension is performed correctly whenever it is required. There are a number of coordination problems that arise here which must be handled in a reasonable fashion.

Finally, the nub is responsible for intercepting application packets and rerouting them to the central site where they are cached for a period of time, as previously discussed.

In conclusion, the nub is responsible for the coordination and correct functioning of the node at which it resides. The

central site is responsible for the coordination and correct functioning of the application as a whole.

## 3.2 Routing and Timestamping of Application Packets

We now follow the course of a packet spawned by some application process as it makes its way through the debugging facility system (see figure 3.2). We provide the reader with an understanding of the distinct roles played by the central site and the nub and how they interrelate to form the larger system. We also introduce the timestamping mechanism.

When an application process desires to send a packet to some other application process it calls the internet package's Send procedure. This, in turn, makes use of a SendBuffer procedure which eventually hands the packet off to hardware mechanisms that actually do the sending. The nub possesses a hook into this SendBuffer procedure. It causes the following extra information to be appended to the application packet:

1) Time of Day - obtained by reading the sending node's time of day clock, implemented in hardware at each node. All time of day clocks are reasonably well synchronized and reasonably dependable.

2) Logical Time - obtained by reading the sending node's logical clock, as discussed previously.

Also, the identification field of the packet is replaced by a unique debugging facility assigned identifier. The original identifier is appended to the end of the packet body so it will not be lost. The reason for assigning a special identifier in this fashion is that the debugging facility must be guaranteed that all packets emanating from a particular node are distinguishable (for purposes of acknowledgement). No two such

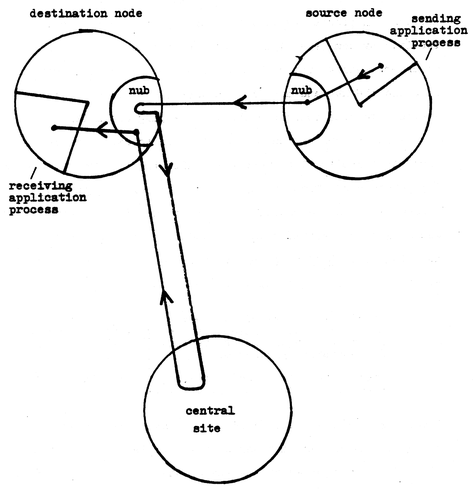

destination node
source node
sending
application
process
nub
nub
receiving
application
process
central
site

Figure 3.2

packets may possess the same identifier. Now it is probable that application processes will have already made sure that this is indeed the case. However, the facility cannot depend on these processes to fulfill this function. The facility must be robust in the face of errors or oversights in the implementation of communication protocols for the application program. Thus it takes this burden on itself.

The ability to add extra words of information to the end of each packet implies that the maximum packet size allowed to the programmer must be a few words less than the real, hardware allowed maximum packet size. In the internet implementation, a number of words at the end of each packet are made invisible to the application writer. Thus, the required extra information can be added regardless of packet size.

We obtain the two time values (real and logical) at the very latest moment possible, just before the packet is handed off to the hardware. This is done in order to avoid the possibility that the times will be obtained and then the sending process will be forced to wait on some monitor lock for an arbitrary length of time, thus nullifying the appended clock values. In the scheme presented here, any delays that occur after the times have been obtained may be attributed to hardware functionality, and are considered as part of transmission delay time.

With this extra information, then, the packet is sent across the Ethernet, arriving at its destination node at some later time. (It is possible that the source and destination

processes reside on the same node, in which case the packet does not physically pass over the Ethernet. However, this is unimportant for our purposes.) For simplicity, we assume that the packet is not lost or discarded, and arrives intact. At the destination node, the packet is routed through the Ethernet Driver and Main Dispatcher (recall chapter one). The latter hands processing off to an InetInput procedure. The nub at the destination node possesses a hook into this procedure. Its first job is to determine that the packet is indeed an application packet that has been sent from some other application node (it is possible that the application packet has come from the central site - we come to this later).

If this is the case, the packet timestamp is now obtained. This is done by reading the time of day clock at the destination node and then performing the following operation:

t = L + (R - S) where

t = packet timestamp
L = logical time packet was sent by source node (from source node's logical clock)
R = time of day packet was received at destination node (approximately - see below)
S = time of day packet was sent by source node (from source node's time of day clock)

L and S were appended to the packet by the nub at the source node.

Thus the timestamp is equal to the logical time on the sender's clock plus the delay time of packet transmission. R is actually obtained just when the internet mechanism would inform the receiving application process that a packet has

arrived. t, then, represents the precise logical time that the presence of the packet is made known to application processes executing on the destination node. The value t is appended to the end of the packet.

It is interesting to note that the logical clock at the destination node does not figure in the timestamping mechanism in any way. Furthermore, it is clear that obtaining a correct timestamp is a cooperative venture between the nub at the source node and the nub at the destination node.

Upon obtaining the timestamp, the nub substitutes the address of the central site in the packet's destination field, after first appending its own address to the end of the packet (exactly how the nub is appraised of the central site address will be discussed later). Now the packet is in a suitable condition for forwarding to the central site.

Notice how the nub at the destination node grabs control of the packet away from the internet code almost as soon as it arrives and does not relinquish this control at any time. Timestamping and all other processing is done privately by the nub. At this time, no application process is aware of the packet's existence. Its arrival and departure are rendered invisible to the application.

The packet now is again sent over the Ethernet, this time to the central site. We assume that it arrives intact. Notice that the packet has been routed to the central site by the destination node, but maintains the address of the source node in its source field (this field was untouched by the nub of

the destination node). Then, the first action taken by the central site is to determine that the packet is indeed from one of the nodes participating in the current debugging session. If this is the case, then the central site causes an acknowledgement packet to be sent to the source node (see figure 7.3). The destination node of the packet need not receive any acknowledgement, although that is the node that routed the packet to the central site. If an acknowledgement is not received by the source node in a reasonable amount of time, it retransmits the packet to the destination node. The central site, then, requires a mechanism to check for duplicates of packets that have arrived due to lost acknowledgements. (If the source node must retransmit the packet, it first obtains a new time of day, which replaces the old time of day previously appended. This is so that the delay time, which will be recalculated at the destination node in an effort to compute a new timestamp for this packet, does not become arbitrarily large. A new logical time is NOT obtained when the packet is retransmitted.) The destination node need not concern itself with any of this, however. It blindly reroutes any application packet it receives, whether original or duplicate.

Having acknowledged the packet, the central site proceeds to restore the original destination node address in the packet destination field and to restore the original packet identification number in the ID field (both of these values having been appended to the packet body). It then cashes the packet

destination
node
transmitted
packet
source
node
rerouted
to central
site
acknowledgement
central
site

Figure 3.3

(discarding the extra information that had been tacked on to the packet, after saving it elsewhere) on a disk file containing all packets bound for the node indicated in that packet's destination field.

Here the packet remains until such time as it is determined that the packet is to be returned to the receiving process on the original destination node (we will soon speak in detail about how it is decided when, or, indeed, if, a packet so cached is to be sent back to its destination node). When the packet is to be returned, the central site retrieves it from the disk. It proceeds to again append the packet identifier to the packet body and to replace it with a unique central site identifier for that particular destination node. Each node participating in the session must see unique identifiers for each packet emanating from the central site (for acknowledgement purposes). No particular correlation need exist, however, for identifiers of packets destined for separate nodes. Also, the central site replaces the original source address field with the address of the node on which it is executing, having previously appended the original source address to the end of the packet. This done, the central site sends the packet over the Ethernet back to the destination node, periodically retransmitting until it receives an acknowledgement in return. Thus, the destination node now receives the packet for the second time. Whereas the first time it received the packet it only needed to blindly reroute it to the central site, now it must be able to handle duplicates arriving due to lost

acknowledgements.

Back at the destination node, the nub determines, by inspecting the source address field, that this packet has returned from the central site (it is not arriving for the first time). The nub restores the original packet identifier, and the original source address. Finally, it causes the packet to be handed off to the application process at that node that is to receive it. It is at this point that the application processes become aware of this packet's existence.

## 3.3 Nub - Central Site Interactions

We stated that the central site and each nub communicate through overhead packets, those which are spawned by the debugging facility for coordination purposes and which are invisible to the application being debugged. Each overhead packet receives a special debugger protocol value in its protocol field (recall chapter one). This value is not used in any application packet types. It allows the receiver (either the central site or a nub) to determine that this is indeed an overhead packet, and not an application packet. We now discuss each overhead packet type in turn, commenting on the function of each.

### 3.3.1 Initialization Packets

A number of packet types are transmitted back and forth in an effort to initially establish communication links between the central site and each nub version. These packet types include the *greetings* packet, the *greeting-response* packet, and the unconditional-execute packet. The roles of these packets will be described fully in the section on initialization mechanisms.

### 3.3.2 Handler-Creation Packets

Handler-creation packets are transmitted by the nub to

the central site. Each must contain a unique value in its identification field for acknowledgement purposes.

A handler-creation packet is used to inform the central site that some application process at the sending nub's node has created a new handler for receiving packets (recall chapter one). It contains two words of information, a protocol number and a timestamp. The protocol number indicates that the application process will only receive packets with that number in their protocol field. The timestamp (obtained by reading the node's logical clock) represents the logical time at which the handler was created.

Upon receiving a handler-creation packet, the central site will acknowledge it and set up tables to indicate that a new packet protocol type is open for receiving at the node from which this packet arrived. Furthermore, all packets already cached at the central site possessing destination fields identical to the source field of this packet and protocol numbers identical to the protocol value shipped by this packet are examined. All such packets with timestamps less than the handler-creation timestamp are flushed from the disk and destroyed (on permission of the user), thereby opening up space for new packets. This is because all packets arriving before the handler was created (according to their timestamps) would never be received by the application process (see chapter one).

### 3.3.3 Receive-Request and Maybe-Receive-Request Packets

Receive-request and maybe-receive-request packets are transmitted by the nub to the central site. Each packet must contain a unique value in its identification field for acknowledgement purposes.

A receive-request or maybe-receive-request packet is used to inform the central site that some application process at the sending nub's node has attempted to receive a packet on its input port via a receive or a maybe-receive, respectively (recall chapter one). Each such packet contains two words of information, a protocol number and a timestamp. The protocol number indicates that the requesting application process receives only packets with that number in their protocol field. The timestamp represents the logical time at which a packet was requested.

Upon receiving a receive-request or a maybe-receive-request packet, the central site will acknowledge it and fork a new process with a function of determining the correct application packet to be returned in reply, if indeed such a packet exists. The algorithm by which this is accomplished will be discussed in detail later. The correct packet to be returned will have a destination field identical to the source field of the request packet and a protocol number identical to the protocol value shipped by this packet.

The central site responds to a receive-request packet with an appropriate application packet, or with a conditional-

execute packet. It responds to a maybe-receive-request packet with an appropriate application packet, or with a cannot-be-satisfied packet. Conditional-execute and cannot-be-satisfied packets are overhead packet types yet to be discussed.

### 3.3.4 Conditional-Execute Packets

Conditional-execute packets are transmitted by the central site to the nub. Each must contain a unique value in its identification field for acknowledgement purposes.

A conditional-execute packet is sent in response to a receive-request packet (it is never sent in response to a maybe-receive-request packet) to the nub that issued the request. It contains one word of information, a timestamp. This packet is used to inform the nub that it must execute up through the logical time indicated by the enclosed timestamp.

Upon receiving a conditional-execute packet, the nub will acknowledge it and save the timestamp. It will then allow the application processes at that node to execute until the logical clock at that node reads the saved timestamp value. At this point, the nub will suspend the node and transmit a give-me-now packet to the central site, indicating that it has performed the action requested of it.

### 3.3.5 Give-Me-Now Packets

Give-me-now packets are transmitted by the nub to the

central site. Each must contain a unique value in its identification field for acknowledgement purposes.

A give-me-now packet is used to indicate that the nub has already requested a packet from the central site, received a conditional-execute packet in response, has executed up to the appropriate logical time, and now expects the central site to forward an application packet to satisfy the original request. It contains one word of information, a protocol number. The receive-request packet that is being followed up by this give-me-now packet is the last one sent with the given protocol number.

Upon receiving a give-me-now packet, the central site will acknowledge it and prepare to send back to the requesting node either an application packet with the given protocol number, another conditional-execute packet, or a cannot-be-satisfied packet. We discuss this in greater detail later.

### 3.3.6 Cannot-Be-Satisfied Packets

Cannot-be-satisfied packets are transmitted by the central site to the nub. Each must contain a unique value in its identification field for acknowledgement purposes.

A cannot-be-satisfied packet may be sent in response to a maybe-receive-request packet or a give-me-now packet whenever the central site cannot find an application packet to satisfy the request. It contains no extra words of information.

Upon receiving a cannot-be-satisfied packet, the nub will

acknowledge it and inform the application process on behalf of which the last maybe-receive-request or give-me-now was made that no application packet exists to satisfy the request. The application processes resume execution without further interference from the nub.

### 3.3.7 Clock-Update Packets

Clock-update packets are transmitted by the nub to the central site. They need not be acknowledged.

A clock-update packet is sent to keep the central site informed of the logical time at the node of the sending nub. It contains one word of information, a timestamp, signifying the logical time at which the packet was sent. These packets are transmitted periodically by the nub of each node participating in the debugging session. In this way, the central site is kept as up to date as possible regarding the logical time of each node. Clock-update packets need not be transmitted by the nub during node suspension (see section 3.4.3).

A tradeoff between efficiency and the number of clock-update packets transmitted exists here. If these packets are transmitted frequently, the logical times can be kept more up to date at the central site and decisions about which application packet to send in response to any receive-request or maybe-receive-request packet can be made more swiftly (see section 3.4.2). However, if packets are transmitted too frequently, they may bottle up the communications medium

causing hardware failures. We have attempted to find a reasonable median here.

### 3.3.8 Package-Destroyed Packets

Package-destroyed packets are transmitted by the nub to the central site. Each must contain a unique value in its identification field for acknowledgement purposes.

A package-destroyed packet is sent when some application process decides to close the internet communications package at the node on which it resides. It contains no extra words of information.

Upon receiving a package-destroyed packet, the central site will acknowledge it and prepare to dismantle all internal tables and data structures pertaining to that node. All packets currently cached at the central site with that node's address in their destination field are flushed from the disk and destroyed (on permission of the user). The net effect is that the central site no longer considers that node to be involved in the debugging session.

Upon receiving the acknowledgement from the central site, and not before, the application is free to destroy the internet package at that node. The nub ceases to execute there, and further application execution takes place independently of the debugging facility.

There is one caveat concerning all this. Subsequent to destroying the internet package, no application process may

attempt to re-create it in order to rejoin the debugging session. This is because it is impossible to tell whether the central site has already destroyed some packets that should have been received by the node when the debugging session recommences (e.g. those packets that are destroyed which contain timestamps that are greater than the logical time at which debugging recommences are possible candidates for such reception). If this capability is desired, the central site must be altered so as not to destroy these packets when a package-destroyed packet arrives.

### 3.3.9 Enter-Debugger Packets

Enter-debugger packets are transmitted by the central site to the nub. Each must contain a unique value in its identification field for acknowledgement purposes.

An enter-debugger packet puts the destination node into the Mesa debugger while under the control of the debugging facility. The user is then able to physically go to the site of this node and debug events occurring there up until the next internode interaction at that site. This ability has not been fully developed, however, as the nub is not coded to correctly handle the logical clock mechanism in the presence of the Mesa debugger.

Upon receiving an enter-debugger packet, the nub will acknowledge it and call the Mesa debugger into execution.

3.3.10 Ack Packets

Ack packets are transmitted in either direction, nub to central site, or central site to nub. Ack packets are used to acknowledge the reception of various other overhead or application packets. They contain one word of information, the unique, debugging facility assigned identification field of the packet that is being acknowledged. Ack packets need not, themselves, be acknowledged.

Upon receiving an ack packet, the receiving site (nub or central site) will cease retransmission of the acknowledged packet.

## 3.4 Low Level Mechanisms

### 3.4.1 Initialization

One goal of the debugging facility is to allow the user to station himself at any node on the network in order to debug an application that may be executing at any other set of nodes on the same network. Thus, when the debugging session commences, the locations of the nub copies are unknown to the central site, and the location of the central site is unknown to any of the nubs. Some method is needed to link up the various parts of the facility, making sure that no application packets are being lost while the linkage is accomplished. Only after linkage has been performed can the debugging session proper get under way.

First we state that the debugging facility places no restriction on the order in which the various nodes involved begin execution. That is, the central site and application nodes may be brought up in any order and no application packets will be lost. The facility will function correctly regardless of this order.

When the central site begins execution (before or after some or all of the application nodes), the user is immediately asked to enter the internet addresses of all nodes participating in the session. As each address is entered, the central site transmits greetings packets to that node. These packets will be sent periodically until acknowledged. Since the node to

which this greeting packet has been sent may not even be executing yet, the central site has no way of knowing when a reply might be received. Therefore it is willing to retransmit greetings packets for a very long time. Eventually, however, if no response is received the central site will alert the user that contact has not been able to be established with that node.

The nub at an application node is not initialized until some application process at that node creates the internet package. Since no packets may be sent or received until this is done, it is obvious that there is no need for the nub to exist until this time. Thus the application processes at that node execute independently until the internet package is created. At that time, the debugging facility assumes control over their execution.

The nub possesses a hook into the internet creation procedure. Its first action is to cause a node suspension until such time as a greetings packet is received from the central site. At this point it does not know the address of the central site, but is able to determine that a greetings packet has arrived by its special debug protocol number. When the greetings packet arrives for the first time (late arriving duplicates are ignored), the address of the central site is recorded and a greeting-response packet is sent back in acknowledgement. This greeting-response packet contains a time value which will be described shortly.

After the nub sends a greeting-response packet it is not free to allow application processes to recommence execution.

Node suspension is still in effect. The central site will acknowledge the greeting-response packet as soon as it is received. However, this is merely so the nub can cease transmitting it. It is not an indication that execution may recommence.

The final stage of the initialization mechanism occurs when the central site receives this greeting-response packet from the node (late arriving duplicates are ignored). It records the fact that this node is aware of the existence and location of the central site and is currently under its control. When such a greeting-response packet is received from <u>every</u> node address entered by the user, then the central site knows that all participating nodes are aware of its existence and location and that they are all under the control of the debugging facility. At this point, unconditional-execute packets are transmitted by the central site to each of these nodes, indicating the fact that they are all free to recommence execution of their application processes.

With the receipt and acknowledgement of the unconditional-execute packets by each node, the initialization mechanisms are concluded and application execution proceeds.

An important procedure is the initialization of logical clocks. The user is given the ability to specify initial values for each logical clock involved in the debugging session. This, however, is an all or nothing proposition. He must either specify initial values for all logical clocks, or he cannot specify them for any. Logical clock assignment

is accomplished by some application process calling a special logical clock assignment procedure bound in with the application modules, but not really a part of the nub proper. The user has the opportunity to specify either the logical time at which execution of the application should commence at that node, or the logical time at which the internet package is created at that node.

If the user has specified a time at which the internet package is created (this must be done before the package is actually created), this value is simply saved for future use. If he has specified a time at which execution should commence (this must be done before execution begins; hence, it must be the first statement executed at that node), this value is immediately placed into the logical clock counter, which will tick uninterrupted until the internet package is created.

When the internet package is created, the nub, as previously mentioned, comes into being. It immediately records two values: the real time of day (from the time of day clock) and whatever value is currently in the logical clock counter. If the user has specified a logical time at which the internet package is to be created, both of these values are discarded and the user specified value is sent to the central site inside the greeting-response packet. If the user has specified a logical time at which execution commences, the value read off the logical clock is converted to a value representing this initial time plus the number of seconds elapsed between the commencement of execution and the creation of the package.

This final value is sent to the central site inside of the greeting-response packet. If no initial clock value has been specified by the user, the time of day is sent to the central site inside of the greeting-response packet.

Thus the central site is informed of the initial value to be assigned to each logical clock.

User assignment of logical clocks is useful in re-creating computations and machine states of interest. It allows each node to begin execution at a specified time relative to all other nodes. It nullifies changes in computations caused by changes in the relative time or order in which execution begins at each node. Thus, the user can bring up each node at his leisure without worrying about how this will affect the computation.

### 3.4.2 Application Packet Selection Algorithm

When the central site receives a receive-request or a maybe-receive-request packet, how does it decide which is the correct application packet, if any, to respond with? We now examine the algorithm that determines this.

Upon receiving the request packet, the central site records the address of the node from which it came, the packet protocol number desired, and the timestamp representing the logical time at which the request was issued by the application process. A new central site process is detached with a function of determining the correct response to the request. When this is finally accomplished, that process is destroyed.

Recall, from chapter one, that a maybe-receive-request can only be satisfied by an application packet which arrives before the request is made. However, a receive-request may be satisfied by an application packet arriving either before the request is made or in the interval between the time the requesting process begins to wait on a condition variable and the time this condition variable times out. In the ensuing discussion, the length of this timeout interval is called t.

The process that is forked by the central site searches through all currently cached packets with a protocol number identical to that found in the request packet and a destination address equal to the source address of that packet.

Let us first examine how a maybe-receive-request is handled (see figure 3.4).

Maybe-Receive Request
Application packet p cached with:
1. correct protocol
2. correct destination
3. timestamp that is:
a) < request timestamp
b) > last handler-creation timestamp for this protocol & destination
c) < timestamp of any other such packet?
1
F
T
Any other logical clock < request timestamp?
2
Any other logical clock < timestamp (p)?
9
F
T
F
T
request cannot be satisfied
3
wait for new status
8
p satisfies request
10
wait for new status
15
report to user
4
report to user
11
user permission to send?
5
user permission to send?
12
F
T
F
T
obey user command
6
send cannot-be-satisfied packet
7
obey user command
13
send p
14

Figure 3.4

The application packet with correct protocol and destination fields that possesses a timestamp which is less than that of the request, greater than that of the last handler-creation timestamp for that protocol and destination, and less than the timestamp of all other such packets, is selected by the central site (box 1). Call this packet p. Suppose such a packet is not currently cached (box 1, arrow F). Then it must be determined whether any logical clock, aside from the logical clock of the requesting node, reads less than the timestamp of the maybe-receive-request (box 2). If none do (box 2, arrow F), then no application packet can possibly be found with a timestamp strictly less than the timestamp of the request which also contains the correct protocol and destination fields. In this case, a correct response to the requesting node is a cannot-be-satisfied packet (box 3). The central site will report its intention to send a cannot-be-satisfied packet to the requesting node (box 4). The user is given a chance to respond to this intention (boxes 5, 6 and 7 - see section 3.5).

If some logical clock exists which reads less than the timestamp of the request (box 2, arrow T) then it is possible that some process at this node will yet spawn a packet to satisfy the requirements in box 1. The central site does not yet know whether this will occur. Thus the process that is attempting to find a correct response to the maybe-receive-request must wait for some new status to arise which will allow it to make a decision (box 8).

At any given time, there are various processes at the central site in states of suspension, waiting for conditions to change so that they may determine the correct response to the request they were created to serve. The central site wakes up all of these processes whenever an updated logical time value is received for some logical clock or whenever a new application packet arrives. Each process will recommence its search for a reply. Perhaps now the correct response can be determined. If not, a process will return to the suspended state awaiting further application packets or logical clock updates. This algorithm is continued until a correct response can be found.

Now, suppose that packet p is found (box 1, arrow T). We ask if any logical clock, aside from the clock at the node of the requesting process, reads less than the timestamp of p (box 9). If not, then p must be the earliest packet capable of satisfying the request (box 9, arrow F; box 10). The central site informs the user of its intention to return p to the requesting node (box 11). The user responds to this intention (boxes 12, 13 and 14).

Finally, suppose that p is found and there does exist a logical clock reading a time less than this packet's timestamp (box 9, arrow T). Then it is possible that some process at this node will spawn a packet which can satisfy the request possessing a timestamp less than the timestamp of p. Since the central site cannot determine at this time whether such a packet will be created, the servicing process must wait for a

new status to arise (box 15).

The algorithm for a receive-request is somewhat more complicated (see figure 3.5).

The application packet with correct protocol and destination fields that possesses a timestamp less than that of the request plus t, greater than that of the last handler-creation timestamp for that protocol and destination, and less than the timestamp of all other such packets, is selected by the central site (box 1). Call this packet p. If not present (box 1, arrow F), we ask if any other logical clocks read less than the request timestamp plus t (box 2). If not (box 2, arrow F), the only processes capable of creating a packet to satisfy the request are those yet to execute between the request time and the request time plus t at the requesting node (box 3). A conditional-execute packet with timestamp equal to the request time plus t (the time the requesting process will time out) is therefore sent by the central site in reply (box 4). This will be responded to with a give-me-now packet when the requesting node reaches the logical time specified by the conditional-execute. However, if before this, some application packet possessing correct protocol and destination is indeed spawned by one of the processes at that node (box 5, arrow T), then this is the packet to satisfy the request (box 11). This is reported to the user (boxes 12, 13, 14 and 15). If no such application packet arrives before the give-me-now (box 5, arrow F; box 6), then the request cannot be satisfied (box 7). This is reported to the user (boxes 8, 9 and 10).

Receive Request
B
1
Application packet p cached with:
1. correct protocol
2. correct destination
3. timestamp that is:
a) < request timestamp + t
b) > last handler-creation timestamp for this protocol and destination
c) < timestamp of any other such packet?
F
T
2
Any other logical clock < request timestamp + t?
17
Any other logical clock < timestamp (p)?
F
T
3
request cannot be satisfied by any process at any other node
16
wait for new status
18
timestamp (p) > request timestamp?
37
wait for new status
B
F
T
19
p satisfies request
4
send conditional-execute packet with timestamp = request timestamp + t
20
report to user
24
send conditional-exacute packet with timestamp = timestamp (p)
5
Any arriving application packet r with:
1. correct protocol
2. correct destination
3. source identical to destination
4. timestamp < request timestamp + t?
21
user permission to send?
E
F
T
22
obey user command
23
send p
F
T
C
D

Figure 3.5

C
6
give-me-now packet arrives
7
report to user request cannot be satisfied
8
user permission to send?
F
T
9
obey user command
10
send cannot-be-satisfied packet
D
11
P satisfies request
12
report to user
13
user permission to send?
F
T
obey user command
14
send P
15


E
25
Any arriving application packet q with:
1. correct protocol
2. correct destination
3. source identical to destination
4. timestamp (q) < timestamp (p)?
F
T
give-me-now packet arrives
26
p satisfies request
27
report to user
28
user permission to send?
29
F
T
obey user command
30
send P
31
q satisfies request
32
report to user
33
user permission to send?
34
F
T
obey user command
35
send q
36

If there is a logical clock reading less than the request timestamp plus t (box 2, arrow T), then a reply cannot yet be determined. A new status must be awaited (box 16).

If p does indeed exist (box 1, arrow T), then it is determined whether some other logical clock possesses a timestamp less than the timestamp of this packet (box 17). If not (box 17, arrow F), we ask if the timestamp of p is greater than the timestamp of the request (box 18). If not (box 18, arrow F), p has been determined to satisfy the request (box 19). This is reported to the user (boxes 20, 21, 22 and 23). If so (box 18, arrow T), a conditional-execute packet is sent to the requesting node indicating that it must execute up to the logical time given by the timestamp of packet p (box 24). If some satisfying application packet arrives from that node before the ensuing Give-me-now (box 25, arrow T), this is the packet to satisfy the request (box 52). This is reported to the user (boxes 33, 34, 35 and 36). If the Give-me-now packet arrives first (box 26), packet p satisfies the request (box 27). This is reported to the user (boxes 28, 29, 30 and 31).

Finally, if p exists and there is a logical clock reading less than the timestamp of this packet (box 17, arrow T), then a new status must be awaited (box 37).

Notice that no reporting to the user is done until such time as the central site has determined the correct reply to the request.

Also notice that whenever a node executes conditionally

up through a specified logical time, it is possible that the application processes at that node will spawn packet requests (for differing protocol types, as only one protocol type can be requested at a time), rather than packets. This serves to complicate the central site request handling mechanism. However, it presents no new conceptual difficulties, and we will not discuss this further.

### 3.4.3 Node Suspension and Logical Clock Maintenance

In chapter two we motivated the need for node suspension and logical clocks. We now discuss how both are implemented in our debugging facility.

A node's logical clock advances in real time whenever application processes at that node are executing. A logical clock ceases to advance whenever the nub causes a node suspension to occur. Node suspension prevents the execution of application processes because, in effect, the nub seizes complete control of the processor.

Node suspension occurs at a node whenever the nub needs to communicate with the central site and some acknowledgement of this communication is required. Node suspension terminates upon receipt of a valid reply from the central site.

We now list those occasions upon which node suspension commences and terminates:

| | | |
|---|---|---|
| 1. | Commences: | Upon internet packege creation. |
| | Terminates: | Upon receiving an unconditional-execute packet from the central site (see section 7.4.1 for more details). |
| 2. | Commences: | Upon sending an application packet spawned at that node to its destination node (from which it is rerouted to the central site). |
| | Terminates: | Upon receiving an acknowledgement of receipt of that packet from the central site. |
| 3. | Commences: | Upon transmitting a receive-request or maybe-receive-request packet on behalf of some application process. |
| | Terminates: | Upon receiving from the central site in reply, an application packet, a conditional-execute packet, or a cannot-be-satisfied packet (see section 7.4.2 for more details). |
| 4. | Commences: | Upon reaching the local time value indicated in a conditional-execute packet and sending a give-me-now to the central site. |
| | Terminates: | Upon receiving from the central site in reply, an application packet, a cannot-be-satisfied packet, or another conditional-execute packet (see sections 7.4.2 and 7.4.4 for further details). |
| 5. | Commences: | Upon sending a handler-creation packet. |
| | Terminates: | Upon acknowledgement of receipt of the handler-creation packet by the central site. |
| 6. | Commences: | Upon sending a package-destroyed packet. |
| | Terminates: | Upon acknowledgement of receipt of the package-destroyed packet by the central site (whereupon the nub at the node ceases to exist). |

Now we examine how node suspension is accomplished.

The nub possesses a hook into each Internet procedure which, upon being invoked by some application process, requires some kind of interaction with the central site. The first action performed by the nub in every case is to save the

current value on the node's logical clock. Then the nub searches all PSB's (recall chapter one) to find all processes at priority one (low priority) that are waiting on a (not disabled) condition variable. The timeout field in each such PSB is saved and then set to zero. In other words, the time-out is disabled. The net effect of this is that all priority one processes waiting on some condition will not wake up while the node is suspended. The nub accomplishes all this in a way that guarantees it will not be interrupted by any other process (regardless of priority) existing at that node.

Now the nub causes the invoking application process to wait until a response is received from the central site. The nub wakes up the looper, a special high priority process which possesses no function except to execute an infinite loop to prevent any application processes (at low priority) from acquiring the processor. The looper periodically yields the processor to other processes at the same priority and can be preempted by processes at a higher priority. This allows other high priority nub processes to execute (as well as processes handling packet reception) but effectively locks out all application processes. By this means, node suspension is achieved.

We point out that the implementation guarantees that the looper is indeed waiting on its condition variable when it is notified to begin execution. If this were not the case, the notifying signal would be lost and the looper would not grab immediate control, perhaps allowing the execution of application processes while a node suspension was supposed to be in effect.

The looper continues to loop (hence, node suspension is in effect) until such time as the nub receives a valid reply from the central site. When this occurs, the looping process is notified. It will determine the amount of time node suspension was in effect by subtracting the current time on the logical clock from the logical clock value saved by the nub at the start of this suspension. It will then restore the timeout field in the PSB of each priority one process that was disabled by adding the node suspension time (adjusted to the units of the hardware timeout clock) to the original saved timeout value. It will then restore to the logical clock, the saved logical time that was first read when node suspension commenced. Finally, it will cause the original interrupted application process to regain the processor.

The net effect of all this is that node suspension is rendered invisible to the application processes. Logical time has not advanced. All application processes waiting on condition variables have not noticed any passage of real time. The interrupted application process is handed back control of the processor at the point of interruption. The ordering of processes on the ready list has not been altered. No user data has been touched. In short, upon relinquishment of the processor, the looper leaves the state of the application in the exact same state it found it when node suspension commenced.

Incidentally, we stated that only priority one processes are locked out by the looper and that only priority one processes have their timeout fields adjusted. Processes with

priority higher than one (e.g. the processes controlling the keyboard and disk) are not affected. This may alter the relative order of processor acquisition between high and low priority processes, causing the node suspension to be not quite transparent to the application.

This cannot be helped, however. We take the position that a high priority process has received that priority because of a desire to insure that it will execute a particular minimum number of times in some time interval, regardless of how long a particular application process attempts to control the processor (this is why, in the Mesa system, application processes are expected, for the most part, to execute at priority one). Furthermore, any system process at priority one is not guaranteed to execute any minimum number of times in some interval because program correctness must in no way depend upon a process yielding the processor within a certain length of time. Thus we feel that (1) we may suspend priority one processes indefinitely and expect no adverse effect on the application program, and (2) we may not suspend processes with priorities greater than one at all, since these processes evidently must execute with a certain minimum frequency. These two statements may not always be true, but they are reasonable in most cases. They imply, then, that when using this debugging facility, all application processes must be at priority one. This requirement is not particularly difficult to satisfy.

### 3.4.4 Deadlocks

Two kinds of deadlocks may arise in the use of the debugging facility, causing a premature abortion of the debugging session. One kind arises due to problems with the application program. These are deadlocks that would have arisen regardless of the presence of the debugging facility. They ought to be seen when the debugging facility is in use, and need not concern us at all.

The second kind is somewhat more troublesome. Deadlocks may arise due to the debugging facility mechanism. If they are not taken care of, they will prevent the debugging of that part of the application yet to execute when the deadlock occurs.

Deadlocks arise when all participating application nodes are in states of suspension because some application process at each node has performed a receive-request or a maybe-receive-request. As long as at least one application node is not suspended, then the application execution is making progress, and there is no deadlock. Deadlocks arise because the debugging facility suspends the entire node whenever a single application process at that node requests a packet. Obviously, this does not occur when the application is executing by itself.

We present two simple examples of deadlock. The first (see figure 3.6) occurs when some application process at each node requests a packet that will be sent at a later time by some other application process at the same node. Since each node is suspended when the request is done, the subsequent

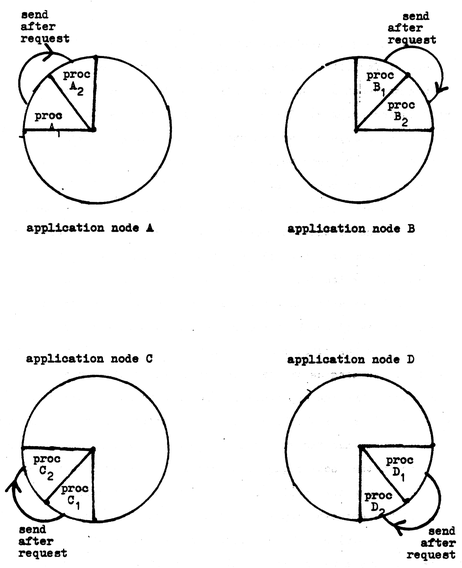

send after request
proc A2
proc A1
application node A
send after request
proc B1
proc B2
application node B
application node C
proc C2
proc C1
send after request
application node D
proc D1
proc D2
send after request

Figure 3.6

send at each node will never have a chance to be performed. This deadlock is called the send-to-self problem.

A more general form of this problem (see figure 3.7) arises when each node expects to receive a packet from some other node, forming a circular request chain, and each node will not send a packet until it has received one. Each node says to the other, "After you!" and nothing ever gets done. This is called the circular-send problem.

When the central site perceives a deadlock, it attempts to "unwind" it in the following fashion. It sends a conditional-execute packet to the node possessing the logical clock at the earliest logical time. The timestamp sent in this packet is the time of the next earliest node's logical clock. The receiving node is then free to execute up to this logical time. During this execution, it is possible that some application packet will be spawned to satisfy some requesting node, or that logical time will advance to enable the central site to perceive a correct response to some outstanding request. In either case, the deadlock is broken.

If neither of these possibilities comes to pass, however, the situation becomes just a bit more sticky. Now two logical clocks read the same minimum time. The central site chooses one of these, and sends a conditional-execute packet to that node indicating that it may execute for one logical tick. If the deadlock is still not broken, the central site transmits an identical packet to the other node. This alternation continues until either the deadlock is broken or until both

application node A
application node B
send message
send message
send message
application node C
send message
application node D

Figure 3.7

logical clocks have reached the time on the third minimum logical clock. At this point alternation continues among all three nodes. The pattern continues until the deadlock is broken.

If the deadlock is not broken no matter how long this algorithm continues, then it is possible that the deadlock has been caused by the application process itself. However, the central site never decides this conclusively, and it is up to the user to abort the session when he runs out of patience.

Incidentally, we point out that when a node executes conditionally, it may, rather then break the deadlock, simply spawn another request for packet! This further complicates the deadlock handling mechanism at the central site. However, it adds no new conceptual difficulties, and we do not discuss it further.

### 3.4.5 Termination

From the description of the deadlock handler, it is clear that the debugging facility will always cause progress to be made in the execution of the application. Thus, if the application itself terminates, so will the debugging session, provided the user has enough patience. The central site will conclude the session upon receiving package-destroyed packets from all participating application nodes.

The only problem that may arise here is caused by listener processes (see chapter five) that never destroy the internet

package but unceasingly monitor the communications lines for packets. If the application to be debugged contains a listener, then the central site can never determine that the session has indeed come to a close (unless, of course, it could somehow be appraised that all other application processes have been destroyed). In this case, it is up to the user to terminate the session when he is through.

## 3.5 User Interface

The debugging facility provides a fairly simple interface to the user to permit both monitoring and debugging of the application to take place. We discuss both of these possibilities in this section.

### 3.5.1 Monitoring

When the central site is about to send an application packet back to a node in response to a receive-request packet, a maybe-receive-request packet, or a follow up give-me-now packet (or as soon as it has decided that the request or give-me-now is unsatisfiable), it reports this to the user via the Alto screen. These are the only events which the facility is capable of reporting.

Each time an application packet is about to be sent by the central site to the requesting nub, the following information is reported to the user:

1. A special identifier assigned to that packet by the central site to which the user may refer at any time until this packet is discarded. This interface identifier is in no way related to the real identifier of the packet as assigned by the application process which spawned it.
2. The real identifier of the packet.
3. The internet address (in octal) of the packet's source node.
4. The internet address (in octal) of the packet's destination node.

5. The protocol number of the packet.

6. The number of requests by the application process which has requested this packet (e.g. this is the $n^{th}$ packet request from that process).

7. Whether this request was through a receive-request packet or a maybe-receive-request packet.

Each time a cannot-be-satisfied packet is about to be sent by the central site to the requesting nub, the following information is reported to the user:

1. The internet address (in octal) of the node at which the requesting application process resides.

2. The protocol number of packets which the requesting process is willing to accept.

3. The number of requests by that application process (e.g. this is the nth packet request from that process).

4. Whether this request was through a receive-request packet or a maybe-receive-request packet.

The reporting of this information allows the user to monitor all interprocess communications via message passing that occurs during the execution of the application.

### 3.5.2 Debugging (User Commands)

The user is given the opportunity to respond whenever an event is reported in the manner described above. He has a number of commands at his disposal for such response. We wish to emphasize that events are reported to the user by the central site before they actually occur. Thus, the user is able to debug his application because he decides whether these

events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
events will automatically be placed in read-only status 7 to 14 days
after data flicker or loss and for view 1only on at start
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup
5.5.2.4 Bus Data Queue Overflow Recovery

If the queue overflows, the oldest data will be discarded
to allow the most recent data to be saved at start .backup

packet or a cannot-be-satisfied indication.

There is no direct way for the user to specify that the application packet should be replaced by a cannot-be-satisfied packet. This is because it is desirable for the user to be aware of all possible application packets that can satisfy a request. It is better for the user to reject all such packets one by one, then to allow a single withhold command to reject all of them. Thus, replacing an application packet by a cannot-be-satisfied packet may only be achieved indirectly by the user issuing the withhold command every time the same request comes up. Eventually, the request must come back as unsatisfiable.

The withhold command can be used to simulate packet loss or to test the code when a particular packet is never sent.

There is no withhold command when the reported request is unsatisfiable.

3.5.2.3 The Replace and Retrieve Commands

If the reported request is satisfiable, but the user wishes to replace the application packet to be sent with a different application packet, he issues the replace command. He is asked to enter the interface identifier of the replacing packet (therefore, the replacing packet must be one that has been reported to the user previously in connection with some other event, and the user must have requested that this packet be saved for future transmission, or he must have replaced

this packet using this same replace command, or he must have delayed this packet - see the next section - or he must have created it - see section 3.5.2.6). If the packet with the indicated interface identifier cannot be found, the user is so informed and no replacement is made. If it can be found, the replaced packet is recached on the disk for future use. If the replacing packet's destination or protocol number is different from the packet being replaced, these will be altered to make the values identical. The user will be informed of this change.

If the user is dissatisfied with his new packet, he may reissue the replace command to obtain yet another one. When he is done issuing replace commands, he may issue a send, a withhold, or a delay (or perhaps even a display or create) command.

If the reported request is unsatisfiable, the user may replace the cannot-be-satisfied packet which would be transmitted by the central site with any application packet of which he is currently aware. Since no application packet is actually being replaced, the user issues the retrieve command instead. Subsequent to this, the replace command may be issued as many times as desired.

3.5.2.4 The Delay Command

If the reported request is satisfiable, but the user wishes to delay the requesting application node's receipt of

the packet, he issues the delay command. He is then asked to enter a delay interval value. This delay value (after suitable units conversion) is added to the timestamp of the application packet. The packet is then reoached on the disk for future use. The central site will fork a new process with a function of finding a new packet to satisfy the request. If the delay time is small, the very same packet may be found. If the delay is large, some other packet may be found or it may be determined that the request is now unsatisfiable. Thus, at some later time, the same event may be reported again, with the same or different application packet or a cannot-be-satisfied indication.

The delay command may be used to simulate packet transmission delays due to hardware malfunctions.

There is no delay command when the reported request is unsatisfiable.

3.5.2.5 The Display Command

The user may at any time display the contents of the application packet that is to be sent in response to the current reported request. He may display any header field or the packet body. The display command is issuable whenever such a packet is present (e.g. even after a replace or retrieve command has been given). The display is in octal.

There is, of course, no display command when the request is unsatisfiable (unless a retrieve command has been issued).

3.5.2.6 The Create Command

The user may at any time create a new application packet. He is asked to enter all necessary header fields as well as the packet body. This is all done in octal. The central site will make the packet, report the interface identifier assigned to that packet, and cache it on disk for future use. The user may then employ this packet in a subsequent replace or retrieve command.

3.5.2.7 The Call Debugger Command

The user possesses a rudimentary ability to cause some application node to be placed into the Mesa debugger. Upon entry of a call debugger command and the internet address of the desired node, the central site will spawn an enter-debugger packet to be sent to that node. The user may then physically go to that node and debug events occurring there via the Mesa debugger.

3.5.2.8 The Quit Command

The user may at any time enter the quit command, terminating the debugging session.

*This empty page was substituted for a blank page in the original document.*

# Chapter Four

## Correctness and Usefulness of the Debugging Facility

We expect that many of the issues discussed in chapter two and some of the implementation aspects of chapter three are familiar to those with knowledge of simulation techniques. Our debugging facility is merely a simulator of distributed applications which also allows interactive debugging to take place during the simulation. More than this, however, the debugging facility causes a probable simulation to take place. This is a term which will be defined later. Probable simulation, we will find, is closely related to the concept of transparency. However, it is a much weaker condition. As stated in the concluding paragraph of chapter two, complete transparency is an ideal which is unattainable by the debugging facility. Therefore, the next best goal has been opted for, that of probable simulation.

Now that we have presented a detailed description of the design and implementation of our debugging facility, we wish to argue for its correctness and usefulness. This chapter presents the basic ideas of such an argument. At times we proceed somewhat informally, as a strictly rigorous discussion is beyond the scope of this work.

The argument can be broken down into three steps. Lamport (Lamport78) points out that for any system of clocks to be correct, a single condition, termed the clock condition, must hold for that system. Thus, the first question to be asked is,

"Does our debugging facility maintain the clock condition?"

Now a system may obey the clock condition without doing anything particularly useful. For our purposes, the useful goal is that we be able to interactively debug an application, P. The first step towards such usefulness is that the facility simulates that system, P. Our second question, then, is, "Does our debugging facility simulate P?"

However, we will find (in discussing simulation in a later section) that the mere simulation of P may not always be useful. We will show that the debugging facility is useful only when it performs a probable simulation. Therefore, the final question to be posed is, "Does our debugging facility perform a probable simulation of P?"

Question one determines the correctness of the debugging facility. Questions two and three determine its usefulness. A positive answer to all three questions will be motivated in what follows.

## 4.1 Maintenance of the Clock Condition

Lamport (Lamport78) defines a logical clock, $C_i$, for each process, $P_i$, in a system. For an event, b, in process $P_i$, $C_i(b)$ is the time of the event as determined by reading the logical clock, $C_i$. Moreover, C is a global function over all $C_i$ such that $C(b) = C_i(b)$ if b is an event in process $P_i$. The clock condition is as follows:

Clock Condition. For any events a, b:
if $a \rightarrow b$ then $C(a) < C(b)$.

All this means is that if one event "happens"before" another, then the logical clock system should reflect this by recording the former as occurring earlier in logical time than the latter. This would appear to be the most reasonable condition to set on a system that is divorced from real time.

From the definition of the $\rightarrow$ relation (as discussed in chapter two), Lamport states two more conditions that, if true, imply the clock condition:

C1. If a and b are events in process $P_i$, and a comes before b, then $C_i(a) < C_i(b)$.

C2. If a is the sending of a message by process $P_i$ and b is the receipt of that message by process $P_j$, then $C_i(a) < C_j(b)$.

To these we add a third condition, because processes in our system may communicate through monitors as well as through explicit transmission of packets:

C3. If a is the relinquishment of a monitor lock by process $P_i$ and b is the next acquisition of that lock, by process $P_j$, then $C_i(a) \prec C_j(b)$.

It should be easy to see that in a system which allows process communication through both message passing and monitor interactions, conditions C1, C2 and C3 together imply the clock condition. We now show that the implementation described in the previous chapter satisfies these three conditions.

First, remember that, in our system, there does not exist a one-to-one relationship between processes, $P_k$, and clocks, $C_k$. Our implementation allows an arbitrary number of processes to read the same clock. This, however does not make any difference towards the satisfaction of the three conditions.

C1 is the most straightforward. Each process clock, $C_i$, is implemented by a counter that increases monotonically. Thus, in a single process, later events will always occur at greater logical times than earlier events. Of course, it is assumed that the counter "ticks" fast enough so that no two events see the same logical clock value. This may not be physically realizable, but the implications of this appear unimportant.

C2 is the most interesting case. Lamport suggests the following implementation rule to guarantee that C2 holds:

IR2. If event a is the sending of a message m by process $P_i$, then the message m contains a timestamp $T_m = C_i(a)$. Upon receiving a message m, process $P_j$ sets $C_j$ greater than or equal to its present value and greater than $T_m$.

We have not followed Lamport's suggestion. Instead, we achieve C2 in a slightly different fashion. Instead of updating $C_j$ to conform to the timestamp $T_m$, we allow $C_j$ to tick, withholding m from $P_j$ until $C_j > T_m$. This is more in keeping with the spirit of transparency in that the process will not be able to detect whether it is executing in physical or logical time. Using Lamport's method, a process could notice unexplainable jumps of its clock, thereby inferring that it is not executing in real time.

The difference in approach is actually a very interesting point. It arises because the problem Lamport is trying to solve is only one part of the problem we are trying to solve. Lamport is attempting to produce correct timing behavior in the execution of any system of distributed processes. We are attempting to reproduce the causal relationships between events that would have occurred had the debugging facility not been present while simultaneously maintaining this correct timing behavior. The transparency issue has modified our approach.

Finally, condition C3 is satisfied by the simple expedient of having all processes that can interact with a monitor M read the same logical clock $C_i$. This is easy to do since the processes residing at a particular node form a natural subset for this purpose. That is, all processes at a node may interact with any monitor module at that node, but may not interact with any monitor modules at any foreign nodes. Furthermore, all processes with access to a particular monitor share the same memory and, hence, are able to read the same

logical clock.

Then, by the semantics of the monitor lock construct, and by virtue of the fact that each logical clock is implemented as a monotonically increasing counter, condition C3 is found to hold.

It is admitted that the assignment of a single clock to all processes residing at a node is somewhat artificial. For the dissatisfied reader, we will discuss, in chapter five, a possible alternative debugging facility design that assigns a unique logical clock to each process in the system. This was not implemented because of the difficulty in maintaining the correct logical time on each logical clock.

In conclusion, having shown that conditions C1, C2, and C3 are all satisfied, we may state that the debugging facility implementation obeys the clock condition.

## 4.2 Proof of Simulation

The first step in determining whether the debugging facility simulates the process system P is to come to a clear definition of simulation. In order to do this, however, we must first introduce the notion of a history array. Our conception of a history array is a slight modification of the history arrays discussed in Van Horn's thesis (Van Horn66).

During the course of the execution of a particular computation, information is constantly being written to the various objects (variables and data structures) involved in the application. Imagine an array (see figure 4.1), to be called the history array, in which there exists a unique row for each object and the $i^{th}$ element of each row contains the information written by the $i^{th}$ write to that row's object. The $0^{th}$ element of each row is considered to house the initial state of the row's object. (For the sake of consistency, we draw a distinction between the creation of an object and the first write to that object. The value assigned to an object at its creation is entered into the $0^{th}$ column of the proper row. The value of the next write to that object, if any, is entered into the first column of the same row. Certain objects may already exist at the commencement of the computation; hence they are not created during the computation. An object of this class is handled by placing its initial value into the $0^{th}$ column of the proper row and the value of the first write, if any, to the object into the first column of the same row.) As

HISTORY ARRAY




Figure 4.1

(based on (Van Horn66), fig. 4.1(c))

execution proceeds the array enlarges since new values are added to the end of each row as new writes occur to that row's object during the computation. Furthermore, at any time the array may possess a jagged right edge (in other words, the number of elements in each row is not necessarily the same) since the number of writes to each object may be independent of the number of writes to any other object. Each row represents the complete history of an object during the computation. (A row of this array is similar in concept to the object history of Reed (Reed79).) The array, as a whole, specifies the complete behavior of the executed computation.

This definition of a history array differs from that proposed by Van Horn in two respects. First, a row exists only for each object involved in the computation. In Van Horn's scheme, a row exists for each "cell" in the machine. Without going into detail about exactly what a cell is, we simply note that cells include all memory words in the maching, as well as other, more esoteric constructs. We, however, are not interested in the values of all the memory cells in the machine. Many of them will possess histories having no importance to the computation in question. As a computation progresses, an observer is interested in determining the values of only, say, x items. To us this implies that there are exactly x objects involved in the computation. Thus, there are exactly x rows in that computation's history array.

Second, the $0^{th}$ column of the history array as defined by Van Horn is identical to $S_R$, where $S_R$ is the initial state of

the run $R = \langle S_R, T_R \rangle$ corresponding to the computation that is about to commence. In our scheme, it is obvious that the $0^{th}$ column may contain values that arise after the computation has started executing, as new objects are created.

Now we present a definition of simulation.

> Definition: The behavior of a set of processes P is simulated by a set of processes Q just when an execution of any possible computation of Q (that is, an execution of any possible run R of the system Q - recall chapter two) produces a history array that is either identical to or contains the history array produced by the execution of some possible computation c of P.
>
> By "contains" we mean that the history array produced by R possesses all of the rows of the history array produced by c (with, of course, the identical number of elements in each row and the identical values for each element) plus other rows denoting the histories of objects absent from the history array produced by c.
>
> One may speak of the simulation of the computation c by Q when the execution of a particular computation of Q produces a history array which either contains, or is identical to, the history array produced by P during the computation c.

One consequence of this definition of simulation is that the process system Q may be substituted for the process system P and this will be invisible to an observer who is unaware that the substitution has occurred. An observer who is aware that a simulation is taking place is interested in, and can determine, the histories of the set Z of z objects involved in the simulation. This set possesses a (possibly proper) subset X with cardinality x ($x <= z$) containing all objects involved in the simulated computation. An observer who believes himself to be witnessing the execution of his system, and not a simulation thereof, will be interested in, and able to determine,

only the histories of objects in the set X. To him, the histories of the objects in the set Z - X are meaningless state values for which he has no use or concern. Furthermore, this uninformed observer will be able to construct some computation c of P which could have produced the resulting history array of the objects in set X. Thus, he is made to believe that he has, in fact, observed the computation c of his system of processes P.

Notice that this definition of simulation does not at all imply that the probability of Q simulating a particular computation c is in any way related to the probability of c occurring when the system P runs by itself. Thus, the unaware observer may perceive highly unlikely behavior when a simulation is taking place, but he will be unable to state conclusively that he is indeed watching a simulation. This is an issue we will discuss at some length in the next section.

Now it will be proven that a simple condition placed on the set of processes Q is sufficient (although not necessary) to guarantee that an execution of any computation of Q will yield a simulation of some computation c of P. Hence, the condition implies that Q simulates P.

Simulation Condition: A process system Q containing q processes will always simulate a process system P containing p processes if:

1) $q \geq p$

2) p processes can be chosen from Q such that each process has the same <u>functionality</u> as some distinct process in P (that is, a one-to-one functionality correspondence exists between the processes of P and the p processes chosen from Q). Call this set of processes with

cardinality p, set A.

3) the remaining q - p processes of Q never write any object read or written by the p processes chosen in condition two. Call this set of processes with cardinality q - p, set B.

Thus, A $\cup$ B equals the process system Q and A $\cap$ B is the null set.

The term "functionality", as used above, requires definition. The functionality of a process signifies what that process will "do" when presented with a system state, S, upon acquiring the processor. In other words, given a history array (representing the history of the computation up to a point), the functionality of a process determines how that history array will be altered (enlarged) during the course of the execution of that process and how the history array will appear upon relinquishment of the processor by that process.

It is possible to speak of the functionality of a process, because processes, consisting of a single sequence of events, execute in a deterministic manner. Systems of processes, as discussed in chapter two, do not execute deterministically, hence it is meaningless to refer to their "functionality".

---

Theorem: If a system of processes Q obeys the simulation condition towards a system of processes P, then Q simulates P.

---

Once the above assertion has been proven, it will be shown that the debugging facility is a system of processes Q which obeys the simulation condition towards a system of

processes P where P is the application being debugged. This implies that the debugging facility does indeed simulate the application P.

Before the proof can be presented, however, we must provide three more definitions, two of them notational.

For convenience, we define the function H(R) to represent the history array resulting from the execution of a run R defining some computation c.

We also introduce the concept of a **prefix run**. Given a run R = $\langle S_R, T_R \rangle$ where the transition sequence $T_R$ contains n elements (each element being a set of process names), a prefix run of R, is defined to be any run of the form P = $\langle S_P, T_P \rangle$ where $S_P = S_R$ and the transition sequence $T_P$ contains m elements such that $0 <= m <= n$ and these m elements are identical to the first m elements of the transition sequence $T_R$. In other words, run P is either identical to run R or is an aborted version of run R.

Finally, the notation $R_m$ is defined to be the prefix run of run R with transition sequence of length m ($0 <= m <= n$, n being the number of elements in the transition sequence of R).

Now for the proof, which proceeds by induction on the transition sequence of the run of an arbitrary computation of Q.

Proof: Let V be the run of some possible computation of Q such that $V = \langle s_0, \tau_0 \rangle$. $\tau_0$, of course, consists of the (possibly empty) sequence $\tau_0, \tau_1, \ldots, \tau_n$ where each $\tau_i$ ($0 <= i <= n$) is the set of all processes in acquisition of the processors during the time interval $[i, i+1)$ (recall chapter two).

The induction is performed over i. In other words, it proceeds over the successively longer prefix runs of run V of the arbitrary computation.

Initially: i = 0

$H(V_0)$ represents the state of all objects (in S, not in X - the sets Z and I have been previously defined) already in existence at the time of commencement of the computation with run $V_0$. No row in the array possesses more than one element.

It is easy to see that the computation with run $V_0$ simulates a computation of P with run $W = \langle s_W, \tau_W \rangle$ such that $s_W = s_V$ and $\tau_W$ is an empty transition sequence. This is because $H(V_0)$ is either identical to or contains $H(W)$. Thus, the simulated computation c of P is that computation with run W. The computation of Q with run $V_0$ simulates c. (In fact, the computation of Q with run $V_0$ may simulate other possible computations of P, those where $s_W \neq s_V$ but the objects in X possess the same values in $s_W$, as they do in $s_V$. However, we are concerned with the existence of only one computation c and do not worry about these others.)

Inductive Hypothesis: i = m, $0 \leq m < n$

Assume the computation of Q with run $V_m$ simulates some computation c of P with run W''. That is, $H(V_m)$ either contains or is identical to H(W'').

Given this, it must now be shown that the computation of Q with run $V_{m+1}$ simulates some computation c of P with run W'''. That is, $H(V_{m+1})$ must be proven to contain or be identical to H(W'''). Thus:

Prove for i = m + 1, $0 < m + 1 \leq n$

$T_{m+1}$ (the last element in the transition sequence of the run $V_{m+1}$ and the only element of that transition sequence not to appear in the transition sequence of $V_m$) is a set containing j processes ($0 \leq j \leq$ the number of processors involved in the execution of the system Q). Of these, k belong to set A (defined in part two of the simulation condition) and j - k belong to set B (defined in part three of the simulation condition). Since A and B are disjoint, these two groups are also disjoint.

Accordingly, the next section of the proof is divided into two parts:

a) Consideration of the effects on $H(V_m)$ by the execution of the j - k processes in $T_{m+1}$ belonging to the set B of the simulation condition.

b) Consideration of the effects on $H(V_m)$ by the execution of the k processes in $T_{m+1}$ belonging to the set A of the simulation condition.

a) The j - k processes do not write any of the objects read or written by the processes in set A. Furthermore, the processes of set A possess a one-to-one functionality correspondence with the p processes of the system P. Thus, it is clear that the j - k processes do not write any of the objects read or written by the processes of P. Therefore, only objects which are never read or written by the processes of P are written by the j - k processes. Objects which are never read or written by any process in P are, it stands to reason, absent from H(W''). Thus, the only effect these j - k processes can possibly have on $H(V_m)$ is to add values to those rows which are absent from H(W''). Thus, the history array resulting subsequent to the execution of these j - k processes will contain, or be identical to, H(W''') where W''' = W''. Therefore, the simulated computation o of P is that computation with run W''' = W''.

b) The processes of set A possess a one-to-one functionality correspondence with the p processes of the system P. Therefore, the k processes possess a one-to-one functionality correspondence with a subset G of the processes of P, having cardinality k. Then the execution of the k processes has an effect on $H(V_m)$ which is identical to the effect on H(W'') produced by the execution of the subset G. Thus, the history array resulting subsequent to the execution of these k processes will contain, or be identical to, H(W''') where W''' is a run such that W'' is the greatest prefix run of W''' not equal to W''', itself, and the last element of $T_{W'''}$, contains the subset G

just delineated. Therefore, the simulated computation c of P is that computation with run W''' as specified.

At first glance it would appear that we are implying that the functionality of both systems of k processes are identical. This, of course, contradicts what was stated earlier, namely that it is meaningless to talk about the functionality of a system of processes because of stochastic effects that cause nondeterminacy. However, we get around this by considering any $T_i$ to represent the set of processes in execution during an interval $[i, i + 1)$ which is sufficiently small so that stochastic variables, such as processor speed, do not have a chance to affect the computation.

Alternatively, we can say that the simulated computation c of P is that which arises when the stochastic processes during the interval represented by the last element of $T_{W'''}$, and the stochastic processes during the interval represented by $T_{m+1}$, affect the causality relationships between events in the k executing processes (in either P or Q) in identical ways.

*

We have shown thus far that some possible computation c of P is simulated when $T_{m+1}$ consists of <u>either</u> the j - k processes of part a) <u>or</u> the k processes of part b). It needs merely to be shown that $T_{m+1}$ may consist of both sets of processes simultaneously, since that is what we originally hypothesized $T_{m+1}$ to be. This is easy to show. But one further coruscation and we are home.

The requirement that the j - k processes of Q never write

any object read or written by the k processes of Q implies that the existence of the j - k processes is invisible to the k processes. Thus, the functionality of the k processes is not affected by the j - k processes. This, in turn, implies that the j - k processes may coexist in execution time with the k processes without affecting the alteration of $H(V_m)$ by any of the latter. The resulting $H(V_{m+1})$ will then still be identical to, or contain, H(W'''). Thus, $T_{m+1}$ may consist of the sum of both the set of k processes and the set of j - k processes. The simulated computation c of P is that computation with run W''' as defined in part b), above. The computation of Q with run $V_{m+1}$ simulates c.

We have shown that any prefix run of V will simulate some computation c of P. Since V was a run of an arbitrary computation, we have that any computation of Q simulates some computation of P. Thus, Q simulates P.

QED

---

Moreover, given a particular computation of Q, with run V, it is not difficult to determine what computation c of P has been simulated. If the $i^{th}$ element in the transition sequence of V contains d processes from the set A, then the simulated computation c possesses a run $R = \langle S_R, T_R \rangle$ where $S_R = S_V$ and the $i^{th}$ element in the transition sequence of R consists of the d processes of P having the one-to-one functionality correspondence with those processes. We point out that it is

possible that this computation with run V may also simulate some other computation of P. However, this is not assured, and is immaterial since we only wish to know that one such computation c exists.

Now it is quite easy to show that the implementation of the debugging facility obeys the simulation condition with regard to the application being debugged. In other words, the debugging facility is comparable to the system of processes Q, while the application being debugged is comparable to the system of processes P. This is most easily shown by examining, in turn, the three parts of the simulation condition:

1) The implementation of the debugging facility consists of processes at the central site and processes at each debugger nub along with the processes of the application being debugged. Thus, $q \geq p$ (in fact, $q > p$).

2) The processes of the application are not modified in any way by the presence of the central site and debugger nub processes. That is, the events defining each application process and the order in which these events occur are not altered. It is obvious, then, that these processes possess a one-to-one functionality relationship with themselves, hence they form the set A, as stipulated in the simulation condition.

3) It is the job of the central site and debugger nub processes to maintain their invisibility towards the application processes. It is obvious, from the implementation description in chapter three, that they do not write any objects read or written by the latter group. In fact, when they (the nub processes, anyway) relinquish a processor, they attempt to restore the exact machine state they observed upon acquiring that processor. Thus, these processes form the set B, as stipulated in the simulation condition.

1), 2) and 3) taken together imply what we have set out to prove. Thus, we state that the debugging facility simulates the application to be debugged.

## 4.3 Probable Simulation

As stated earlier, the knowledge that the debugging facility simulates an application is not enough to feel assured of its usefulness as a tool in debugging that application. This is because it is possible for the debugging facility to repeatedly simulate computations that would almost certainly never occur in real use of the application. The determination that lurking bugs are absent from certain improbable computations alone would not be sufficient to assure correctness of a practical application.

In chapter two, we considered the execution order of a set of processes at an unsuspended node in the face of the suspension of another node where communication streams were open between the nodes. We stated that during a normal execution (that is, without node suspension) the execution sequence at the unsuspended node was I Q I Q . . . With node suspension, the execution sequence was along the lines of I I I . . . I Q I I . . . I Q I I . . .

In the Alto/Mesa environment, one major design goal is that all processes of the same priority have an equal opportunity to acquire the processor. Thus, in this environment, we would classify the computation corresponding to the first execution sequence I Q I Q . . . as a <u>probable computation</u>, one which we would not be particularly surprised to observe. Moreover, since the second sequence I I I . . . would appear to go against the grain of this design goal, we classify the corresponding computation as an <u>improbable computation</u>.

We must point out that this discussion can only be appreciated on an intuitive basis. We cannot draw a clear distinction between probable and improbable computations. There is no definite demarcation between the two. We can, however, establish a correlation, of a sort, between improbable computations and the notion of system failure.

As Lamport has pointed out, ". . . the entire concept of failure is only meaningful in the context of physical time. Without physical time, there is no way to distinguish a failed process from one which is just pausing between events." (Lamport78) We may consider such a "pausing between events" to take place when a process relinquishes the processor to allow the execution of other processes at that node. In general, improbable computations (at least in this system, and probably in many others) are marked by the unusually swift "pause between events" of some processes and the unusually lengthy "pause between events" of others. This leads to a higher than normal failure perception rate by the former set of processes for two reasons. First, the interval between packet arrivals from the "long pause" processes is greatly increased, proportionately increasing the chances that a "short pause" process will mistakenly perceive a failure when there is none. Second, the "short pause" processes execute many times for each single execution of a "long pause" process. If the "short pause" processes base their failure perceptions not simply on elapsed time, but on the number of times a particular variable is checked for a certain condition (this, in turn, is

actually based on elapsed physical time - so it does not contradict Lamport's assertion that failure is based solely on physical time), then it is likely that the number of checks will be exceeded before the "long pause" process can make the condition true. Again, failure perception is likely to occur.

To be more concrete, consider again the example of chapter two. We said that process Q had a communication stream open with a process on another node. Process I, on the same node as process Q, was to make sure that this stream functioned correctly and was to close the connection if it perceived a failure. If the process with which Q was conversing was of the "long pause" type, it caused an improbable computation, with execution order I I I . . . I Q I I I . . ., to occur at Q's node. I made >z checks of a monitor variable, and, finding no effect on this variable by Q, closed the connection. The causal chain of events was thus: use of the debugging facility causing a "long pause" process to arise causing an improbable computation to occur at Q's node causing I to make >z checks on some data before Q could affect that data causing I to perceive failure causing the premature closing of the stream.

In short, to repeat what was said in chapter two, failures occur because the "real time expectations" of processes are not met during an improbable computation. We state, without proof, that the more improbable the computation, the more likely the chance of a perception of failure.

It should be pointed out that the occurrence of failures depends on the semantics of the application in question. In

our discussion of lurking bugs, in chapter two, we asked whether the computation with process execution order A B B C A B B C . . . was correct? We now know that, in the absence of bugs, it is meaningless to talk about a computation's correctness. All possible computations are "correct". We can only talk about a computation's probability (or improbability) of occurrence or whether it will produce a failure; the latter is determined by semantics. For example, the programmer may decide that two consecutive executions of B ought to be considered a failure and wirte code to print out an error message when this occurs, or write code to abort the computation, or write code containing certain tests to make sure that B will not read the same value twice. Alternatively, he may decide that the results of the execution are not made incorrect by two consecutive executions of B. It all depends on how the programmer attaches meaning to his application.

Finally, we state that there are varying degrees of failure severity. The premature closing of a communication stream is usually, but not always, a severe failure. Some other failure caused action may not be as severe (as, for example, printing out a message as opposed to aborting a computation, as discussed above). Thus, the set of improbable computations may be considered to house a subset of computations, termed undesirable computations - those that lead to severe failures due to the improbability of their corresponding runs.

In this work, it is the task of the debugging facility to

produce a simulation of a probable computation to act as the foundation upon which debugging is performed. The user may then alter the communication streams as he is inclined, to produce other computations of varying degrees of probability in order to detect lurking bugs. This would seem to be the most reasonable approach in designing a debugging tool for distributed environments.

The notion of a probable computation is, again, somewhat intuitive. It is a computation one would not be surprised to observe in a particular system. Its form depends on many parameters - hardware characteristics, transmission medium, distance between nodes, the particular dispatcher algorithm in use, to name a few. For example, a dispatcher that favored certain processes over others would generate computations with certain characteristics. The set of probable computations for this system would reflect this. Moreover, the substitution of a new dispatcher in the same system would yield a different set of probable computations. Again, the distance between nodes has an effect on the delay time between packet transmission and reception which, in turn, may create computations with particular characteristics. These are reflected in that system's set of probable computations.

We speak of a <u>set</u> of probable computations. For complex systems with many independent processes, the number of probable computations may be quite large. Thus, the question arises, "Which probable computation (of this set) is the debugging facility attempting to simulate?"

The goal of the debugging facility, when a debugging session is started up at time t with machine state S, is to attempt to simulate the computation c that would have arisen beginning at time t with machine state S, if the application had been executing without the debugging facility.

We must stress the intuitiveness (again) of the notion of a computation which "would have arisen". Given an initial machine state it is, of course, impossible to determine what computation will arise due to the inherent nondeterminacy of parallel processing. Moreover, if the application commences execution at time t under control of the debugging facility, then one cannot tell which computation would have arisen had execution commenced at time t without the debugging facility. Thus, the computation c, above, is only a hypothetical, but useful, idea. In short, it is possible to attempt to simulate a computation without actually knowing what that computation is.

This particular computation, c, has been chosen to be simulated for two reasons. First, the computation c is one which it is possible for the debugging facility to simulate. In the previous section, we proved that the facility will simulate at least those computations with runs possessing initial states identical to the initial state at which simulation commences. Since both the simulation computation and the hypothetical computation c begin at time t, it is obvious that the facility is capable of simulating c.

Second, we postulate that the probability of this computation, c, being a probable computation is high. Thus, it

is reasonable to expect that the facility is simulating a probable computation. This may not always be true (for example, during and after time t the communications medium may be experiencing unusually heavy traffic leading to unusually lengthy transmission delays) and may conceivably lead to problems. However, we feel that it is too much to ask of the debugging facility to create probable computations under improbable conditions. The development of a tool to handle this ought to provide an intriguing area for future research.

We have stated that the goal of the debugging facility is to attempt to simulate c. Is it actually able to do this? Unfortunately, the answer is no. The mere existence of the debugging facility will have an effect on the system causing a different computation, c', to be simulated rather than c. The facility affects the application both spatially and temporally. It has a spatial effect by altering the layout of the application code in memory, perhaps forcing some code to disk that would have remained in main memory. Hence, a resulting fetch to disk may occur that would not have occurred had the debugging facility not been present. This can alter the computation that is performed. Also, the debugging facility code requires a finite amount of time to execute. Hence there is a temporal effect in that any portion of the application code will execute at time t + x rather than at time t + y with $x > y$. Furthermore, as execution continues, application code will be executing later and later than it

would have had the debugging facility not been present. The consequence of this is that stochastic processes (of the kind mentioned in chapter two) will be in different states at time t + x than they would have been at time t + y, having different effects than they would have and possible causing a different computation to be performed.

Let us be more concrete about this by again examining the disk. One stochastic process involved in the disk operation is how long it will take (seek time) to access a particular disk location. Suppose process A requested a disk fetch of that location, waiting to be notified by the high priority disk controlling process, D. Then process B began to execute, during the course of which process C was notified (placed on the ready list). Now, without the debugging facility, the request by A would have occurred at time t + a, and the disk head would have been very near the location to be accessed. Thus, D would have retrieved the contents of the requested location and notified A, taking the processor away from B before B could notify C. Then A would be placed on the ready list before C. On the other hand, when the debugging facility is present, the request by A occurred at time t + b (b > a). At this time, the disk head was very far from the location to be accessed. Thus, when B began executing it was able to notify C before being preempted by D. Thus C was placed on the ready list before A. A new set of causal relationships ensued, hence a different computation, c', was performed instead of the original computation c.

In light of all this, we can say that the facility simulates the computation c that would have arisen at time t up through the point of execution where its first spatial or temporal effect is made known to the application. If all stochastic processes could be controlled throughout the entire execution, then c could be simulated completely. Van Horn (Van Horn66) discusses this possibility at some length. When stochastic processes are not controlled completely, the user loses precise control (as discussed in chapter two) over the events that occur during the debugging session. Interprocess communications are then governed not only by explicit user commands, but also by implicit side effects caused by such stochastic processes. In our example, the user is able to control precisely only the events of the computation c', which are the events of the original computation c as they have been altered by stochastic processes.

Having shown that c' is simulated rather than c, we ask whether c' is a probable computation? If so, then the third question posed at the beginning of this chapter is answered in the affirmative, and we have proven all that we set out to prove.

Remember that we have defined the probability of a computation in terms of failure, or the lack thereof. This, in turn, was shown to be related to the disparity between "pause intervals between events" among the different processes in the computation. But a process can only be made aware of the pause interval of another process by the time it takes to

receive successive communications from that process. The timestamp mechanism assures that this interval is (for the system in question) a reasonable one in logical time for communications that proceed by message passing (we obtain "reasonable" intervals between successive communications by ensuring that, if an average transmission delay time between two nodes is x seconds, then the timestamp of a packet sent from one of these nodes to the other will equal the logical time of the sending node when the packet is actually sent plus x seconds plus or minus $\xi$, where the value of $\xi$ depends on stochastic processes within the communications hardware - see the timestamping mechanism described in chapter three - and is usually much less than x). We note that these stochastically dependent timestamps represent those that would have been assigned in the computation c', not in the computation c. For communications that proceed by monitor interactions, reasonable intervals are maintained by assigning a single logical clock to all processes that can access the same monitor.

Thus, each process has its "real time expectations" reasonably well fulfilled by every other process. All communications are seen to proceed reasonably in time. Therefore, c' is a probable computation (we state again, though, that c' is probable to the extent that all stochastic processes within the system possess probable values during the course of the debugging session). Without the timestamping mechanism, the computation that would be simulated, with

messages experiencing transmission delays of minutes or hours, is of an extremely low probability.

## 4.4 Probable Simulation vs. Transparency

In the previous section, we introduced two computations, c and c', to make clear the difference between probable simulation and transparency. If the debugging facility were able to simulate the computation c, then the goal of complete transparency would be achieved. To answer the question posed at the very end of chapter two, then, this computation, c, is that entity towards which we have attempted to maintain transparency.

We have shown, however, that spatial and temporal effects, as well as stochastic processes, prevent the realization of complete transparency. We are able only to simulate c', a probable computation. Probable simulation is, as stated, weaker than transparency because c' is not the computation that would have arisen at time t, c is. Thus, the debugging facility is simulating the "wrong" probable computation. We feel, however, that the computation c' is sufficiently "similar" to the computation c (we state this without proof and ask the reader to accept the notion of "similarity" on an intuitive basis) so that the facility is still quite worthwhile despite this shortcoming.

*This empty page was substituted for a blank page in the original document.*

# Chapter Five

## Related Ideas and Suggestions for Further Research

In this final chapter we discuss some of the shortcomings, problems and generally interesting aspects of the implementation presented in chapter three. We also discuss some of the possible ways in which the research reported here can be extended. We touch on certain features that we did not have time to implement, refused to implement because of a firm belief that they were incorrect, or simply could not figure out how to implement. Issues in all three areas are, of course, open to the reader for examination. We hope that this chapter will stimulate interest in further research in debugging techniques for distributed systems. The field, as we shall see, is by no means exhausted.

## 5.1 Fragmentation

The Internet Protocol definition provides for the passage of large datagrams through networks that are not equipped to handle such sizes by the method of fragmentation. Fragmentation consists of the splitting up of a large packet into several smaller packets at the gateway entering the network, and the reconstruction of the original datagram from these packets at the gateway exiting the network.

Our debugging facility currently operates at the datagram rather than the fragment level. That is, the user is not made aware, and has no control over, the flow of fragments during interprocess communications. We have considered fragments to be below the level at which the user ought to be concerned. However, it is conceded that the ability to debug at the fragment level may at times prove useful and a debugging facility with this extended power might make a reasonable research project.

The reason for the datagram rather than fragment orientation lies in the concept behind the timestamping mechanism. We assign a timestamp only when the entire packet has arrived and the application process is about to be so notified by lower level internet processes. The assignment of a timestamp to each fragment would necessitate moving "deeper" into the code. A fragment timestamp would represent the time at which some internet process was first notified by yet a lower level mechanism that a fragment had arrived. This is, of course,

possible to implement, but it was deemed advisable to maintain the hook into the debugging facility at as high a level as possible, rather than deep inside the internet implementation.

## 5.2 Bottlenecking

It would seem reasonable that a debugging facility which allows the user to simulate all kinds of error conditions such as losing packets, causing packets to arrive out of order, etc, would also provide a way to simulate bottlenecking. Bottlenecking occurs when some portion of the transmission medium experiences more traffic than it can handle. Since bottlenecking is often a real danger, especially in complicated systems with many concurrently executing applications, a user would probably be interested in determining the reaction of his application to such artificially induced conditions.

It is interesting to point out that our debugging facility does not allow bottlenecking to be simulated. This is because a user is permitted only to determine what packet is to be received by a particular request for packet from some process. He is not allowed to send packets indiscriminately when such requests do not exist. In particular, he has no means at his disposal to flood the network in order to create bottlenecks.

We do not consider this to be a shortcoming of our system. The realm of the debugging facility extends over the functionality of an application, not of the communications hardware. Insofar as the functionality, or lack thereof, of the hardware affects the application itself, then bottlenecking ought to be an issue for us. That is as far as we go. To be more concrete, bottlenecking, while conceivably affecting the communications hardware in a number of adverse ways, has the same net effect

on the application as losing a group of packets (either through physical loss by the hardware or by packet buffer overflow at some node). Losing packets is something the user can indeed simulate via the debugging facility. Hence, the need to create bottlenecks is obviated. However, the design of some kind of tool to debug hardware, working in tandem with our debugging facility, might prove useful in certain cases.

## 5.3 Order of Event Reporting

To enable monitoring of the program being debugged, conventional debugging tools report various events to the user. These debuggers report items such as instruction traces or state transitions of user specified program objects, among, perhaps, others. Our facility reports events related to inter-process communications. Specifically, it informs the user of each request for a packet by any application process in the system and discloses the result of that request. That is, it tells whether the request is satisfiable and, if so, which application packet is to be sent in response.

It is implicitly understood in most cases that when conventional debuggers report events to users in a particular sequence, that sequence represents the order of occurrence of those events in real time. For example, an instruction trace represents the order of execution, in real time, of a set of instructions by the processor.

It ought to be clear, however, that our facility, being divorced from real time, has some difficulty in complying with this implicit assumption. In particular, the interface reports an occurrence of a request for packet (an "event" in our system) as soon as the correct response to that request is determined. This is in no way related to the real time order in which such requests are rendered. In fact, it is also in no way related to the system logical time order in which such requests are rendered. (By system logical time, we

are referring to Lamport's function C, a global function over all logical clocks in the system such that $C(b) = C_i(b)$ if b is an event in process i which reads logical clock $C_i$.)

One improvement that could be added to the user interface, then, is to cause events to be reported to the user chronologically with respect to this function C. The central site could delay reporting a request until all logical clocks have exceeded that request's timestamp. Then the user is sure that he is made aware of events in the order in which they occur in logical time.

One interesting consequence of this is that if event a is reported to the user before event b (implying $C(a) < C(b)$), it is not necessarily true that event a is capable of causally affecting event b ($a \not\rightarrow b$). In other words, a and b may still be concurrent. As Lamport has correctly pointed out, the converse of the clock condition is not necessarily true. That is:

Clock Condition Converse: For any events a, b:
if $C(a) < C(b)$ then $a \rightarrow b$

does not necessarily hold.

A debugging tool which could make causal relationships clear to the user would involve complicated mechanisms well beyond the scope of this research. It is debatable whether the information gained would be worth the time spent in constructing such a tool. This might make an interesting area for future research.

## 5.4 The Multi-Application Problem

Lauer and Needham (Lauer78) discuss two distinct approaches in the design and implementation of operating systems. These two approaches have been termed message-oriented and procedure-oriented. Any operating system can be placed into either category based on how it views the concepts of process and synchronization. These alternate views greatly affect the way in which the notion of an application is regarded in that system. "Process" and "application" are terms which we have used extensively thus far.

Procedure-oriented systems are marked by the sharing of data between processes, which is controlled by synchronization mechanisms such as monitors. In these systems, processes change contexts for data access through procedure invocations, ". . . which can take a process very rapidly from one context to another. . . A process typically has only one goal or task, but it wanders all over the system (by means of calling procedures to enter different contexts) in order to get that thing done. As a result, the system resources tend to be encoded in common or global data structures and the applications are associated with processes whose needs are encoded in calls to system-provided procedures which access this data." (Lauer78)

Message-oriented systems are characterized by, of course, message passing for interprocess communication. In these systems, processes are resource guardians. "Each process tends to operate in a relatively static context. Virtual memories or address spaces are usually placed in one-to-one correspondence with

processes. Processes rarely cross protection boundaries (except to briefly enter the executive or kernel), and they rarely share data in memory. As a result, processes tend to be associated with system resources, and the needs of applications which the system exists to serve are encoded into data to be passed around in messages." (Lauer78)

What is important here is the relationship between processes and applications in the two systems. In procedure-oriented systems, this relationship is tight in that a process, or group of cooperating processes, can be clearly seen as representing a particular application. In message-oriented systems, however, processes are bound to resources, not applications. Thus, a single process may concurrently service the needs of many distinct applications. We show why this leads to difficulties for our debugging facility.

It ought to be clear that distributed systems are, of necessity closer to message-oriented than procedure-oriented environments. This is because it is, in general, impossible (except for processes having the good fortune to reside at the same node) for processes to communicate through shared data. The system on which our debugging facility is implemented is message-oriented. It contains processes designated as listeners. These listeners are, as mentioned above, the processes which control resources. They are constantly sensing the network for resource requests from any application and then servicing those requests (or, at least, handing them down to internal processes for servicing). An example of a listener is the process existing at a file server which handles requests for

internode file transfers.

Program writers consider these listeners to be a given part of the system (almost like the hardware) and write their code to correctly interface with them. Since they are assumed to function correctly, the user is not at all concerned with debugging them. It would be nice if the user could simply install the application (which interacts with some listener) on some set of nodes and begin debugging right away. Unfortunately, he cannot do this. This is because any process involved in the application (including the listener) must be suspendable by the debugging facility. If the listener is suspended (made to run slower) then the performance of all other applications in the system interacting with it will be degraded significantly, usually intolerably. The net effect is that all users monitoring their private applications and unaware that some user is currently debugging his own application will notice inexplicable delays due to the slowdown of the listener. This is a consequence of the fact that processes, in a message-oriented facility, may simultaneously "belong to" (interact with) more than one application. Thus, we refer to this as the multi-application problem.

Currently, of course, the user is forced to bring up his own private copy of the listener on some private node. This is not always possible, as the user may not possess access to the listener code, may not understand the code even if he does, and (for example, in the case of the file server) may not be able to duplicate necessary conditions on his private node for

the correct execution of the listener process. This is a tremendous liability which, because our implementation is so heavily dependent on the notion of node suspension, we have not been able to solve. A facility which allows the user to simply "plug in" his application and start debugging right away would make an extremely worthwhile project for future research.

## 5.5 Controlling Monitor Entries

Our debugging facility allows the user to create many different computations of his application in order to test each of these for lurking bugs. However, the set of such computations is only a proper subset of the set of all possible computations of the application. Thus, there are sets of causal relationships that it is beyond the power of the user to test.

In particular, the user is not given the ability to specify or alter the order in which processes enter monitor modules. This entry order is decided within the system itself, partially by the dispatcher, partially by process priorities, partially by the algorithm in use to determine the next process to acquire a monitor lock, partially by stochastic processes which affect interprocess timing relationships, and, perhaps, partially by yet other indirect causes. The user is able to influence the order of monitor entries only indirectly by influencing the order in which the processes in question receive packets prior to acquiring the monitor lock. That is, if two processes both receive a packet and then attempt to enter the same monitor, the user can affect the entry order by delaying the packet to one of the processes. However, this "feature" is merely a side effect that cannot be counted on. Nor is the scenario which gives rise to it guaranteed, or even likely, to occur.

Yet we have seen the duality between the two communications methods - message passing and monitor interactions - and it

may seem somewhat artificial to limit the user's ability to alter the former but not the latter. We regard monitor entry as being akin to packet reception. Both consist of the acquisition of an ability by a process to observe a data state created and left by another process. Likewise, exiting a monitor and packet transmission are dual concepts since both consist of relinquishing a data state constructed by a process for the purpose of making it available to another process for examination. In fact, there appears to be no semantic difference between the two types of communication. The only difference we note is in the method - any process at a node may examine the state of a newly relinquished monitor while it is usually the case in message passing that communication channels exist only between specified pairs of processes. Of course, this difference is easily eliminated through the use of a "mailbox", where a process sends a packet to a particular node's mailbox (some previously determined memory area) which can be picked up by any process at that node willing to accept it. Mailboxes and monitor modules appear to be identical concepts.

(Incidentally, Lauer and Needham (Lauer78) attempt to make a case for the duality of operating systems based on these two types of communication mechanisms. They drew parallels between various constructs in the two systems. Much to our chagrin, however, they do not draw parallels between monitor entries and exits and packet receptions and transmissions. All we can say is that, for our purposes, the comparisons we have

drawn are much more useful than those presented in that paper.)

It is probably not too difficult to implement a mechanism that would halt a process whenever it tried to enter a monitor (similar to halting a process when it attempted to receive a packet) and reporting this attempt to the user. Probably, since user processes may enter both user implemented and system monitor modules, entry into the latter would not be reported by the facility as it would require the user to have extensive knowledge of the underlying system. Such information would be (to use a term coined by Model) "below the grain" of the environment under investigation. In this way, the user could control the sequence of all interprocess communications (he is given the ability to alter any of the wavy arrows in Lamport's diagram, figure 2.1). He could create any possible set of causal relationships, hence simulate any possible computation of his application. The design and implementation of a tool to accomplish this probably represents a worthwhile area for future investigation.

But how such a tool might be implemented is not so clear. It would imply the ability to suspend a single process (delay it from entering a monitor) while allowing other processes at the same node to continue executing. This would appear to render invalid the use of a single logical clock for all processes at the node. Each process would need to have its own private logical clock since the suspension of one process would be independent of the suspension of any other at that node. Then an algorithm similar to that used for packet

reception might be employed for monitor entry, namely:

1. recording the logical time at which the process desires to enter the monitor by reading its logical clock.
2. determining whether all other process clocks at that node have gone beyond this logical time.
3. if not, suspending the process until such time as this becomes true.
4. if so, determining whether the monitor is currently locked by some other process (e.g. the parallel to determining whether there is a packet ready to be received.)
5. if the monitor is not currently locked, reporting this entry attempt to the user and waiting for his reply.
6. whatever the state of the lock at this time, the process attempts to acquire it while its logical clock ticks (akin to a receive call with disabled timeout).

It must be pointed out, however, that our use of logical clocks was solely for the purpose of maintaining transparency towards the application. We wanted to simulate a probable computation as a basis upon which debugging could be performed. In the case discussed here, logical clocks would be used for the same purpose. However, after much thought, we have not been able to devise a reasonable method of assigning to and advancing logical clocks when there exist multiple clocks at each node (perhaps the reader would like to try his hand at this). Thus, we are not sure whether logical clocks would prove useful in this case.

We present a simple example to show some of the intricacies involved in such a scheme. The central difficulty is that the maintenance of transparency necessitates a view of logical time such that the logical clock of a process is considered to

advance whether or not that process is actually executing (as long as it has not been artificially suspended by the debugging facility). This is the method employed in our implementation.

Now, suppose that two processes, A and B, are residing at the same node. A is currently executing; B is on the ready list. Logical time is advancing for both processes. Suppose process A wishes to enter a monitor. This event is duly reported to the user who decides to delay A's entry until after process B has entered that same monitor. Therefore A is suspended (at logical time x) and B starts to execute. Now the question is, "What time do we assign to B's logical clock?" More precisely, since it does not matter (for our purposes) what time B sees until it tries to enter a monitor or receive a packet, what time is assigned to B's very next attempt to perform one of these two actions? In the interest of transparency, B should not be aware that A has been artificially suspended. Thus, at the outset of B's execution, B's clock should read x plus however long A would have executed had it not been suspended. But, of course, it is impossible at this time for the facility to know how long that would have been. Thus, the difficulty in assigning a reasonable time to B's logical clock is apparent. It is easy to see how more complicated execution patterns would render logical clock maintenance by the debugger facility virtually impossible.

An alternative approach would be to abandon logical time altogether and let the user be responsible for creating probable computations. Then, transparency would no longer be a goal of the implementation and debugging would entail a

sequence of decisions about which process ought to enter a monitor next, or which process ought to receive a packet next. The user would possess total control in determining which computation is performed. Total control, of course, brings with it a tremendous amount of detail for the user to cope with. The user becomes responsible for deciding all matters pertaining to interprocess timing relationships, both at a single node and among separate nodes. As such, he must be intimately familiar with the code he is attempting to debug, if he is to debug intelligently. Coping with detail is a significant research problem in itself.

## 5.6 Future User Interface

The interface presented to the user by the central site is currently of the form of a "glass teletype" and is somewhat primitive. The facility presents information to the user by printing out lines of text. Likewise, the user controls the debugging session by typing in lines of text. Since the Alto possesses powerful I/O hardware and software facilities, there is room for a good deal of improvement in this area. We see this as yet another worthwhile subject for future research.

The interface reports two kinds of entities, events and data. Events, which are defined to be requests by any process to receive a packet, are reported sequentially to the user by listing various pertinent information such as the node on which the requesting process resides, whether the request is satisfiable and, if so, the identity of the satisfying packet, and the process from which that packet originated. Data, which consist of the contents (header and body) of packets, are likewise reported in a simple fashion. The display is of the form of a sequence of octal values representing each word in the packet. There are a number of ways by which this interface can be improved upon.

### 5.6.1 Multistepping and Slow Stepping

Model (Model79) has discussed in detail a number of worthwhile attributes concerning information display for interactive debugging. As he has pointed out, one failing of many conventional debuggers is that they report too much information to

the user. The user is either forced to discard much of it, or is overwhelmed by it. The former is wasteful, the latter catastrophic. Our implementation currently is also guilty of this failing. All events (as we have defined events) are reported to the user. Since many dozens (hundreds, or even thousands) of packets may be transmitted and received during fairly simple transactions (e.g. a simple file transfer), it seems clear that the user will not wish to be made aware of all of them.

Even more debilitating, not only is the user informed of each pending event, but he is asked to make a decision about each one. This mode of operation is called single stepping; a pause occurs between each step (event) and the user is given the opportunity for analysis. This can prove excruciatingly slow when each individual event accomplishes very little.

An enhancement on this is the concept of multistepping, where only selected events are reported for user observation and analysis. The events to be reported are selected either by the system or the user. The user might instruct the system to suppress the reporting of the next x requests from process y or node z, all requests arising in the next w (logical) seconds, all requests for packets with protocol u, etc. In this way, unimportant events are easily filtered out and debugging can proceed more swiftly.

Incidentally, Model states that the entity constituting a "single step" is not always obvious. For example, in Algol, ". . . should the notion be defined in terms of single lines

of code, statements which do not contain other statements, or individual operations in the language, such as function calls and arithmetical operators?" (Model79) This ambiguity arises because the concept of an "event" is not well defined. We do not have this problem because of our precise (although not necessarily optimal) definition of what constitutes an event.

Somewhere in between single and multistepping lies the notion of slow stepping. This can be employed when the user desires to be informed of all events in a certain class (as in single stepping) but does not want to make decisions about each one (as in multistepping). Thus, the emphasis is on monitoring rather than debugging. The user ought to be able to specify how swiftly events are to be reported and should be able to adjust this rate at will. The ideal interface would allow intermixing of single, multi, and slow stepping during different stages of the same debugging session.

### 5.6.2 Graphical and Analogical Display of Data

One of the central themes espoused by Model is that information ought to be presented, if possible, in a graphical or analogical fashion. The hypothesis is that pictorial displays are more swiftly and easily understood than sequences of symbols (such as numbers). Thus an iteration variable ought to be presented as a kind of "percent-done" indicator (see figure 5.1) representing how much headway has been made thus far. This is an example of an analogical display. Data

Figure 5.1

(based on (Myers80), fig. 2.1)

structures ought to be displayed in the way they are intuitively understood; the interrelationships between the various values comprising the structure ought to be clearly marked. Thus, a Lisp list should be displayed with pointers (see figure 5.2) rather than as a sequence of octal values which the user must fathom for himself.

Myers (Myers80) has implemented a system, called Incense, for displaying graphically and analogically the data structures of a program during interactive debugging. It possesses the ability to display both predefined and user defined structures. We feel that a powerful debugging tool would result from the combination of an Incense-like facility and the facility described herein. Incense could probably be modified without too much effort to display the data structures and other information that make up datagrams. One simple approach is to use the packet protocol number as a convention for determining how to display the packet information. Since packets sent sequentially from the same process may represent different portions of the same data structure (as in the transfer of a file), perhaps a way could be devised to graphically display groups of packets to build more complete diagrams. This concept could be used in conjunction with slow or multistepping where the user could indicate that he wishes to see the contents of the next x packets to be sent from process y, etc.

In short, since a picture is worth a thousand words (and probably even more octal digits) and since many of the user's debugging decisions will be based on the contents of particular

The list ( A ( C ( E F ) D ) B )

Figure 5.2

packets, analogic or graphic display of the contents of these packets should allow debugging to proceed more swiftly and easily.

5.6.3 Dynamic Display of Events

Having discussed some possible ways to display data, we now turn to future methods for displaying events. Since processes are made up of events, we may consider the totality of all reportable events in our system to represent a kind of communication "process" (not at all like a Mesa process, of course). Model has stated that in order to fully appreciate the functionality of a process, one must view it as a flow of events, a movie as it were, rather than as a series of snapshots of states arising from the execution of those events. As it is currently constituted, our interface only displays the communications process as an isolated sequence of events.

A more dynamic, movie-like display providing a graphic representation of the communications process might prove quite worthwhile. Such a display would have certain fixed areas set aside on the screen to represent the various nodes involved in the debugging session. The transmission of a packet could be indicated by a dot flowing from the sending to the receiving node. The user could focus his attention by examining particular parts of the screen containing the nodes in which he is currently interested. Thus, the interface might look not unlike an air-traffic controller's screen (this is not a

facetious comparison; just as the air-traffic controller directs the path of airplanes, so the user directs the path of packets).

The advantage of such a dynamic approach is that it gives the user a "feel" for certain aspects of the communications process which it would be difficult or impossible to derive from a more static, sequential reporting of isolated events. In particular, during slow stepping the user could learn where communications are most extensive, where bottlenecks are most likely to occur, and which nodes are busiest at what times. These concepts could be inferred from a more static approach, but only with great difficulty.

Of necessity, however, a complicated display such as this would require most of the memory of an Alto, leaving little room for the central debugger site code. One solution is for the user to do his monitoring from two Altos placed in close proximity. One could display the more advanced interface, and the other could have the simple interface of chapter three, with, perhaps, Incense-like display capabilities. The user would enter his commands at the latter site. Coordination between these two monitoring stations would proceed through message passing. Thus, a user command issued at one node would be reflected by the user interface of the other node. This configuration bears similarity to the network concept of Metric, mentioned in chapter one. We do not speculate on how easily such an implementation could be realized.

We point out that the network concept is made necessary

only by the small size and present performance capabilities of each Alto. There is no inherent reason why the two displays could not be handled by a single, more powerful processor.

Incidentally, such a dynamic display would still suffer from an inability to make clear the causality relationships among the events it reported.

## 5.7 Towards an Integrated Debugging System for Distributed Computational Environments

The reverse of the problem of the debugging facility reporting too much information to the user (as discussed in section 5.6.1) is the danger that it will report too little information. Requests for packets are but one class of event, and a small class at that. It may prove difficult for the user to detect many kinds of lurking bugs based solely on knowledge of the communications process. He may need a method of getting at those system events that occur "between" communication events. We are speaking, of course, of traditional events such as assignment, arithmetic processing, etc, which make up the bulk of most processes and which are performed privately by the process in which they occur without the need for any interprocess communication.

We have already spoken (see chapter two) of how the user can be made aware of such events under the current implementation. After monitoring communications through some point in the execution, he may abandon the central site and physically go to the node at which reside the processes containing the private events in which he is interested. At this node he is able to monitor events using the conventional single node debugger existing there (however, that debugger may need to be modified in order to maintain accurately the logical clock existing there by accounting for the correct flow of logical time while the debugging takes place). Debugging can continue in this fashion at this node until the next attempt at

interprocess communication via message passing. At this point, in order to maintain transparency, control must be relinquished to the central debugger site. The user may then, if he so chooses, abandon this node and physically go to another node using the conventional debugger existing there to monitor events private to that node. This can be repeated for all nodes in the session.

We also said that the user may choose to employ some remote debugging or remote monitoring system (Teleswat, for example). For large networks, the distance between nodes would make a remote debugger imperative. Such a debugger would allow the user to perform all of his debugging directly from the central site. This would involve the ability to interrogate and to issue commands to remote nodes from the central site. Issues of node autonomy may come into play in this area.

Tailoring a remote debugger to the environment presented by the debugging facility described herein would be a profitable pursuit. The resulting system would constitute a totally integrated facility for debugging distributed applications. All pertinent events could be monitored and debugged from a central area, possessing total control over the proceedings. Combined with some of the other ideas in this chapter, it would make for an extraordinarily powerful debugging tool.

## References


(Bryant77) Bryant, R.E., "Simulation of Packet Communication Architecture Computer Systems", S.M. thesis, M.I.T. Laboratory for Computer Science Technical Report TR-188, November 1977.

(Canon80) Canon, M.D., Fritz, D.H., Howard, J.H., Howell, T.D., Mitoma, M.F. and Rodriguez-Rosell, J., " A Virtual Machine Emulator for Performance Evaluation", CACM 23, 2, February 1980.

(Hoare74) Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", CACM 17, 10, October 1974.

(ISI80) "DOD Standard Internet Protocol", Information Sciences Institute (University of Southern California) RFC #760 IEN #128, January 1980.

(Jaffe79) Jaffe, J.A., "Parallel Computation: Synchronization, Scheduling, and Schemes", Ph.D. thesis, M.I.T. Laboratory for Computer Science Technical Report TR-231, August 1979.

(Johnson75) Johnson, P.R. and Thomas, R.H., "The Maintenance of Duplicate Databases", Arpanet NWG/RFC #677, January 1975.

(Lamport78) Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", CACM 21, 7, July 1978.

(Lampson80) Lampson, B.W. and Redell, D.D., "Experience With Processes and Monitors in Mesa", CACM 23, 2, February 1980.

(Lauer78) Lauer, H.C. and Needham, R.M., "On the Duality of Operating System Structures", Second International Symposium on Operating Systems, IRIA, Rocquencourt, France, October 1978.

(McDaniel77) McDaniel, G., "Metric: A Kernel Instrumentation System for Distributed Environments", Proceedings of the Sixth Symposium on Operating Systems Principles, November 1977.

(Metcalfe76) Metcalfe, R.M. and Boggs, D.R., "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM 19, 7, July 1976.

(Mitchell79) Mitchell, J.G., Maybury, W. and Sweet, R., "Mesa Language Manual", Version 5.0, Xerox Palo Alto Research Center, Report CSL-79-3, April 1979.

(Model79) Model, M.L., "Monitoring System Behavior in a Complex Computational Environment", Stanford Ph.D. thesis available as Xerox Palo Alto Research Center Report CSL-79-1, January 1979.

(Myers80) Myers, B.A., "Displaying Data Structures for Interactive Debugging", M.I.T. S.M. thesis available as Xerox Palo Alto Research Center Report CSL-80-7, June 1980.

(Pouzin78) Pouzin, L. and Zimmermann, H., "A Tutorial on Protocols", Proceedings of the IEEE 66, 11, November 1978.

(Reed79) Reed, D.P., "Implementing Atomic Actions on Decentralized Data", Preprints for the Seventh Symposium on Operating Systems Principles, Pacific Grove, California, December 1979.

(Svobodova79) Svobodova, L., Liskov, B. and Clark, D., "Distributed Computer Systems: Structure and Semantics", M.I.T. Laboratory for Computer Science Technical Report TR-215, March 1979.

(Swinehart74) Swinehart, D.C., "COPILOT: A Multiple Process Approach to Interactive Programming Systems", Stanford Ph.D. thesis available as SAIL Memo AIM-230 and CSD Report STAN-CS-74-412, July 1974.

(Teitelman77) Teitelman, W., "A Display Oriented Programmer's Assistant", Xerox Palo Alto Research Center Report CSL-77-3, March 1977.

(Van Horn66) Van Horn, E.C., "Computer Design for Asynchronously Reproducible Multiprocessing", Ph.D. thesis, M.I.T. Project MAC Technical Report TR-34, November 1966.

(Xerox79a) "Alto: A Personal Computer System Hardware Manual", Xerox Palo Alto Research Center, May 1979.

(Xerox79b) "Mesa System Documentation", Version 5.0, Xerox Palo Alto Research Center, April 1979.

(Xerox79c) "Alto Subsystems", Xerox Palo Alto Research Center, October 1979.

**CS-TR Scanning Project**
**Document Control Form**

Date : 8/18/95

**Report #** LCS-TR-364

Each of the following should be identified by a checkmark:
Originating Department:

- ☐ Artificial Intellegence Laboratory (AI)
- ☒ Laboratory for Computer Science (LCS)

Document Type:

- ☒ Technical Report (TR)
- ☐ Technical Memo (TM)
- ☐ Other:_____

## Document Information

Number of pages: 192(198-images)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
- ☐ Single-sided or
- ☒ Double-sided

Intended to be printed as :
- ☐ Single-sided or
- ☒ Double-sided

Print type:
- ☐ Typewriter
- ☒ Offset Press
- ☐ Laser Print
- ☐ InkJet Printer
- ☐ Unknown
- ☐ Other:_____

Check each if included with document:
- ☐ DOD Form
- ☐ Funding Agent Form
- ☒ Cover Page
- ☐ Spine
- ☒ Printers Notes
- ☐ Photo negatives
- ☐ Other: _____

Page Data:

Blank Pages(by page number): FOLLOW TITLE PAGE, 2, 3, 126, 157

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

| | Description : | Page Number: |
|---|---|---|
| (A) | IMAGE MAP: (1-6) UNNUMBERED TITLE PAGES, BLANK, 2, BLK, 3, BLK | |
| | (7-192) PAGES NUMBERED 4-126, UNNUMBERED BLK, 127-157, UNNUMBERED BLK, 158-187 | |
| | (FIG: 198) SCANNING TARGET, COVER, PRINTERS NOTES, TARGETS (3) | |
| (B) | COVER IS BLUE WITH WHITE TYPE. | |

Scanning Agent Signoff:
Date Received: 8/18/95 Date Scanned: 9/25/95 Date Returned: 9/28/95

Scanning Agent Signature: Michael W. Cobb

## Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 9/25/1995
M.I.T. Libraries
Document Services

darptrgt.wpw Rev. 9/94