611

ORDERING ERRORS IN DISTRIBUTED PROGRAMS

by

AARON JONATHAN GORDON

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1985

ABSTRACT

With processors becoming small and inexpensive, many researchers attempt to decrease program runtime by combining processors into a multicomputer and running programs distributed over these processors. Debugging in a distributed environment is different from debugging on a uniprocessor. On a uniprocessor, the order in which a process's events occur is deterministic. In a distributed environment events occur concurrently on different processors. The order in which events occur cannot be easily determined; a program that works correctly one time may fail subsequently if the timing between processors changes. Traditional methods of debugging (such as putting in print statements and recompiling the program or recompiling the program with a debug flag on) are inadequate since they change the program and therefore change the timing.

For this research, I have investigated distributed program bugs that depend on the relative order between events. These *ordering errors* include events which always occur in the wrong order and events whose order of occurrence is time-dependent. In this research, I characterize these *timing errors* and *misorderings* and show necessary conditions for their occurrence. Using my model of a distributed system, I prove which features can be used in combination to avoid ordering errors. I use these results to make suggestions to those writing distributed programs, developing distributed programming languages and designing

distributed operating systems. I then explain drawbacks to preventing ordering errors and show ways to detect them as they occur. Finally, I describe a tool (called TAP) to aid the programmer in discovering the causes of ordering errors in running programs. I also show that TAP is useful in finding other types of distributed program bugs.

ACKNOWLEDGEMENTS

This research would not have been done without the advice and support of many people. I want to especially thank the community of fellow graduate students with whom I shared the excitement as well as the frustration of graduate school. This group includes friends Bryan Rosenburg, Bill Kalsow, and Ennio Stacchetti with whom a discussion over coffee was often the high point of my day; Toby Lehman for putting up with me in the same office especially during these last few months; Hung-Yang Chang, Yeshayahu Artsy, Cui-Qing Yang and the rest of the Charlotte research group for producing a working distributed operating system and helping me to adapt it to fit my needs; and Michael Scott for suggestions of papers to look at and blazing the trail to find a way to produce a document on the LN01 acceptable to the graduate school.

Additionally, I would like to thank Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel whose paper "Interprocess Communication in Charlotte" I condensed to produce most of section 3.1. Their paper has been submitted to the 1985 International Conference on Parallel Processing.

I want to thank the members of my committee, the readers Raphael Finkel, Bart Miller, and Mary Vernon for their advice and suggestions, and the non-readers Miron Livny and Frank Scarpace for taking the time to serve on the

committee. I want to further thank my advisor Raphael Finkel for the support these last three years and for his patience in helping me translate my writing into English.

I also want to thank the department staff for making my life a little easier. Those deserving special thanks include Sheryl Pomraning and Pat Nehm for typing, photo copying and kind words, and Paul Beebe for listening to my complaints and keeping the machines running.

Most of all I would like to thank my family: my wife, partner, and friend Donna Jo Blake for continued support and encouragement; my son Joshua for demanding so little and providing so much; and my parents for not asking too often "when are you going to get a job?".

Finally, I need to mention that this research was supported in part by NSF grant MCS-8105904 and DARPA grant N0014-82-C-2087.

CONTENTS

Chapter 1

Introduction

The goal of this research is to investigate ordering errors in distributed programs. An *ordering error* is an error caused by the relative order in which events occur. We will concentrate on two types of ordering errors, misorderings and timing errors. Misorderings are events that always occur in the wrong order. Timing errors are time-dependent orderings of events that lead to an error. For both types of ordering errors we will show necessary conditions for their occurrence, methods for their prevention, and methods for their detection. We will then describe the implementation and use of a tool called TAP, similar to a postmortem debugger, which helps the user to discover the causes of ordering errors.

Trying to find the reasons for ordering errors is a recent aspect of debugging. There have been bugs in computer programs as long as there have been programmers. A *bug* is an error in a computer program where the program does not always do what it is supposed to do. To *debug* a computer program is to find and correct bugs in that program. At first, debugging was heuristic. Programmers used the console switches, pored through core dumps, put extra output statements in their program, and simulated the computer with pencil and paper to find the reasons for their problems. Soon came the idea of using an interactive

program, called a *debugger*, to aid in the debugging process. By the late '50s and early '60s there were many such tools [Evan66, Gilm57].

Most debuggers provide similar operations. Debuggers allow the programmer to *peek*, examine a memory location; *poke*, put a value in a memory location; set *breakpoints*, choose a place or time that execution of the program suspends and control returns to the debugger; *patch*, add to or change the instructions of a running program; and *trace*, produce a history of the statements the program has executed.

The first debuggers are considered *low-level debuggers* since they only let the programmer deal with the low-level objects of the program being debugged. These low-level objects include memory locations referenced by memory address and machine-level instructions.

Later *high-level debuggers* allow conversations about the program to refer to elements of the programming language in the programming language. That is, variables can be referred to by name, not just by absolute address, and the debugger can show the actual high-level instructions that are in error, not just the machine-level instructions. High-level debuggers have many traits in common [Balz69, Kuls69, Reis75, Ditz78, Poul78, Hodg80, Card83, Kish83]. These traits include: symbolic data access, instruction and data breakpoints, single stepping, view of the source program, reversible execution, and the ability to call

procedures and functions from the debugger. The compiler helps to implement theses features by producing extra code when the *debug flag* is set. Though no debugger has all of these traits most try to incorporate as many as possible.

A different type of debugger was developed by Balzer [Balz69] to allow the programmer to examine a program after it terminates. For this debugger, a history tape is written while the program executes. This tape contains a record of the events that occurred during the execution of the user's program. Balzer provides tools which analyze the information on the tape to help determine where errors may have occurred.

Another complexity added to programming environments that debuggers have come to handle is multiple processes. The JOVIAL debugger [Pars79], COPILOT [Swin74], and a design by Weber [Webe83] all allow the user to select one or more processes in which to set breakpoints when instructions are shared by multiple processes. COPILOT also allows the user to save the context of a process for later restoration. This facility allows the programmer to test various features from a given point in the program without having to run the program to that point again.

Goal of This Research

With processors becoming small and inexpensive, many researchers attempt to decrease program runtime by combining processors into a multicomputer and

running programs distributed over these processors [Alme80, Fink83a]. Debugging in a distributed environment is different from debugging on a uniprocessor. On a uniprocessor, the order in which a process's events occur is deterministic. In a distributed environment events occur concurrently on different processors. The order in which events occur cannot be easily determined; a program that works correctly one time may fail subsequently if the timing between processors changes. Traditional methods of debugging (such as inserting print statements and recompiling the program or recompiling the program with a debug flag on) are inadequate since they change the program and therefore change the timing.

For this thesis, I have investigated distributed program bugs that depend on the relative order between events. These ***ordering errors*** include events that always occur in the wrong order and events whose order of occurrence is time dependent. In this research, I characterize two types of ordering errors, ***timing errors*** and ***misorderings***, and show necessary conditions for their occurrence. Using my model of a distributed system, I prove which features can be used in combination to avoid ordering errors. I use these results to make suggestions to those writing distributed programs, developing distributed programming languages and designing distributed operating systems. I then explain drawbacks to this prevention and show ways to detect ordering errors as they occur. Finally, I describe a tool (called TAP) to aid the programmer in discovering the

causes of ordering errors in running programs. I also show that TAP is useful in finding certain other types of distributed program bugs, caused by non-ordering errors. The implementation of TAP was done on the Charlotte operating system [Fink83a]. A description of Charlotte is included in chapter 3.

1.1 A Model of Distributed Programming

In this section we will present a model of distributed programming. We will use this model in subsequent chapters to show underlying causes of ordering errors. In this model, a distributed program runs on a multicomputer. The processes that comprise the distributed program communicate through messages; they do not share memory. The following are some formal definitions.

□ Multicomputer:

A *multicomputer* is a collection of computers, each containing a central processing unit and private memory; no memory is shared between computers. The computers are connected by some type of communication hardware such as a token ring.

□ Process:

For a *process* we mean an independent unit of execution comprised of program instructions and data. A process can only access instructions and data in its own address space. Two processes do not share space. We do not prohibit multiple threads of control in a single process.

□ Distributed Program:

A *distributed program* is composed of one or more processes working together on a computation. The processes may or may not be on the same processor. The processes communicate with messages. A *message* is a stream of bytes uninterpreted by the operating system.

□ Communication Subsystem:

The *communication subsystem* is the hardware and software underlying the operating system whose job it is to deliver messages on behalf of processes.

Two processes communicate when one receives a message sent by the other. These messages are sent over a *communication channel.* An important aspect of communication channels is whether they are private or public. A private channel only permits two processes to communicate; a public channel allows more than two processes to communicate.

The sending of a message has five distinct phases.

(1) Send request

(2) Copy message to local communication subsystem

(3) Copy message to remote communication subsystem

(4) Copy message to recipient

(5) Report to sender

In the first phase, the process *requests* that the communication subsystem deliver the message. The process provides the communication subsystem with a buffer containing the message and specifies who is to receive it. The sending process can specify a specific recipient or a set of recipients, each of which is to receive the message (this latter case is called broadcast). We call these two types send(specific) and send(all). We assume a send request occurs at a distinct point in a processes code.

Phases 2 - 4 involve copying the message. In the second phase, the message is copied from a buffer controlled by the requesting process to a buffer controlled by the local communication subsystem. The communication subsystem becomes responsible for the message and goes to work to deliver it. In the third phase, the message is transferred to a buffer controlled by the remote communication subsystem. In the fourth phase, the message is copied into a buffer controlled by the receiving process. All copying is atomic. Both the owner of the buffer being copied from and the owner of the buffer being copied to view the copy as a single action. They will never use a buffer with a message partially copied. Once a message is copied the recipient becomes the party responsible for the message. In an actual implementation of a system some of these phases might be combined.

The sending process can be notified after each of these phases has occurred or there might not be notification until all four phases have finished. This final notification, called *report*, is the fifth and final phase of sending a message. This report is notification that all phases of the send have occurred. The report phase will be discussed in more detail later in this section.

We call the *completion point* of a send the point at which the entire message has been copied out of the requesting process's address space. At this point we say *completion* occurs. The completion point for a send can be at the second, third, or fourth phase depending on the design of the particular system. We will show in Chapter 2 that if the completion point occurs synchronously with another of the sending process's statements, then a certain type of timing error cannot occur.

The receipt of a message has three phases.

(1) Receive request

(2) Copy message to recipient

(3) Report to recipient

The first phase is the request. The requesting process provides the communication subsystem with a buffer to hold the message and specifies from whom to receive it. The receiving process can specify a specific sender or a set of

senders, from any one of which a message must be received. We call these two types receive(specific) and receive(any). We assume a receive request occurs at a distinct point in a processes code. The second phase is the copying of the message into the requesting process's buffer. The third phase is the reporting to the requesting process that the other phases have occurred.

We call the *completion point* of a receive the point at which the message has been copied into the requesting process's address space. The completion point for a receive is at the second phase. We will show in Chapter 2 that, if the completion point occurs synchronously with another of the receiving process's statements, then a certain type of timing error cannot occur. We assume that if a receive can obtain more than one message that it obtains the first qualified message received by the local communication subsystem. We also assume that all send requests from a processor A to a processor B are received in the same order as the send requests are made.

In some systems a process can make a send or receive request and continue processing [Fink81, Arts84, Stro83, Andr82]; the result of the request is reported later. In other systems all phases of a send or a receive occur as an atomic unit [Unit83, Cher83, Lisk83]; the requesting process is suspended until the results of the request are reported. The former case is referred to as *non-blocking* communication. The latter case is referred to as *blocking communication.*

Communication can also be partially blocking. For example, a send request might block the requesting process until phase two (transfer to the local communication subsystem's buffer) or phase three (transfer to the remote communication subsystem's buffer) finishes and then allow the process to proceed.

It is our belief that the use of non-blocking communication will lead to increased parallelism for many applications. With non-blocking communication processes are able to do other work while the communication subsystem is handling the message. One example is a file server process that handles file requests for other processes. After requesting a send of information to one client, the file server can handle requests from other clients without being blocked waiting for the first client to receive the information.

When blocking communication is used the requesting process receives the report implicitly. The fact that the process is unblocked means the requested action has occurred. When non-blocking communication is used there must be an explicit mechanism for the requesting process to receive the report. This mechanism can be an interrupt to a predetermined user routine or can be a system procedure call made by the requesting process.

There are two types of system procedure calls used for report. These are *wait* and *poll*. A wait specifies a specific send, a specific receive, or a set of sends and/or receives. The user process is then blocked until a matching send or

receive completes. Poll is like a non-blocking version of wait. Poll returns the report of the corresponding send or receive if there is one. If not, it returns "nothing has happed yet" and the requesting process can continue. We assume that wait(any) and poll(any) match the earliest qualified send or receive.

When discussing communication we will use the following terms.

□ Communication Event:

A *communication event* is any event that affects the transfer of a message from a sending process to a receiving process. A communication event is any phase of sending or receiving from the request through the report. The group of events that comprise a send or a receive are called the send or receive *activity*. The time from the start of the request up to and including the end of the corresponding report is called the *lifetime* of the activity.

□ Match:

We say a send and a receive *match* if the message sent by the send is the message received by the receive.

We say a wait/poll and a send/receive *match* if the wait/poll is the report phase of the send/receive.

□ Remote Procedure Call:

A *remote procedure call* is the exchange of two messages between two processes called the *client* and the *server*. The client sends a message to the server, posts a receive request, and blocks until the receive is reported. The server receives the message, may or may not do some computing, and finally sends a reply.

1.2 An Overview of Timing Errors

In a distributed program events occur concurrently on different processors. The order in which events occur cannot be easily determined; a program that works correctly one time may have an error and fail subsequently if the timing between processors changes. We call these errors *timing errors.*

□ Timing Error:

A *timing error* is a program error that may or may not occur depending on time-dependent ordering of events.

Timing errors depend on sequences of events. A program containing a timing error might run correctly many times before the timing error manifests itself.

Examples of timing errors include:

(1) failure to use old information before it is overwritten by new information

(2) attempt to use information before it arrives

(3) failure to send old information before it is overwritten by new information

(4) arrival of messages in an unexpected order (in a multiple-process conversation)

(5) failure to keep a conversation synchronized

In a distributed environment, timing errors are tied to interprocess communication.

Examples of Timing Errors

The following are five examples of timing errors. Non-blocking communication is used in all of the examples.

Example: #1.1: Overwriting old information before it is used

A process inadvertently receives a message into a buffer before using the information last received into that buffer. For example, a programmer means to write:

```
Receive(buffer1);
Receive(buffer2);
Wait for first receive to complete.
Compute with buffer1.
```

Unfortunately the programmer targets buffer1 for both receives. Since both receives are non-blocking, the information used during the computation phase may be from the first receive, the second receive, or a combination of the two.

Example: #1.2: Attempt to use information before it arrives

The following example shows how a process can use information before it arrives.

```
Receive(buffer1);
```

```
Compute with buffer1.
```

Since there is no Wait statement, the process might start computing before the data had been received in the buffer.

Example: #1.3: Changing a buffer before the send occurs

The following example shows how a process can inadvertently change information before before it is sent.

```
Send(buffer);
Change buffer.
Send(buffer);
```

The first Send above informs the operating system that the process wants to send the information in the buffer, but the send might not actually occur until later. Since the operating system sends messages directly from the user's address space, the actual sending of the message could occur before the user changes the buffer, while the user is changing the buffer, or after the user has changed the buffer.

Example: #1.4: Arrival of messages in an unexpected order

A distributed program might fail sporadically because messages arrive at a process in an unexpected order. This unexpected order of message arrival can cause the receiving process to produce an incorrect result or perform an incorrect action. For example (see figure 1.1), assume there is a distributed

program with a process called the *global data handler* (GDH). The GDH administers the global data for the distributed program, storing values sent to it and returning the values of variables that are requested of it. Process S keeps statistics about the distributed program. At the end of each round of computation, process S asks the GDH for the values of various variables. Process S waits to make this request until process L informs it a round has been completed. Process L is the last process to perform a computation in a given round and L sends its computed value to the GDH before informing S that a round has ended. Since sending a message does not block a process, it is possible that the message from process S to the GDH will arrive before the message from process L and the GDH will send the wrong information to S. The action performed by the GDH is the same no matter in which order the messages arrive; the results, how-

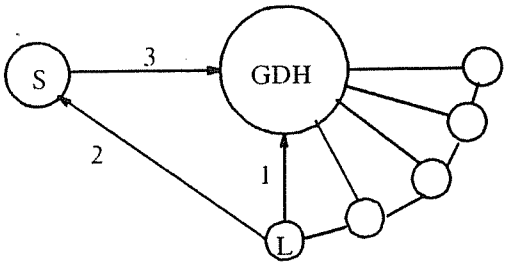


Figure 1.1: example 1.4: The Global Data Handler

ever, will differ.

Example: #1.5: Failure to keep a conversation synchronized

A conversation is synchronized when all participants agree on where in the conversation each message occurs. If at least two participants disagree then the conversation is not synchronized. For example: A server process handles student records for a university and will answer YES or NO when asked various questions such as "is student X enrolled this semester". A client process makes a stream of requests. If a request or a response is lost then the conversation will become unsynchronized.

A lost message need not be the fault of the communication subsystem but might be caused by the client process thinking it had made a request when none had been made. For example, a programmer expects a message to be sent as part of the evaluation of a logical expression. If the program is written in a language that short-circuits logical expression evaluation, then the communication part of the expression could be skipped (See figure 1.2).

if flag **and** Send(msg) **then** ...

When 'flag' is false the Send will be skipped.

Figure 1.2: Short-circuit of logical expression

1.3 Related Work

Other researchers [Garc84, Gros83, LeBl83, Mcda77, Phil82, Schi81] have discussed special problems in debugging distributed programs. Debugging distributed processes differs from debugging multiple processes on a single processor in two important ways. In the latter case, the programmer can look at the order in which events occur in order to determine causes of errors. The programmer can also stop all processes at the same instant when an error is suspected. In a distributed environment, processes execute concurrently. The only way to tell the order in which events in one process occur in relation to events in another process is to look at the communication between the processes. Similarly, by disabling communication, all processes can be suspended at the same relative time to examine the state of the program. Most of these researchers agree that for a debugger to handle the special problems inherent in a distributed environment, the debugger has to deal with interprocess communication.

Some researchers have addressed the theoretical issues involved in debugging distributed programs [LeBl83, Gros83, Garc84]. Both LeBlanc and Gross suggest that two major, necessary features of a distributed debugger are the ability to interact with the operating system and the ability to control the communication between processes. They claim distributed programs are not only characterized by their internal state but also by the state of kernel tables, containing information such as which processes can communicate with which other processes and what messages are pending between which processes. They point out that the only real difference between debugging a sequential program and debugging a distributed program is monitoring the interprocess communication. They suggest a hierarchy of debuggers: a "traditional" debugger to aid in testing each sequential process and a higher-level debugger to monitor communication. Our work supports their claim; the tool we have developed is a debugger to monitor communication.

A group headed by Garcia-Molina also address the issue of debugging distributed programs [Garc84]. Their model has an operating system, a communication module, and a debugging module (DM) on each processor. There is also one process per system, the master debugger, which handles the user interface. The DM is integrated with the operating system, the communication module and the programming language. The DM monitors interprocess communication and

can find the values of the variables in the application program. Certain events, such as sends and receives, are traced by the DM which records their occurrences. Part of the information kept when recording an event includes a time stamp.

These researchers claim that observed results are hard to reproduce in a distributed environment, so tracing must play a predominant roll in debugging. We agree with this assessment and incorporate tracing in our implementation. They also note that the inclusion of the DM will affect timing and therefore mask some bugs. They claim little can be done about this problem. We do not entirely agree with this claim as we show in Chapter 3. To look at the saved information all of the program's processes need to be stopped. They suggest both a user initiated and a system initiated "stop-all-processes" capability. We have also made use of a "stop-all-processes" capability.

The Garcia-Molina group envisions distributed debugging taking place in two phases. The first phase would be tracing as described above. If tracing does not find the bug it should narrow down the search. In the second phase the area of the program that is still in doubt would be thoroughly tested. They claim that the second phase "is needed because in many cases it is simply not feasible to collect all the necessary information during the first phase".

Using the DM, as described above, is again more like program testing than program debugging. The Garcia-Molina group recognizes the possibility of bugs occurring in a released program and claims that tracing will have to be turned on again and an attempt made to reproduce the bug. If this is one of the bugs masked by the inclusion of the DM then they have no suggestion of a method to use to find the cause of the bug. They list more than twenty important events to be stored in the trace but fail to describe the way the information can be used.

Other researchers [Baia83, Phil82, Schi81, Owen81, Smit83] have implemented debuggers for distributed environments. A debugger by Baiardi [Baia83] monitors the user's program during execution and traps if problems arise. The four debuggers by Phillips [Phil82], Schiffenbauer [Schi81], Owen [Owen81], and Smith [Smit83] are interactive debuggers for distributed environments.

Baiardi's debugger has two modes as suggested by LeBlanc and Gross, sequential and concurrent. Users supply this debugger with a formal description of the process behavior. The debugger compares the actual performance of the user's program with the formal description and traps to the user if there is a difference. The formal statement describes conditions such as which processes interact and asserts a partial ordering on events.

Interactive distributed debugging was attempted by Owen, Smith, Philips, and Schiffenbauer. Owen [Owen81] has implemented a debugger for distributed

programs that allows the user to set breakpoints on message send and receive and saves all messages in a file. The SPIDER debugger [Smit83] provides the facility for the user to create and manipulate interprocess events. SPIDER keeps transcripts of messages sent and provides demon processes to automatically monitor and modify communication. It also allows the user to test an individual process by handling all communication to and from that process.

Similar interactive debuggers were implemented in the Blackflag debugger [Phil82] and a debugger developed at MIT by Schiffenbauer [Schi81]. Both debuggers allow the programmer to interpose a debugging process between two communicating processes to monitor message traffic.

With Blackflag a user can inspect messages before having them forwarded to the receiving process or have them forwarded automatically. The design of Blackflag calls for allowing the user to set breakpoints dependent on the contents of a message, edit messages before forwarding them, delete messages, and create messages from scratch.

One positive aspect of Blackflag is that it requires no changes to the operating system or to the compiler. The only cooperation required is from user processes that must have global names for all kernel ports used. Having global names presumably lets the debugger find the information it needs to eavesdrop.

Schiffenbauer recognizes the possibility of timing errors (called lurking bugs) and attempts to deal with them. This debugger has two parts, a *nub* on each processor and a *central site* (CS) residing on one processor communicating with the user. All packets are sent by the nub on the sending processor to the CS which asks the user for an action to perform with the packet. Possible actions include: send the packet, lose the packet, delay the packet, and change the packet. Schiffenbauer attempts to limit the impact the debugger will have on the relative timing between the user processes by having all processes use logical instead of real time.

These debuggers are more program testers than they are debuggers. They cannot help the user find errors unless the program is run under the control of the debugger. With Blackflag present, a program will not necessarily behave the same as it would at other times. The presence of Blackflag potentially affects the timing between processes in three ways. First, a message has to be sent twice to get to its destination, once from the sending process to Blackflag and once from Blackflag to the receiving process. Second, the user can delay the delivery of a message. Third, the inclusion of Blackflag adds another process to the system; both the order in which the user's processes are run and the allocation of processes to processors might change. Schiffenbauer tries to overcome the first two of these problems by having his processes use logical time (as opposed to real

time). Presumably he could make sure the processes were allocated to processors as they would be if there were no debugger. Though Schiffenbauer recognizes the problems caused by lurking bugs, he makes the naive assumption that the programmer will have an intuition as to where the lurking bugs might occur and will test these sections thoroughly with the debugger. Neither of these debuggers allow the programmer to find the causes of bugs as they occur in a released program.

The approach taken in this thesis is to characterize timing errors and explain the underlying basis for them. It is our belief that understanding which features of interprocess communication lead to timing errors will help the programmer avoid them when possible and will better clue the programmer as to their possible location when they do arise. We advocate using the debugger on the "release" version of the program when the bug is detected, not a test version compiled with special features after the bug has been detected. Our approach to finding timing errors is similar in many ways to two approaches taken to performance analysis of distributed programs [Mcda77, Mill85]. METRIC [Mcda77] was designed to measure performance in distributed programs, but the author claims it can also be used to detect timing errors (called cosmic ray bugs).

METRIC is composed of three parts, the probe, accountant, and analyst. The probe is the user interface to METRIC. It is a procedure call provided to

the user that sends information to the accountant. The accountant collects these messages and acts as a filter, storing those that are interesting and discarding the rest. This saved information is then made available to the analyst if needed.

Miller's system [Mill85] has three similar components. The metering phase collects information and passes it on to the filter. The filter discards any messages not matching a user-supplied pattern and stores the rest for analysis. The analyst uses the data from the filter to build graphs that show a partial ordering of events and the amount of time between events. The analyst then looks for a mapping of processes to processors that will increase the parallelism in the graph.

Both of these approaches could be used to find timing errors but they have significant flaws. The methods used to record events would more than triple the message traffic since each message exchanged between users would generate a message to the accountant (filter) from the probe (meter) at the sending processor and another message to the accountant from the probe at the receiving processor. METRIC was able to sidestep this problem because the system on which it was implemented provided a separate physical channel on which these messages could be sent. Miller provided a buffering mechanism to store many records and forward them as a unit. Drawbacks to this technique under our implementation are discussed in section 3.2.

To use METRIC to detect timing errors, the runtime could execute a probe procedure call to record each important event such as a send or a receive. Using METRIC this way would not catch all timing errors on systems such as Charlotte since the probe would be initiated by the user when the send or receive request is made and not at the completion point.

In Miller's system event recording is initiated in the operating system, providing the facility to record all events whose account is necessary for finding timing errors. The filter could be adjusted to weed out extraneous events. The partial ordering of events extracted from the graphs can be used to detect timing errors. These graphs will be discussed further in section 1.4.2.

1.4 Timing Graphs

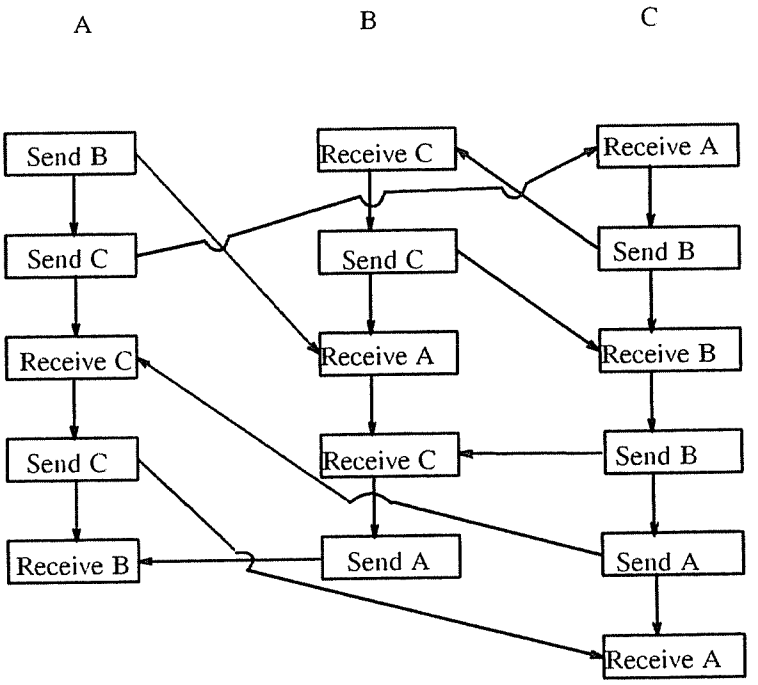
The Use of Timing Graphs

Timing errors are caused by events failing to occur in the same order from one run or one iteration to the next. To find timing errors, we need to be able to show the order in which events occurred. Lamport [Lamp78] defines a *partial ordering* of events for a distributed program. He shows this partial ordering can be used to construct, what we call here, *timing graphs.* A *timing graph* is a directed acyclic graph representing this partial ordering. We will use the symbol $\rightarrow$ to represent the relation *precedes.* We say A $\rightarrow$ B if

(1) A and B are in the same process, and A happens before B, or

(2) A is the completion point of a *send*, and B is the completion point of the matching *receive*, or

(3) There is some event C such that A $\rightarrow$ C and C $\rightarrow$ B.



For example, we can see that A's send to B precedes B's second receive from C. We can also see that A's send to B precedes B's first receive from C.

Figure 1.3: A Timing Graph

To construct the timing graph (see figure 1.3), an arc is drawn from $node_A$ to $node_B$ if: $A \rightarrow B$ and there is no $node_C$ such that $A \rightarrow C$ and $C \rightarrow B$. The nodes of the timing graph are events and an arc from $node_A$ to $node_B$ means A directly precedes B in time. Timing graphs may be constructed from information known at compilation and execution time. In practice, the entire timing graph need not be built. We propose that information should be saved during execution of a distributed program so that timing graphs can be built, if needed, when the existence of a bug is discovered. Our implementation continually builds only a *skeleton* of this graph and derives other parts of the graph from it when needed. The timing graph is then used to show the relative order of events in the distributed program. Our claim is that this information is sufficient to find causes of timing errors.

The Philosophy of Timing Graph Use

Communication event tracing is keeping a history of the communication events as they occur in a running program. This is necessary for creating timing graphs. We do not think that the tracing should be activated and the program rerun when a timing error is suspected. We propose to keep this tracing facility active at all times for two reasons. First, timing errors may not be easily repeatable. The precise timing between processes that caused the error may not show

up again for many runs once tracing is activated. Second, tracing affects the execution of a program. More instructions are executed to accomplish the tracing, so the timings between processes can change. A program that encounters a timing error while tracing is disabled might never encounter the same error with tracing active. We view tracing as being similar to subscript checking supported by many current compilers; tracing is active unless specifically disabled by the user.

Since this tracing is always active it must have minimal impact on the performance of the user's program. The tracing should also be *transparent* to the user in that the programmer should not have to write any special code to help with the tracing. We will discuss an implementation of this tracing facility in Chapter 3.

1.5 Organization of this Thesis

Chapter 2 describes the theoretical aspects of ordering errors including timing errors and misorderings. We show ways to prevent timing errors and misorderings, discuss reasons why such prevention is not always desirable and discuss ways of detecting ordering errors as they occur. Chapter 3 describes TAP and Charlotte. Chapter 4 describes the experiments run using TAP. We show examples of distributed programs containing timing errors and use TAP to

discover the causes of the errors. Chapter 5 contains suggestions for writing distributed programs, designing distributed programming languages, and designing operating systems for distributed programming based on lessons learned in chapter 2. Chapter 6 contains conclusions and suggestions for future research.

Chapter 2

Ordering Errors

An *error* occurs in a process when the process behaves incorrectly. This misbehavior is reflected in the process's state, which is invalid after an error occurs. An *ordering error* is an error caused by the relative order in which events occur. We will discuss two types of ordering errors in this chapter, timing errors and misorderings. The relationship of ordering errors to other classes of errors is shown in figure 2.1.

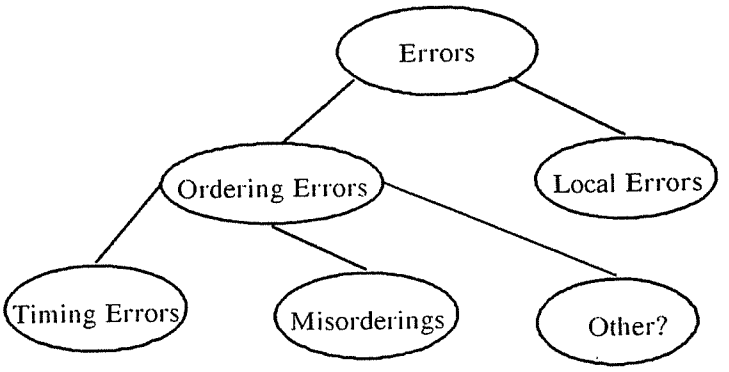


Figure 2.1: Ordering errors

2.1 Prevention and Detection of Timing Errors

In chapter 1 we have shown examples of timing errors and the approaches others have used to deal with them. In this section we characterize timing errors. We discuss types of timing errors, conditions that permit them to occur, and ways to prevent and detect their occurrence. The following definition is repeated from chapter 1.

□ Timing Error:

A *timing error* is a program error that may or may not occur depending on time-dependent ordering of events.

Timing errors depend on sequences of events. A program containing a timing error might run correctly many times before the timing error manifests itself. For this discussion we make two assumptions. First we assume each process is sequential. The only interrupts that might affect a process are those dealing with communication. Second, we assume every activity reaches the report phase. This assumption does not affect the analysis we present since any uncompleted activity can be assumed to reach report after the program terminates.

Timing errors fit into two disjoint categories, single-event timing errors, those involving the order between a communication event and a local event, and multi-event timing errors, those involving the order between two communication

events. We discuss each category separately.

Preventing Single-Event Timing Errors

The first category of timing errors is single-event timing errors (shown in figure 2.2). In this section we will define single-event timing errors, show their necessary conditions, and show ways to prevent their occurrence. We begin with the following definitions.

□ Single-event Timing Error:

A *single-event timing error* is a timing error involving exactly one communication event.

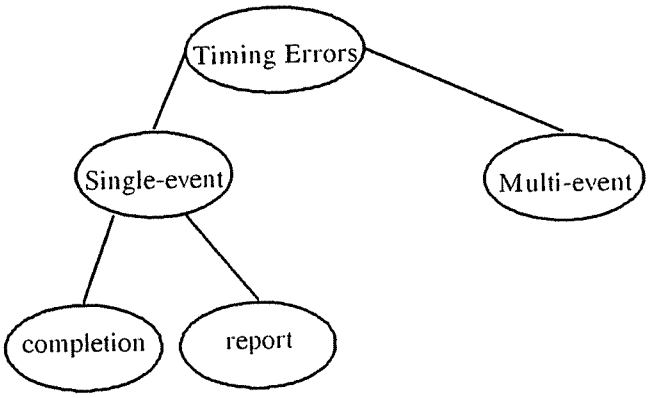


Figure 2.2: Single-event timing errors

Examples

- Send, change (see section 1.2, example 1.3)
- Receive, use (see section 1.2, example 1.2)
- An interrupt occurs when a message is received. The interrupt routine uses and changes many variables. The process that uses those variables at other points might produce different results if the interrupt occurs at different points in the code.

□ State:

The *state* of a process at a given point in time is the value of its variables in its address space and the information kept for the process by the operating system. The state includes information kept by the operating system kernel, the communication subsystem and any utility processes such as a file server.

□ Space:

Let M be the set of memory locations that can be part of any valid state of a process P. The *Space* S used by a communication event is that subset of M used by the communication event. This space includes the subset of M used for the report.

□ Local Event:

A *local event* is any operation, not part of a communication event, that accesses or changes a memory location in a process. Examples of local events include assignment statements, expression evaluation, and function calls.

With these definitions we can state the following theorem.

Theorem #1: There are two necessary conditions for the occurrence of a single-event timing error.

(1) A communication event and a local event must be able to occur in either order (we say the communication event occurs *asynchronously*).

(2) The communication event's space must be shared with this local event.

Proof

(1) From the definition of timing error there must be at least two events whose relative order varies. Since only one of these events is a communication event the other must be a local event.

(2) Let C be the communication event involved in the timing error. Let L be a local event such that the relative order of C and L varies. Assume L neither accesses nor changes any locations in C's space. Then, no matter in which order C and L occur, there can be no timing error because the behavior of the program will not vary.

We can prevent single-event timing errors by preventing the occurrence of one or both necessary conditions stated in theorem #1. One way that asynchronous communication events can be prevented is through the use of blocking communication.

Theorem #2: A program that uses only blocking communication contains no single-event timing errors.

Proof

By theorem #1, the order between a communication event and some other event must vary to have a single-event timing error. Blocking communication guarantees that every communication event will occur in the same order with respect to all other events. Since there is no asynchrony there can be no single-event timing errors.

To prevent single-event timing errors, we can use conditions weaker than the requirement of blocking communication. First we note that only communication events that meet both conditions in theorem #1 are completion and report.

Lemma #1: Completion and report are the only communication events that can be involved in a single-event timing error.

Proof

Since completion and report are the only communication events that are able to both share space with other events and occur asynchronously with other events, then they are the only communication events that can be involved in a single-event timing error.

Buffering in the communication subsystem can prevent single-event timing errors while still allowing non-blocking communication for processes that use synchronous report. This use of buffering can prevent asynchronous completion points as follows. On a send request, the communication subsystem copies the message to a buffer of its own before returning to the user's process, thereby preventing the user's process from changing the message before the send occurs. A receive request asks the system to hold on to a message if it arrives. The matching wait causes the system to copy the message into the user's buffer. With this method, the completion point of every receive must occur with the report and the completion point of every send must occur with the request.

Asynchronous report can be avoided by using blocking report (wait). Blocking report has to occur at a distinct point in the process's execution. Asynchronous report is not avoided by the use of polling or interrupts, since with polling and interrupts, report does not have to occur at a distinct point in the processes execution.

Theorem #3: The use of buffers in the communication subsystem, where the requesting process is blocked on a send request until the message is copied to a system buffer and a message is transferred to the recipient at the time of report, along with the use of synchronous report, prevents single-event timing errors.

Proof

By theorem #1 and lemma #1, to have a single-event timing error the relative order of occurrence between completion or report and some other event must vary. Using the buffering algorithm with wait for report, the completion point of every send must occur with the request, the completion point of every receive must occur with the report, and the request and report must occur at known points. Therefore, there can be no single-event timing errors.

There are, however, problems associated with buffering messages in the communication subsystem. There is added overhead; every off-machine message has to be copied an extra time and the communication subsystem has to worry about buffer management. To avoid this extra overhead, the Charlotte kernel [Fink83a] provides no buffering.

Another way to guarantee that no single-event timing error can occur is to make sure the second necessary condition is prohibited. That is, we can prevent single-event timing errors by guaranteeing no communication event shares space with any other event during the lifetime of the communication event's activity. We call this restriction *bounded asynchrony.*

Let the *terminus* of an activity be the earliest point in the process's code by which the report must occur. For synchronous report terminus occurs immedi-

ately after the last possible wait statement that can match the request. For polling, terminus occurs some time after the last possible poll statement that can match the request. For example, given the following code:

```
A:    while poll(X, received) == false do
                .
                .
                .
      end while
B:
```

terminus occurs at B since this is the earliest location by which we can guarantee the report has occurred. For interrupt report terminus might not occur until the end of the process. We call the time from an activity's request until its terminus the *extended lifetime* of the activity.

Since, by lemma #1, completion and report are the only communication events that can be involved in single-event timing errors, we can prevent single-event timing errors by insisting that all completion and report events share no space with other events with which they may asynchronously occur. Completion and report must occur between two well known locations. These are the corresponding request and terminus. Therefore we can state the following theorem.

Theorem #4: No single-event timing error can occur for a given activity if the report event's space and the completion event's space do not intersect with space used by other events that dynamically occur between the request and the terminus for that activity.

Proof

Since both completion and report must dynamically occur between the corresponding request and terminus, then the only events that can be in asynchrony with them lie in this same region. Since they do not share space with events in this region, they have no events with which they meet both necessary conditions for single-event timing errors.

	NECESSARY CONDITION AVOIDED	
Method	Shared Space	Asynchronous Communication Event
Blocking Communication		X
Kernel Buffering		X
Bounded Asynchrony	X	

Figure 2.3: Methods for preventing single-event timing errors

---

In this section we have shown necessary conditions for the occurrence of single-event timing errors. We have also shown that single-event timing errors can be prevented in several ways by preventing at least one of these necessary conditions. Our findings are summarized in figure 2.3.

Preventing Multi-Event Timing Errors

The other category of timing errors is multi-event timing errors (shown in figure 2.4). In this section we will define multi-event timing errors and show ways to prevent their occurrence.

□ Multi-event Timing Error:

A *multi-event timing error* is a timing error involving two or more communication events.

Examples

- GDH (see section 1.2, example 1.4)
- Conversation becomes unsynchronized (see section 1.2, example 1.5)
- Receive, Receive, Wait, use (see section 1.2, example 1.1)

We can now state necessary conditions for multi-event timing errors.

Lemma #2: For a multi-event timing error to occur, the relative order between two communication events must be non-deterministic and they must share space.

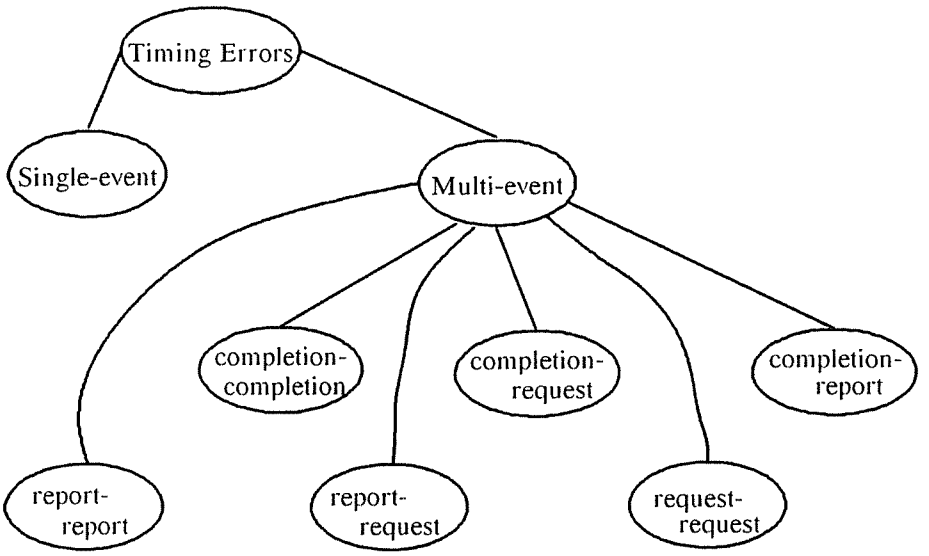


Figure 2.4: Multi-event timing errors

Proof

The proof follows directly from the definition of multi-event timing error and the definition of timing error.

We can now use these necessary conditions to develop methods of preventing multi-event timing errors. The only communication events that can share space with each other are request, report, and completion. Therefore, multi-event timing errors can only arise from the relative order of occurrence of two or more request, completion or report events. We will use the following definitions

to talk about multi-event timing errors.

□ Conflict:

We call a multi-event timing error occurring because of the relative order of two reports a *report-report conflict.* Similarly, we have *report-completion conflict, completion-completion conflict, report-request conflict, request-completion conflict,* and *request-request conflict.*

Conflict occurs because the communication events involved occur asynchronously or because other communication events that affect the order of the ones in conflict, occur asynchronously. Request events cannot occur asynchronously; request-request conflict refers to the use of requests that can potentially match more than one corresponding request. For instance, receive(any) might potentially match any send request from the specified set. Because these sends occur asynchronously, request-request conflicts arise.

□ Deterministic:

We say the relative order of two communication events is *deterministic* if their relative order is a function only of reproducible events and objects such as the input data, previous communication events, and the program code. If the relative order between two communication events is not deterministic we say their relative order is *non-deterministic.*

Examples

The following examples assume non-blocking communication.

- If a process has the following segment of code, then the order of completion of the two sends is deterministic.

```
Send(1, msg1);
Wait(1, send);
Send(2, msg2);
```

- If the following segment of code is in a process, then the order of completion (point at which the sending process is no longer responsible for the message) of the two sends is non-deterministic.

```
Send(1, msg1);
Send(2, msg2);
Wait(all, send);
```

□ Deterministic match:

A *deterministic match* is a match that is a function only of reproducible events and objects such as the input data, previous communication events (those occurring before either end of the match), and the program code.

Deterministic send-receive match can be ensured by the use of specific, private channels for communication. The use of specific, private channels guarantees the receive can only match one channel, and a message on a channel can only come from one sender. Since messages from one process to another are delivered in the order sent, the receive can only match one send.

□ Deterministic Process:

If the relative order of every pair of the process's communication events (those communication events whose activity has its request as one of the process's statements) is deterministic and every match involving at least one of the process's communication events is a deterministic match, then we say the process is a ***deterministic process***.

□ Well-Ordered:

Let $\xi(P)$ be the set of events comprising process P. A process P is ***well-ordered*** if for every pair of communication events A and B ϵ $\xi(P)$, the relative order of the occurrence of A and B is deterministic.

A distributed program is ***well-ordered*** if all of its processes are well ordered.

With these definitions we can state the following lemmas.

Lemma #3: If the relative order of occurrence of two communication events is deterministic, then there is no multi-event timing error involving the order in which the two communication events occur.

Proof

This lemma follows from the definition of timing error and the definition of multi-event timing error.

Lemma #4: A well-ordered process has no multi-event timing errors.

Proof

Since a process is well-ordered, a given input will always produce the same order for any two communication events (by the definition of well-ordered and the definition of deterministic). Since the order of all communication events is deterministic, there are no multi-event timing errors (by definition of multi-event timing error).

Lemma #5: A program has no multi-event timing errors iff each of its processes has no multi-event timing error.

Proof

For a multi-event timing error to occur there must be two communication events in the same process whose relative order varies (by definition of multi-event timing error). If a program has no multi-event timing error then no such process exists. If none of a program's processes contains a multi-event timing error then the program cannot, since the error has to be in a process.

Lemma #6: A process is well-ordered if it uses only blocking communication and specific, private channels (implies a ***deterministic match***).

Proof

For conflict between two communication events to arise, the events have to have the possibility of occurring in either order. Blocking communication guarantees that each activity will occur as an atomic unit so the relative order between communication events in different activities is determined by the relative order of their requests. Request-request conflict can only occur if the receive requests can match more than one send. Receive requests that specify specific, private channels have only one possible send as a match (a ***deterministic match***). Therefore, no conflicts can occur, so the relative order of occurrence of all communication events in the process is deterministic. Therefore, by definition of well-ordered, the process is well-ordered.

With the above lemmas we can now discuss methods of preventing multi-event timing errors. One way to prevent multi-event timing errors is through the use of remote procedure call.

Theorem #5: A process cannot have a multi-event timing error if the only method of communication used by the process is the client end of remote procedure call.

Proof

Remote procedure call is actually a send followed immediately by a blocking receive on a specific private channel. Therefore, by lemma #4 and lemma #6, a process that only uses remote procedure call can have no multi-event timing errors.

Though the use of remote procedure call can potentially degrade parallelism, it will avoid the possibility of timing errors involving the process making the remote procedure call for the communication events involved in the remote procedure call.

We can also use the above lemmas to show that if a distributed program is restricted to using blocking communication on specific private channels, then it can have no timing errors.

Theorem #6: Using blocking communication and deterministic match (which means no receive(any)) guarantees a program is free of timing errors.

Proof

(1) single-event timing errors cannot occur (by theorem #2)

(2) each process is well-ordered (by lemma #6)

(3) therefore, no process has a multi-event timing error (by lemma #4)

(4) therefore, the program has no multi-event timing errors (by lemma #5)

(5) With no single-event and no multi-event timing errors, the program has no timing errors.

We have shown that all timing errors can be prevented by the use of blocking communication on specific, private channels. These restrictions can, however, degrade performance of a program by diminishing the amount of parallelism between its processes.

Relaxing Restrictions

To prevent timing errors we have placed restrictions on the characteristics allowed for both communication channels and communication events. There are, however, instances where timing errors are prevented with some of these restrictions relaxed.

One restriction that we can relax is the requirement of blocking communication. We have shown that single-event timing errors can be prevented without requiring blocking communication by insisting no event uses a communication event's space during the lifetime of the communication event's activity. Similarly, while allowing non-blocking communication, we can prevent multi-event timing errors by insisting that no two activities, sharing space, have intersecting life-

times. To insure lifetimes of space-sharing activities do not intersect, we make the restriction that deterministic report must be used.

Theorem #7: Multi-event timing errors cannot occur if only deterministic match is used (no receive(any) or wait(any)) and activities with intersecting lifetimes do not share space.

Proof

By the definition of multi-event timing error, for a multi-event timing error to occur, communication events must share space and be able to occur in either order. If the lifetimes of two activities do not intersect then their relative order is determined by the order of their requests, which is deterministic. If the lifetimes of two activities do intersect then they have no space in common. In either case, multi-event timing errors cannot occur.

Another restriction we can relax is deterministic match. A process using non-deterministic match for report can handle events as they occur instead of having to wait for each one individually. This flexibility allows a server to have greater throughput in handling requests and prevents one client from blocking a server by failing to receive a message. However, if the match for a report is non-deterministic, control of the process can be affected. The report might cause different sections of the process to execute depending on which request the report

matches (See figure 2.5). If we can guarantee that the behavior of the program does not depend on which section executes, then we can prevent multi-event timing errors while still allowing this non-determinism.

We will use the following definitions to deal with processes containing this non-determinism.

□ Section:

A *section* of a process is a dynamically contiguous group of one or more statements.

The Wait can match either Receive. This match will determine which part of the if statement executes.

```
Receive(X, buf1);
Receive(Y, buf2);
return_code := Wait(any);
if return_code.who = X then
        .
        .
else
        .
        .
endif
```

Figure 2.5: Non-deterministic report affects control

If A and B are sections then we use AB to mean the execution of section A immediately followed by the execution of section B.

□ Independent:

Let V_0 be the set of variables in state S_0. If the channels used for communication requests and reports in section A are different from those in section B, then consecutive sections A and B are *independent* iff, for any reachable state S_0 that immediately precedes the execution (in either order) of the two sections A and B, a unique state S_n is reached after this execution of the two sections and we can produce 4 disjoint sets of variables V_1, V_2, V_3 and V_4 where $V_1 \cup V_2 \cup V_3 \cup V_4 = V_0$ such that

(1) The value of all $v \in V_1$ are referenced (used or changed) only in section A

(2) The value of all $v \in V_2$ are referenced only in section B

(3) The values of all $v \in V_3$ are unchanged in sections A and B.

(4) The values of all $v \in V_4$ are changed on their first reference in both sections A and B.

When sections are independent they cannot interfere with each other. The variables in V_1 and V_2 are only referenced in one section, so their use cannot affect the other section. The variables in V_3 are read-only variables and therefore their use in one section cannot affect the behavior in the other section. The variables in V_4 are changed before they are read, so their values at the start of

either section do not affect behavior.

□ Independent Report:

Let X and Y be two activities. If the section that report(X) causes to execute is independent from the section report(Y) causes to execute and the state arrived at after their execution is independent of the order in which the sections execute, then we say report(X) and report(Y) are *independent reports.*

Example

```
Receive(A, buffer1);
Receive(B, buffer2);
for i := 1,2 do
      case Wait(Any)
      A:    compute with buffer1;
      B:    compute with buffer2;
      end case;
end;
```

If cases A and B are independent sections then the two reports are independent.

The concept of independent sections is important for showing situations where report with non-deterministic match cannot lead to multi-event timing errors. In this situation a process can potentially achieve greater parallelism. With the above definitions we can now show requirements for the safe use of report with non-deterministic match.

Theorem #8: There is no report-report conflict with independent reports.

Proof

For a multi-event timing error to occur involving two reports, the behavior that occurs because of the occurrence of one report must affect the behavior that occurs because of the other report. Neither of the above can happen for independent reports.

There are instances where non-deterministic send-receive match (receive(any)) does not lead to request-request conflict. These situations occur when the behavior of the process depends on what is received instead of where in the code it is received and the sections where this behavior takes place are independent. To illustrate this point we use the following two examples.

In the first example there is receive-receive conflict because the behavior of the process depends on where in the code the message is received.

```
BlockReceive(any, buffer1);
find sum of values in buffer1;
BlockReceive(any, buffer2);
find product of values in buffer2;
```

In the next example there is no receive-receive conflict because the behavior of the process depends on what is received and the sections where this behavior takes place are independent.

```
BlockReceive(any, buffer1);
if sender = A then  Aaction;
else                Baction;
BlockReceive(any, buffer2);
if sender = A then  Aaction;
else                Baction;
```

Procedures Aaction and Baction are independent.

We use the following definitions to prove these situations are free of request-request conflict.

□ Connected:

Let Send S match Receive R and let Receive R match Report T, then we say S and T are ***connected*** and we call T the ***connection*** of S.

□ Action:

Let S be a send activity and let T be the connection of S, then ***action***(S) is the section that report(T) causes to execute.

Theorem #9: Let process P contain two receive requests R1 and R2 that might be in conflict. Let S1 and S2 be the only two send activities that can match R1 and R2 such that each send matches exactly one of the receives. There is no request-request conflict for R1 and R2 if action(S1) and action(S2) are deterministic and if the connection of S1 and the connection of S2 are independent reports.

Proof

P's behavior is the same no matter which send matches which receive. Since the behavior does not change, the order of the receive requests does not lead to a multi-event timing error so there is no request-request conflict.

In this section we have shown how the choice of communication primatives can lead to preventing or allowing multi-event timing errors. We refer to the table in figure 2.6 as we summarize our findings here. The table shows choices in three dimensions, blocking or non-blocking communication; buffering or non-buffering in the communication subsystem; and deterministic send-receive match, deterministic wait match, both or neither. For buffering in the communication subsystem we assume the sender of a message is blocked until the communication subsystem has a copy of the message.

Making the following choices can prevent the possibility of multi-event timing errors.

A and B:

With blocking communication and deterministic match there can be no time-dependent behavior therefore, no timing errors (by theorem #6).

C: If synchronous report is combined with these choices there are no asynchronous communication events. Therefore, a process that only uses these

DETERMINISTIC MATCH	BLOCKING		NON-BLOCKING	
	BUFF	NON-BUFF	BUFF	NON-BUFF
ALL	**A**	**B**	**C**	**E**
SEND-RCV	Y	Y	**D**	**F**
REPORT	XA	XB	XC	XE
NONE	Y	Y	XD	XF

A-F, XA-XF: cases where multi-event timing errors can be prevented
Y: undefined cases

Figure 2.6: Preventing Multi-Event Timing Errors

features is well-ordered and has no multi-event timing errors (lemma #4). Combined with asynchronous report, these features allow multi-event timing errors to occur since report-report conflicts, report-completion conflicts and request-report conflicts can occur. Adding the requirement that activities with intersecting lifetimes do not share space guarantees that no multi-event timing errors can occur (theorem #7).

D: This case is similar to case C except non-deterministic report makes report-report conflicts possible. If synchronous reports are independent then report-report conflicts are avoided (theorem #8).

E: This case is like case C except that non-buffering in the communication subsystem admits the possibility of completion-request, completion-completion and completion-report conflicts. Requiring that no space is shared between intersecting activities guarantees that multi-event timing errors cannot occur (theorem #7).

F: This case is like case E but, as in case D, for a process to prevent multi-event timing errors while using only synchronous report care has to be taken that the process only uses independent reports.

XA Each of these cases allows non-deterministic send-receive match. To prevent multi-event timing errors while using the features in these sections the precautions discussed for the corresponding section (Xn corresponds to n) have to be followed. In addition, the actions of all sends that might conflict have to be deterministic and independent.

Case Y cannot occur since there are conflicting semantics in the choices of non-deterministic match for report and blocking communication. Blocking communication uses implicit report which must match the specific request on which the process blocked.

Determining whether lifetimes intersect is difficult when asynchronous report is used. Since an activity's lifetime is contained in the activity's extended lifetime, substituting ''extended lifetime'' for ''lifetime'' in the theorems proved

in this chapter creates stronger conditions for guaranteeing no timing errors.

As we have shown in this section, all timing errors can be prevented by proper system design and proper use of language constructs for communication. We will discuss the implications of preventing timing errors in chapter 5. There are, however, reasons for not using the methods advocated in this section. If we are not going to prevent timing errors then we must at least detect their occurrence.

Detecting Timing Errors

Techniques that prevent timing errors can cause loss of parallelism and potential deadlock. The use of blocking communication in systems such as Charlotte can exact a large cost in decreased parallelism. In the time it takes to send a message and get back the acknowledgement, even when the receiver is ready, a process can execute on the order of 6000 instructions. If the receiver is not ready then the cost is higher.

If systems such as Charlotte allowed only blocking communication, a deadlock problem might occur. A server process, such as the fileserver, might be blocked forever by an incorrectly written user process that failed to do a required send or receive. If this happened, no other process would be able to use the fileserver. Though there are ways to avoid this problem without using

non-blocking communication, they involve the overhead of more processes; one way would be to have a dedicated server for each user. This solution would increase the number of processes in a system and may degrade performance if the system is not able to cope with a large number of processes. Since we do not always want to use techniques that prevent timing errors, we need other techniques that will allow us to detect timing errors when they occur.

To detect any program error we need to detect the program's unacceptable behavior. To detect errors, we have to define behavior that is always unacceptable. One such behavior is premature use of message buffers, as in example 1.1 and example 1.2. By using tags, in a method similar to that used in NIL [Stro83], we are able to detect premature use of message buffers. A buffer is marked either *free* or *pending*; it is initially free. A send or a receive causes the buffer to be marked as pending. At completion, the buffer is again marked as free. An exception occurs for a process that either tries to receive into a buffer or tries to use a buffer where a receive is pending. Similarly, if after requesting a send, a process tries to change the contents of the send buffer marked pending, an exception is raised. A process is allowed to initiate multiple sends from the same buffer. The buffer is marked free after notification for all of the sends.

Theorem #10: A program that uses tagged buffers can detect all single-event timing errors.

Proof

The use of tagged buffers will detect all intersections of the use of space by a communication event and all other events. By theorem #1, these intersections are symptoms of all single-event timing errors. Therefore, the use of tagged buffers will detect all single-event timing errors.

There are problems with using tagged buffers in systems such as Charlotte where any variable, array, or record can be used as an input or output buffer. The tagged buffer technique is not free. Additional space has to be provided to mark each data item, and there is added overhead of checking tags on every variable access. A system could be implemented where the microcode would do this checking.

Another incorrect behavior we can detect in distributed programs is deviation from a protocol. The reason for not following a correct protocol can be that the program was written incorrectly or that there is a timing error (as in example 1.5). Other researchers [Baia83, Bate81, Rose85] have discussed ways for specifying correct communication behavior. The communication subsystem can then compare this description with the program's actual behavior and trap if there is a

discrepancy. This method will detect protocol errors and allow the user to search for the causes with a tool similar to TAP (described in the next chapter). There are drawbacks, however; the added checking degrades performance and the specification of the protocol is difficult to write.

Other multi-event timing errors can be detected by the processes themselves when improper conditions arise. This detection is accomplished through user-generated runtime checks testing for inconsistencies in the program. Failing a check causes the program to trap, allowing the user to search for the cause of the detected error.

To detect errors in a process, deviant behavior must be defined for that process. Some of the techniques presented in this section can be used in general situations but detection of many errors depends on the goal of the program and can only be accomplished with help from the programmer.

2.2 Misorderings

It is possible for ordering errors to be deterministic. In this section we discuss errors called misorderings (see figure 2.1). We start out with the following definitions.

□ Depend:

We say a message M *depends* on an event j if j comes before the sending of M in the partial ordering ($j \rightarrow Send(M)$).

□ Most Recent Ancestor:

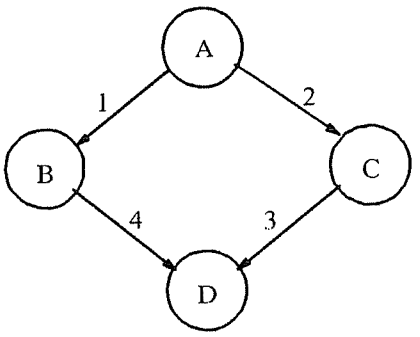
We say an event e is the *most recent ancestor* of a message M with respect to a process P ($e = MRA_P(M)$) if event e is in $\xi(P)$, M depends on e, and there is no event f in $\xi(P)$, $e \rightarrow f$, such that M depends on f.

□ Misorder:

Given any two messages M_1 and M_2 received by process A with $e_1 = MRA_P(M_1)$, $e_2 = MRA_P(M_2)$, and $e_1 \rightarrow e_2$, if A receives M_2 before A receives M_1 then we say A has a *misordering* and the receipt of M_1 and M_2 is *misordered.*

Example

A sends a message to B and later sends a message to C. After receiving the message from A, C does some computations of its own and sends the results to D. After D receives the information from C, it receives some information from B.



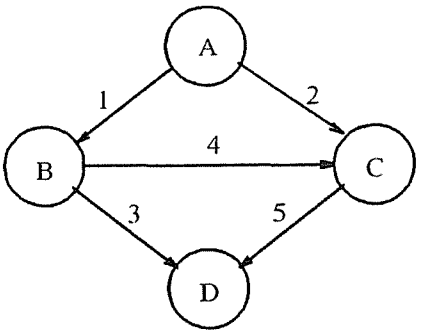
D has a misordering since
$MRA_A(4) \rightarrow MRA_A(3)$.
The receipt of messages 3 and 4 is misordered.

□ Safe Program:

We say a program is ***safe*** iff it is impossible for a misordering to occur. If a program is not safe we say it is ***unsafe***.

Example

A sends a message to B, then sends a message to C. C blocks until it receives the message from B then blocks until it receives a message from A. B sends a message to D, blocks until D has received it, then sends a message to C. Finally, C sends to D.

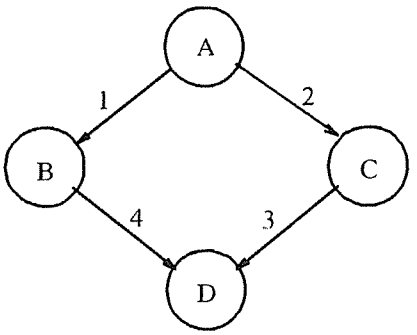


Only C and D can have a misordering since they are the only processes that receive more than one message. For C, $MRA_A(4) \rightarrow MRA_A(2)$. Since C must receive message 4 first there can be no misordering at C. For D, $MRA_A(3) \rightarrow MRA_A(5)$ and $MRA_B(3) \rightarrow MRA_B(5)$. Since D must receive message 3 first there is no misordering at D. Since there are no potential misorderings this is a safe program.

□ Safe Process:

We say a process is ***safe*** iff it is impossible for a misordering to occur with messages received by that process. If a process is not safe we say it is ***unsafe***.

Example



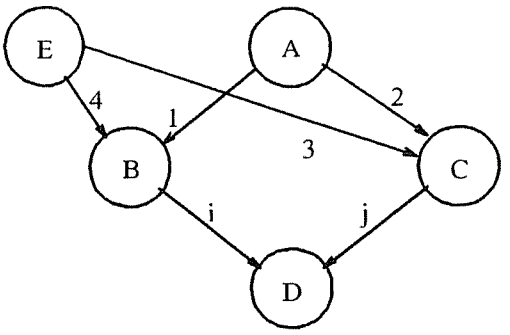
A is safe; D is unsafe.

□ Fatal:

We say a program is *fatal* if a misordering must occur.

Example

A sends a message to B, then sends a message to C. E sends a message to C, then sends a message to B. After receiving both values, B and C send messages to D. No matter in which order D receives the information from B and C a misordering will occur.



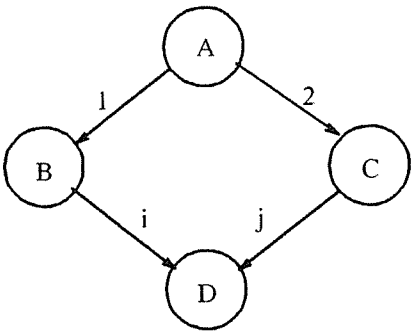
For D,
$MRA_E(i) \rightarrow MRA_E(j)$ and $MRA_A(j) \rightarrow MRA_A(i)$ or
$MRA_E(j) \rightarrow MRA_E(i)$ and $MRA_A(i) \rightarrow MRA_A(j)$.
In either case D is misordered with respect to messages i and j.

□ Dangerous:

If a program is neither safe nor fatal we say the program is *dangerous.*

Example

A sends a message to B, then sends a message to C. After receiving a message from A, B and C each send a message to D. If D receives the information from C before it receives the information from B a misordering will occur. If the message from B arrives first then no misordering will occur.



If message i arrives before message j then there is no misordering.

□ Out-of-order Timing Error:

An ***out-of-order timing error*** is a multi-event timing error where two or more messages are misordered.

Example

GDH (example 1.4)

Using the above definitions we can state the following lemmas and theorems:

Lemma #7: A program is safe iff all of its processes are safe.

Proof

Since an unsafe program must have at least one unsafe process, having all of its processes safe makes a program safe.

Theorem #11: A safe program contains no out-of-order timing errors.

Proof

A program with an out-of-order timing error must contain a misordering. A safe program contains no misorderings and therefore, contains no out-of-order timing errors.

Theorem #12: Unsafe programs allow out-of-order timing errors.

Proof

From the converse of definition of safe program we see that in an unsafe program it is possible for misorderings to occur. If a timing errors results from one of these misorderings then an out-of-order timing error will have occurred. Whether these errors do occur depends on the contents of the messages and whether, in the case of dangerous programs, misorderings actually do occur.

There are two important points to remember here. The first is that not all misorderings are errors. For example, a server gathering numbers to add is not concerned with the order in which they arrive (the threat of overflow notwithstanding). The second important point is that not all misorderings that are errors are timing errors. If the misordering error always happens then it is not a timing error. Fatal programs may differ in which misordering they hit but there will always be a misordering.

Detecting Misorderings

We can detect misorderings by using a variation of Lamport's logical clocks [Lamp78]. We call the following the *vector detection* method.

Each process keeps a count of the number of messages it has sent so far. Each process keeps a vector containing the highest count it has heard of concerning each of the processes in its program. When sending a message, each process includes a copy of its vector. When a message is received by a process the vector in the message (V) is compared with the vector stored locally. Each element in the local vector is increased to the value of the corresponding element in V if the new element is greater than the stored one. If any received element is less than the corresponding stored element then a misordering has occurred. This misordering may or may not lead to an error in the program. The process can then

fail, print a warning, or go on depending on how the programmer sets up the program.

Theorem #13: The vector detection method will find all misorderings.

Proof

Given two messages, M_1 and M_2, received at some process, B, with e_1 being the last event depended on by M_1 in some process A and with e_2 being the last event depended on by M_2 in process A. Suppose a misordering occurs. This means either, (1) $M_2 \rightarrow M_1$ and $e_1 \rightarrow e_2$ or (2) $M_1 \rightarrow M_2$ and $e_2 \rightarrow e_1$. Without loss of generality we will assume (1). The i^{th} component of M_1's vector must contain a value corresponding to e_1 and the i^{th} component of M_2's vector must contain a value corresponding to e_2. Since $e_1 \rightarrow e_2$, the corresponding value in M_1's vector is less than the corresponding value in M_2's vector. Therefore the misordering will be reflected in the values contained in the vectors.

Preventing Misorderings

In non-fatal programs, the possibility of misorderings can be eliminated while still allowing non-blocking communication. To accomplish this, the operating system uses the partial ordering found in timing graphs to guarantee that a program is safe. This is done using a variation of the vector detection

method which we will call the *vector prevention* method.

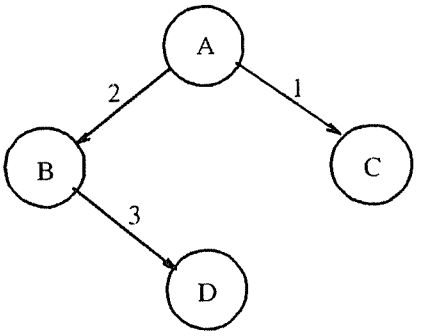
Each process keeps a count of messages sent and a vector of known values of the other processes as in the vector detection method. When a message is received by a process the vector in the message (called V) is sent to all other processes to see if they have elements in their stored vectors that are less than the corresponding elements in V. If no process has such an element then the original message is accepted. Each element in the local vector is increased to the value of the corresponding element in V. If any process questioned has an element less than the corresponding value of V then acceptance of the message is delayed.

Theorem #14: The vector prevention method will prevent all misorderings.

Proof

Since the vector detection method will find all misorderings (theorem #13), and no messages will be accepted if any stored element of any vector in any process in the program could cause a misordering to be detected, then no misorderings will be detectable by the vector detection method and therefore no misorderings will occur.

The vector prevention method will guarantee that no misorderings occur. However, deadlock will occur in fatal programs and can occur in dangerous and



Message$_1$ is sent to C, message$_2$ is sent to B. and then message$_3$ is sent to D. D will ask the other processes for their vectors and discover C has an earlier number in its vector for A than is contained in the message from B. Therefore the message from B will be delayed forever since there are no other messages to update the local vector values.

Figure 2.7: Deadlock with the vector prevention method

even safe programs, as shown in figure 2.7. Even if deadlock did not occur, the overhead of checking vectors and sending extra messages makes this method impractical to use.

2.3 Uses For Time-Dependent Behavior

Processes can have time-dependent behavior without having timing errors.

A server process, such as a terminal handler, might converse with many user processes. The work done for these processes (output to the terminal) will depend on the order in which the server receives requests. This is acceptable as long as no incorrect work can be done. (There is no incorrect way the information can be displayed on the shared terminal.)

A programmer might want a program to perform differently due to time-dependent behavior in the same way that many programs perform differently by using random numbers. Random numbers can be generated by time-dependent behavior and the use of random numbers can be replaced by time-dependent behavior. A random-number generator might derive its values from the sequence of messages it receives. Allowing time-dependent behavior might make it possible to produce a truly random sequence of values. Alternatively, this non-determinism can replace the use of random numbers. Applications that formerly used random numbers might use time-dependent behavior to affect their course of action.

Time-dependent behavior is not always undesirable. A programmer who wants time dependent behavior in a program can purposely use the constructs

that lead to timing errors.

Chapter 3

Finding Causes of Timing Errors

We have implemented a tool, called TAP, to help the user find the causes of detected timing errors. The implementation was done on the Charlotte [Fink83a] operating system. The first part of this chapter is a description of Charlotte. The last part of the chapter is a description of TAP.

3.1 Charlotte

Charlotte is a distributed operating system that provides a powerful interprocess communication mechanism employing duplex *links* to facilitate distributed applications. A *link* is a bound communication channel between two processes upon which messages can be sent, received, awaited, or canceled. Processes may acquire new links, destroy their end of the link, or enclose their end as part of an outgoing message. Thus a *link* is a dynamic capability-like entity that permits only the holder access to its communication resource.

Overview

The idea of using links for interprocess communication was introduced in the Demos operating system [Bask77]. Charlotte has enhanced the idea in several ways. First, a Charlotte link is a duplex communication channel, while a

Demos link allows traffic in only one direction. Both holders of a Charlotte link have equal rights to use, move, or destroy it. Moreover, they can do so simultaneously, independently from each other. In addition, Charlotte does not buffer messages between processes. When both processes are in the same machine, Charlotte copies the data directly from one user's addressing space to another. This decision places the burden of buffer management on the programmer, not the kernel. Experience with Arachne, a predecessor of Charlotte, shows that buffer management at the kernel level can lead to complex deadlock [Fink81].

For applications requiring the transfer of large amounts of data, Charlotte eliminates the constraint on message size. Since buffers are allocated out of user address space, large transfers present no difficulties for the kernel implementation. However, Charlotte does provide a small cache of buffers purely for efficiency. The cache stores messages arriving from the network that do not match a pending Receive.

The ability to cancel is unique to Charlotte. It is important in pipeline protocols. Later stages of the pipeline can send control requests to earlier stages. Those earlier stages cancel their current work and service the control request.

Posting a Send or Receive is synchronous, but completion is asynchronous. Charlotte provides several facilities for dealing with this asynchrony. First, a process may explicitly wait for a Send or Receive to finish. Second, a process

may poll the completion status of a Send or Receive. Third, a process may convert the completion event into an interrupt. (This third facility has not been implemented due to semantic clashes with other features.)

Communication

Charlotte provides five major operations on links. Their descriptions are detailed elsewhere [Fink83a, Arts84]. We summarize their main features here.

Send(transmission link, buffer, buffer size, enclosed link)

initiates a transfer of data from the indicated buffer along the indicated link. The operation remains in progress until its completion is reported by the **Wait** system call. An end of a link may be enclosed in the message. If **Send** should fail, the enclosed link, if any, is restored to the sender.

Receive(transmission link, buffer, buffer size)

allows a message to be received on the indicated link and placed in the buffer. The link identifier *AllLinks* permits the acceptance of a message on any link held by that process. **Receive** on *AllLinks* may not coexist with **Receive** on a specific link because it may cause inconsistency (from user point of view) in choosing the buffer for arriving messages. If the buffer is not large enough to hold the entire message, as much as fits is placed in the buffer and the tail is lost. In all cases both the sender and the receiver are

notified via the completion event (discussed shortly) how much data was accepted.

Wait(transmission link, direction): completion event

queries the result of a previous communication request. The direction may be incoming (**Receive**), outgoing (**Send**), or both, on a specific link or any link. An event descriptor is returned by the kernel. It contains the matched link, direction, completion code, number of bytes transmitted, and whether a new link was acquired. The user may choose to be blocked until the operation completes or to continue immediately.

Cancel(transmission link, direction)

requests cancellation of a **Send** or a **Receive** operation on the specified link. **Cancel** returns an error when the operation does not exist, and failure when the operation has progressed beyond the point where the kernel can stop it. This call blocks the process until the kernel is able to report success or failure.

Destroy(link)

requests that the given link be closed. **Destroy** always succeeds unless the link does not exist or is being transferred. This call will abort all outstanding **Send** or **Receive** requests on that link. It blocks the user until any necessary cooperation by the other kernel has completed. The process at

the remote end will get a failure code from any call related to that link. That end is reclaimed once a remote process invokes **Destroy** on its end. A request to destroy a link being transferred is an error.

Implementation

Charlotte is implemented on the Crystal multicomputer [DeWi84]. It resides above a communication package called the *Nugget* [Cook83], which provides a reliable transmission service. Charlotte's kernels (one per machine) implement the abstractions of processes and links. In order to provide the facilities described earlier, kernels communicate with each other through a low-level communication protocol. Significant events for this protocol include messages received from remote kernels and requests from local processes.

One of the design decisions for the Charlotte operating system is to keep the kernel, which will reside on the each node machine, efficient, concise, and easily implemented. As a result the kernel provides only those services essential to the entire system, such as inter-process communication and process control. All other services are implemented through utility processes, which wait for requests coming from client links. These utility processes include the kernjob, starter, fileserver, connector, command interpreter, and terminal handler.

The kernjob

The kernjob is a utility process that is always resident on every node. It acts

as a representative on its node for programs that need special actions locally. In particular, the starter controlling a node may reside on a different node. It uses the kernjob's ability to make and control processes. A ***control link*** is a special link through which the holder can exercise control over a process associated with the link. Control links are implemented by links to the kernjob and by kernel calls only allowed by the kernjob. To exercise control over another process the kernjob provides the following services to the holder of the control link: ***peek***, get information from the process's address space; ***poke***, change the values stored in the process's address space; ***inspire***, start the process executing; ***expire***, kill the process; ***suspend***, suspend the process; ***resume***, continue the suspended process; and ***gethistory***, return the address of the process's communication history. Another service provided along control links is ***signal***, the controlled process can send a signal (an integer) to the holder of the control link. All these services are accomplished through the privileged kernel calls allowed only for the kernjob.

The starter

The Starter is a utility process that manages the creation of new child processes for the clients. One Starter may control more than one node. To start a process, the client must have a link to a Starter. The clients send the

request to the Starter with a file name from which a new process is to be created. If the Starter succeeds in starting the child, according to the request of the user, a control link, an umbilical link or both the links will be returned to the client as an enclosure in a message. With the request of both links, the control link will go first and umbilical link goes next. The control link allows a parent to exercise some degree of control over its child. The *umbilical link* usually is the link from the newly created process to its parent.

The fileserver

The fileserver provides services to manipulate a file. Any client that needs to read or write files communicates directly with the fileserver after acquiring a link to it from the switchboard. Since Unix files are used in the implementation, the operations are identical to those available under Unix: *open, read, write, create*, and *seek*.

The switchboard

The Switchboard is a utility process designed to allow other processes to find each other and to exchange links. It allows a server process to *register* a link under a given character string name, and it allows a client process to *locate* a registered server and obtain a link to it.

The connector

The connector is a tool to establish initial links in a group of processes. It is implemented as a free-standing utility registered with the switchboard. A program that wishes to initiate a connection episode (usually a command interpreter, but in general any top-level entity in a group of processes) will be called a "parent", and the members of the newly connected group will be called "children". To start a connection episode, the parent should first have a description file in which all the child processes and their interconnections are defined. Then the parent needs to send a request with its description file name to the connector. For most of top-level users which only have the interface with the Charlotte command interpreter there is a Charlotte command to invoke a connection episode with the description file name as an argument.

The command interpreter

Part of Charlotte initialization is to start a command interpreter processes. The command interpreter prompts the user when ready to execute a command. The user may type the name of an executable file with arguments. There is also a ''connect'' command, with which a user can invoke a connection session to a group of processes defined in a description file given by the user as the command argument.

The terminal handler

Another utility process provided is the terminal handler. The terminal handler for a node talks to the user through the console terminal. Processes on any node can connect to any terminal handler to interact with the user.

Charlotte is intended as a vehicle for large-scale distributed computation. Distributed applications can be programmed for Charlotte in several languages. The simplest interface is to use the communication primitives directly from a program written in C or Modula. The kernel of Charlotte and most utility processes are fully operational. Work is progressing in enhancing its programming environment and in implementing a process migration facility.

3.2 TAP

In chapter 2 we discussed reasons for not using the mechanisms to prevent timing errors and we discussed ways to detect timing errors. Here we introduce a tool, TAP, that a programmer can use to find the causes of timing errors that have been detected. We have not implemented a detection mechanism other than normal exception handling; all other detection must originate within the user's program. To help the user find the causes of timing errors we exploit the partial ordering of events implicit in timing graphs as mentioned in Chapter 1. Our goal is to use a history-saving mechanism to collect enough information at runtime to

construct the timing graph if needed. This information is stored in the process's *communication history* and used by an interactive tool, called TAP, which traverses the graph, showing the programmer the order in which events occurred. The programmer can then look at this ordering to find the events that occurred in an unacceptable order.

There are two main problems involved in finding the cause of a timing error. First, we never know when a timing error might occur. Second, changing a program to help discover the cause of a timing error changes the relative timings and therefore might mask or unmask timing errors. To solve these problems we advocate keeping the history-saving mechanism active at all times. The final version of a program is, therefore, the same as the debug version. We feel that the history-saving mechanism is like subscript checking; they both have a cost but they are both worth doing most of the time. Because the history-saving mechanism is always active, performance degradation must be minimal. A user can disable this mechanism if a speed up of the program is desired. The tool described here requires the history-saving mechanism.

In the rest of this section we will discuss the tool (TAP) we developed, the details of implementation, and obstacles to building an effective tool.

The tool (TAP)

TAP is a tool similar to a postmortem debugger and is the user interface for examination of the timing graphs. With TAP, a user can look at the information at the current node in the timing graph or move along an edge to another node. The new node then becomes the current node. For ease of use, TAP provides commands to let the user skip to nodes not joined by an edge to the current node. TAP has control links to all of the user's processes and, therefore, can get at all of the information stored in the process's communication history.

A copy of TAP is loaded with the user's program and is given the control link for each user process by the connector. After initialization, TAP blocks, waiting to be activated. It can be activated by the user typing a special break character to the terminal handler or by a signal from a user process over its control link.

When TAP is activated it suspends all user processes to which it has control links and waits for orders from the user. With TAP, the user can step through the timing graph reflected in the saved communication histories to see the order in which events occurred and the contents of messages. This information should help the user find the cause of timing errors that have occurred. Examples of TAP commands are shown in figure 3.1.

```
c     - continue user program (quit TAP)
h     - halt user program
n     - next record in this history
o     - go to the other end of this message
p     - previous record in this history
q     - quit
r     - reset pointer to most recent entry
s     - status: look at communication history
w     - who: show nickname of current process
N     - Next record in history on this link
P     - Previous record in history on this link
R     - Repeat last search (n, p, s, N, P, R)
W     - who: show nickname and link information for all processes
.     - Repeat last search (n, p, s, N, P, R)
```

Figure 3.1: Examples of TAP commands

TAP can be used to find non-timing errors that manifest themselves in messages. In a system like Charlotte, the content of messages is input to the receiving process and output from the sending process. If a process does not adequately handle some input, then looking at the communication history can show the programmer both where in the code the input was received and what the input contained. For example, in a case statement if there is no case for a particular value received in a message, then that value will be in the communication history and the programmer will not have to supply a print statement to reflect the bad value. Similarly, looking at the messages sent by a process can clue the pro-

grammer both that possible errors exist and where in the program they might be in the same way that printed output can give similar clues. Traditional debuggers can provide access to the same information using breakpointing and displaying variables but the processes in question would have to be recompiled to support debugging and the program would have to be rerun.

We say message A is the ***cause*** of message B if the input of A to a process leads to that process producing B. After finding an error in a program, for example the sending of an incorrect message, the programmer can use TAP to find the cause of the error. We can look at the conversations in which a process is engaged to see if the contents of previous messages might have caused the observed deviant behavior. This is similar to Miller's [Mill85] use of causality to identify commonly traveled paths in server processes.

Examples of the use of TAP and the results of experiments done with TAP are shown in chapter 4.

Implementation

Changes to the runtime support

The Charlotte operating system is written in Modula using the UW-Modula compiler [Fink83b]. For this research, it is assumed that user programs are also written in Modula and are compiled by the UW-Modula compiler. Programs

written in other languages could benefit from TAP with modification to their run-time support. The startup code was modified to notify the kernel of the address of the user process's communication history. To maintain transparency while allowing user processes to activate TAP, the Modula runtime was also changed to catch exceptions (such as an attempt to divide by zero) and to notify TAP of the exception by sending a signal over the control link. Also included in the runtime are user-callable procedures to notify TAP of the process's user-defined name.

Changes to operating system

Charlotte was changed in many ways. First the kernel was altered to keep a record of all request, completion, and report events. To accomplish this goal, a system call was added to allow the user process to pass its communication-history address to the kernel. Whenever a communication request, completion or report occurs, the kernel saves information in this communication area for TAP. The information saved by the kernel includes: the type of event (send, receive and so on), a walkback of the process's stack at the time the event actually occurred, the local and remote link numbers associated with the event, the machine and process at the other end of the link, a count of this type of event on this link, and the first twelve bytes of the message.

Second, the kernjob was modified to allow signals on control links so a process can let its parent (TAP) know when an exception occurs. New requests were added for control links to help TAP in its work. These new requests include: get the address of the process's communication history and get the process's machine number and process identifier. New system calls were added to the kernel to help the kernjob perform these tasks. TAP uses other kernjob requests that were already in existence, these requests include: suspend the process, resume the process, and peek at a memory location.

Third, the connector was changed to start TAP, give it the control links to the user's processes, and connect it to a terminal handler.

Finally, the terminal handler was modified to talk to TAP when a special break character was hit. A user can be interacting with the application program and on noticing an error switch to TAP by hitting the break (^B) key.

Unimplemented Features

TAP was created to investigate the use of timing graphs in finding timing errors in distributed programs. We did not deal with many other problems in distributed debugging. A usable distributed debugger should provide all of the features of a traditional debugger as well as features needed to handle problems arising in a distributed environment. These traditional features, such as break-

points, single stepping, and the ability to look at current values of the program's variables, were not provided here.

We also did not investigate the issue of user interface. For a distributed environment, a debugger might provide windows so that the user can concentrate on various areas of activity without being burdened with too much information.

On a system like Charlotte, a debugger might display the user's program in graphical form with the processes as nodes and the links as edges between the nodes. The debugger could use the saved information in the communication history and display the communication activity on this graph. Whenever a message is sent, the debugger could display the message moving along the edge of the graph. The user could watch this display and quickly get a global view of the interaction between processes. The debugger could also provide the means to zoom in on a section of the graph to concentrate on the interactions within a subset of the processes.

Another graphical form in which the debugger might display the saved information is as the timing graph. Given a global view of the timing graph and the ability to zoom in on particular areas of the graph might provide the user with a view of the stored information that would be more effective in finding errors. The user might be able to recognize proper and improper patterns of communication from a global view that would not be evident just stepping through the timing

graph.

Currently up to 100 messages are saved in a process's communication history. Storage for the communication history adds 6488 bytes to the user's data area. Though saving 100 messages has proven useful, it is possible to save the entire user communication history by writing the buffer out to disk before overwriting any information. This technique was not tried partly because the added overhead would not be insignificant. When the communication history is full, a process could suspend itself and send a message to TAP. TAP could then issue a Peek to grab the entire history, continue the process, and write the history to disk. Every time a communication history is full there would be five extra messages (notify TAP, peek, get the results of the peek, continue the process, and send the message to the fileserver to write out the buffer) and the extra overhead of having TAP run. Since for each message activity three events are stored in the communication history (request, completion and report), the number of message activities needed to fill a history is one third the number of slots available in the history. Two processes communicating only with each other would record the same amount of information. Their histories would both become full at the same time and both would be written to disk. With this method, the message overhead is 10/((size of history)/3) or 30% in the case of a history size of 100. There is also the overhead of having TAP run and the lost time while the

each process is suspended.

TAP, as developed, is only useful for programs run from a terminal. Programs run batch, not connected to a terminal, would not benefit from TAP. TAP could be modified so that, upon receiving a signal from a user process, TAP would suspend all user processes and save their images in disk files. The user could later use TAP to inspect the saved files to investigate the cause of the program's failure.

Examples of the use of TAP to find program errors are shown in the next chapter.

Obstacles

There are obstacles that can potentially limit the effectiveness of the approach described here. One obstacle is the size of main memory. Since this size is finite, only a finite amount of information can be kept in the communication history. Once the space allotted for this history fills, old information is lost as newer information arrives.

Another obstacle is the difficulty in suspending all processes at the same instant. Many messages might be exchanged between processes from the time an error is discovered until all processes have suspended. Communication events occurring before an error discovery are probably more meaningful to finding the

error than are communication events occurring after the error discovery. If the space allocated for the communication history is too small then meaningful information could be lost before all processes are suspended.

Another obstacle in Charlotte [Fink83a] is the difficulty in knowing the exact moment a communication event occurs. Charlotte runs on top of a low level communication subsystem. The time Charlotte finds out a message has arrived or gives a message to the communication subsystem to be delivered may not be the actual time the event occurs. Therefore, the order in which Charlotte thinks events occur may not be accurate. For example (see figure 3.2), after a process makes a send request, Charlotte might pass a pointer to this message to the Nugget and record a send completion in the process's history. The Nugget might then delay in delivering the message out of the user's address space to the remote machine. If the user modifies the message between the time Charlotte

---

```
Send(X, buf);
    .
    .          ← Charlotte thinks the send completes here
Change buf;
    .          ← the send actually completes here
```

Figure 3.2: Charlotte's misconception of when a send occurs

---

records the send completion and the time the lower level actually sends the message, then a timing error will occur but the error will not be reflected in the communication history. A more conservative recording of send completion can be achieved by recording the completion when the Nugget notifys Charlotte that the message has been sent.

Another obstacle to deal with is the time needed to collect the information for the communication history. This time is significant and, if too large, can make using the history saving mechanism unattractive.

The final obstacle, common to many source level debuggers, is the reordering of machine level instructions by the optimizing compiler. To show the relative order of all events, the program counter (PC) is recorded for all communication events. This value can later be compared with the addresses of the machine instructions that correspond to source level instructions in the appropriate process. The source level instruction nearest the recorded PC can be determined to find the relative order between communication events and local events. If the code for a process is optimized, then knowing the PC of a communication event and a local event's source level statement will not necessarily show the actual order of occurrence between the events.

Despite these obstacles, using the communication-history mechanism with TAP is an effective method for finding the causes of timing errors.

Chapter 4

The Experiments

In this chapter we discuss the results of using TAP. At the end of this chapter we discuss the impact that keeping the communication history has on a running program.

To gauge the usefulness of TAP as a debugging tool various experiments were run to see how easily TAP could be used to find the causes of timing errors. The problems considered include a synchronization error (as discussed in example 1.5), requesting two receives into the same buffer (a combination of example 1.1 and example 1.2), and arrival of messages in an unexpected order (as discussed in example 1.4).

4.1 Synchronization

The problem

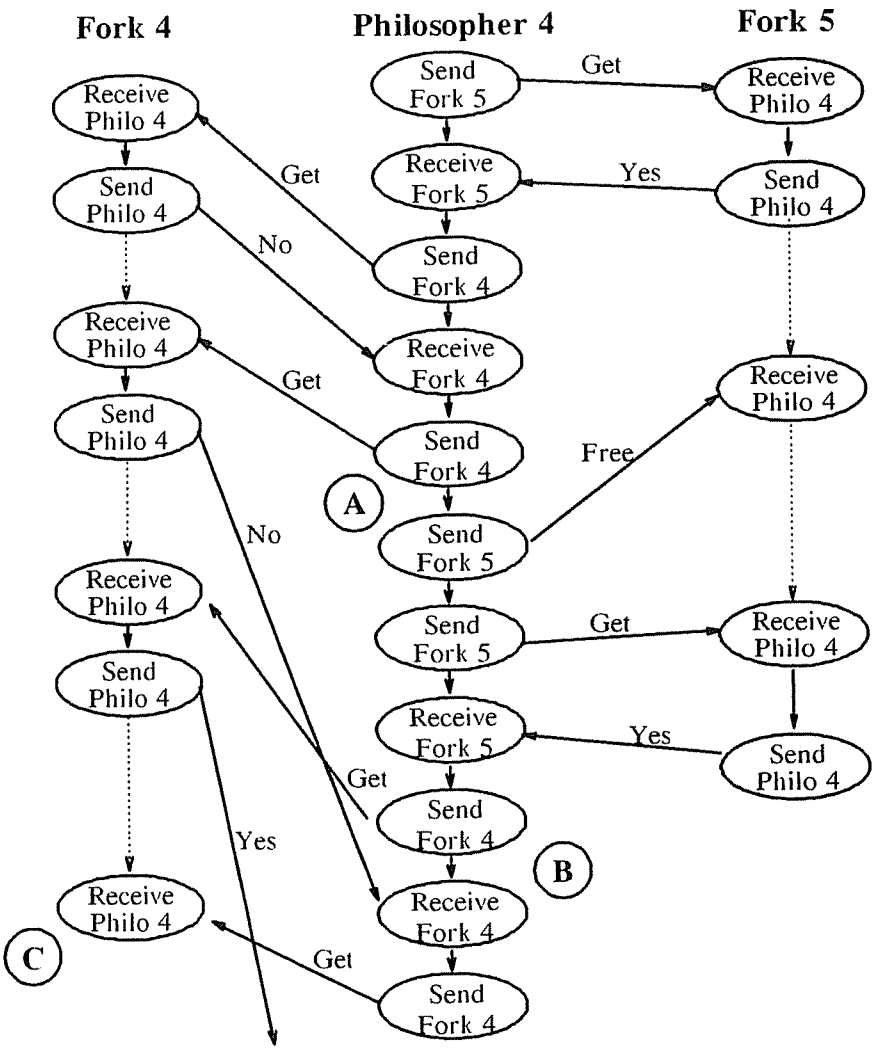
To test the usefulness of TAP in finding synchronization bugs, a version of dining philosophers was written that had the potential of becoming unsynchronized. Ten user processes were created, five philosophers and five forks. The algorithm used called for each odd philosopher to attempt to pick up its left fork before picking up its right fork. The even philosophers picked up the right fork

first. To pick up a fork, a philosopher sends a ''get'' message to the fork. The fork then replies ''yes'' or ''no.'' The philosopher would repeatedly send the request to the fork until a ''yes'' message was returned or until starvation occurred. To return a fork, a philosopher would send a ''free'' message to the fork. The fork would respond with ''yes'' if it accepted the return or ''no'' if the philosopher had not been granted the fork. At starvation, a philosopher returns any forks he has obtained and, after a short wait, begins again to try and get both forks.

The error in the philosopher process is in the following code where the philosopher attempts to obtain a fork.

```
while (not starved) and (Receive(fork) <> 'yes') do
        Send(fork, 'get');
        Check(starved);
end;
if not starved then Use_Fork;
else Starve;
```

The problem arises when the philosopher starves; the logical expression in the 'while' statement short circuits and does not do the Receive. The fork, on the other hand, has answered the previous request. This answer will be received after the philosopher is rejuvenated and tries again to obtain the fork. Once the fork has been granted the philosopher will not know it since he is one message behind. The subsequent request will cause an error since the requester has already been granted the fork. This interaction can be seen in figure 4.1.



A - Philosopher 4 starves and fails to receive the reply from Fork 4
B - Philosopher 4 receives the previous "No" instead of the "Yes"
C - ERROR: Fork 4 receives "Get" from the holder of the fork
(Dotted lines indicate some intervening events have been omitted)

Figure 4.1: a portion of the dining philosopher's timing graph

This error is a timing error because it will only show up under certain timings between the processes. Specifically, the fork process is doing a non-deterministic receive (receive(any)) to get requests from the philosophers. This allows various orderings between the requests of the philosopher who gets the fork and later returns it and the other philosopher who keeps requesting the fork. This variability in ordering leads to a timing error since the actions of the conflicting sends are not independent. Without this non-determinism the philosophers would always hit the error or never hit the error. With the non-determinism the error is only occasionally hit.

Finding the Cause

The error was detected by a fork process (in the experiment, Fork4). Using TAP the fork's history was examined and the offending philosopher (Philo4) was discovered (see figure 4.1). Checking the philosopher's communication history revealed that two requests for the fork were sent without an intervening receive request to find out if the fork was granted. Knowing the line number in the code from where the requests were made led to finding the bug. The interaction with TAP is detailed in Appendix A.

4.2 Use of One Buffer for Two Receives

The problem

For the second experiment we ran a program made up of three processes A, B and C. Process A named the same buffer for two receive requests, one for a message from B and one for a message from C (See figure 4.2). Process A then waited for the first message to arrive. The message was supposed to contain four identical integers. After dealing with the first message A waited for the second and ignored it.

Finding the Cause

Process A detected the error itself. After receiving an array of four integers it tested them twice (lines 7 & 12) to verify they were identical (see figure 4.2). When it turned out they differed the process signaled TAP. Using TAP we found that the messages process A received completed at three points in the code. Two of those points were the points A waited for the receive to complete (lines 5 and 16) and no errors occurred when the receives completed at these points. The third receive completed between the two waits (line 12), overwrote the information being used by the process, and led to the error. The interaction with TAP is detailed in Appendix B.

```
1)    i       := 1;
2)    while i < 1000 do
3)          err    := Receive(b,adr(msg),size(msg));
4)          err    := Receive(c,adr(msg),size(msg));
5)          err    := Wait(ALL_LINKS, Received, result);
6)          num    := msg[1];
7)          if (msg[2] <> num)  or (msg[3] <> num) or
8)                                 (msg[4] <> num) then
9)                SignalParent(1); dummy    := Suspend(-1);
10)         end;
11)         num    := msg[1];
12)         if (msg[2] <> num)  or (msg[3] <> num) or
13)                                 (msg[4] <> num) then
14)               SignalParent(1); dummy    := Suspend(-1);
15)         end;
16)         err    := Wait(ALL_LINKS, Received, result);
17)         inc(i);
18)   end; (* while *)
```

Figure 4.2: process A expects 2 receives in the same buffer (msg)

4.3 Unexpected Order

The problem

The third experiment tested the usefulness of TAP to help detect errors caused by messages arriving in an unexpected order. For this experiment we wrote a program that consists of four processes, an accumulator process and three worker processes. The accumulator process issues a Receive(Any) in a

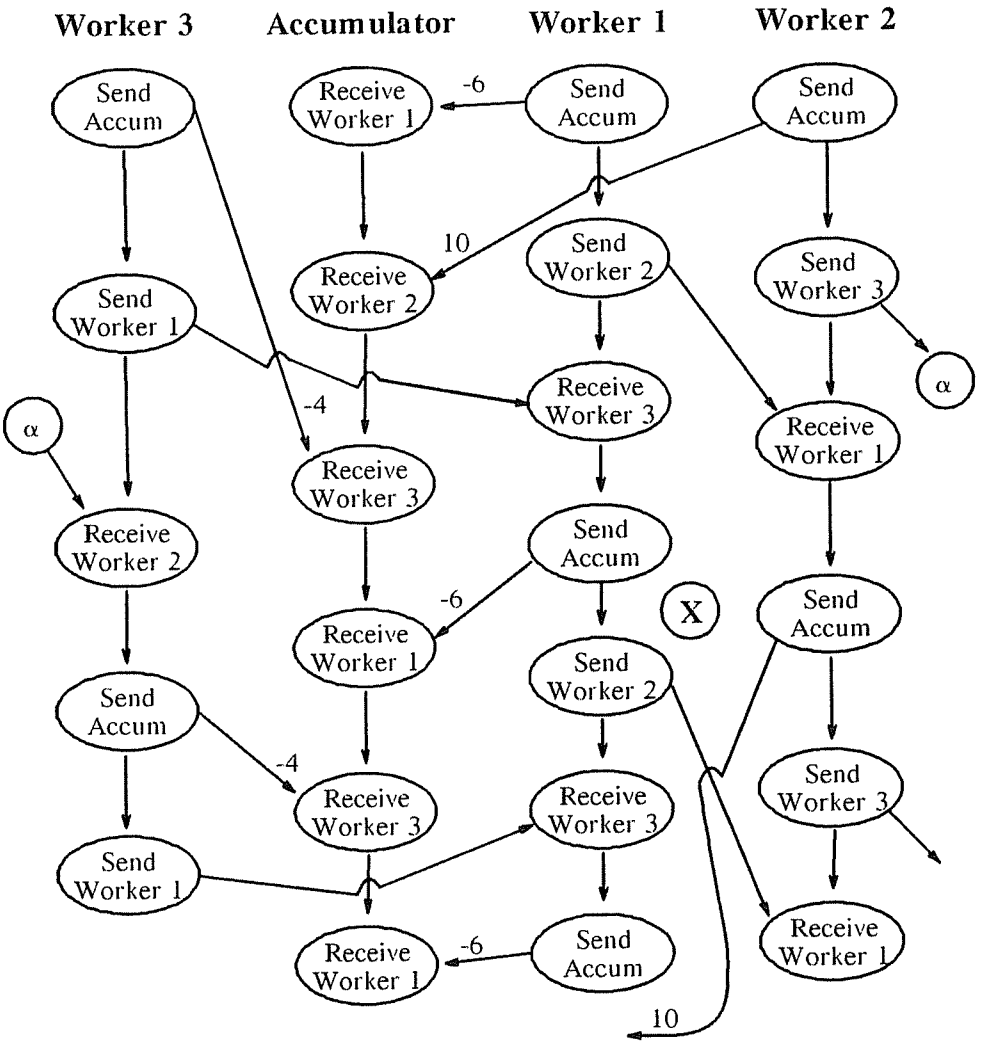
loop, gets values from the other processes and adds their values to a global total. This total is initialized to 10 and it is an error for the value to fall below 0. The three worker processes send the values -6, -4, and 10 to the accumulator, exchange messages with each other and send their values to the accumulator again. It was hoped that the exchange of messages between the worker processes would synchronize them between rounds to prevent any one of them from getting too far ahead of the others. This synchronization attempt failed. An error occurred causing the accumulator's value to fall to -6.

Finding the Cause

The accumulator process detected the error (see figure 4.3). Using TAP showed the accumulator received the values -6, -4 and -6 from $worker_1$, $worker_3$ and $worker_1$ respectively without receiving a value from $worker_2$. The synchronization did not keep the workers close enough together. The interaction with TAP is detailed in Appendix C.

4.4 The Impact of Keeping a Communication History

Six programs were run to measure the performance degradation attributable to the history-keeping mechanism. For all programs the system time was recorded before and after loops which contained nothing except communication



X - With non-blocking send Worker 1 proceeds without waiting for Worker 2

Figure 4.3: the timing graph for the accumulator/worker problem

activity, necessary work to sort out what communication activity to do next, and loop overhead. The amount of time taken for non-communication activity accounts for less than The complete results for the first two programs are summerized in the table in figure 4.4. Results of all of the tests are summerized in the table in figure 4.5. The follow are descriptions of the programs that were run.

(1) A program was run consisting of a two processes on one machine, a producer and a consumer. The producer sent the same 100 byte message 10,000 times to the consumer. This program was run under three different versions of Charlotte. Version 1 is Charlotte without the history mechanism; version 2 is Charlotte with the history mechanism present but disabled; version 3 is Charlotte with the history mechanism active. As can be seen

Time per Activity(in ms)	Location	Version
20.3	Same Machine	No history mechanism (version 1)
20.7	Same Machine	History mechanism inactive (version 2)
25.2	Same Machine	History mechanism active (version 3)
29.5	Different Machine	No history mechanism (version 1)
30.9	Different Machine	History mechanism inactive (version 2)
33.0	Different Machine	History mechanism active (version 3)

Figure 4.4: Times for Communication

	NH	HA	NH/HA
1) Producer/Consumer - same machine	20.3	25.2	.806
2) Producer/Consumer - different machines	29.5	34.1	.865
3) 1 Producer - 2 Consumers	24.7	27.2	.908
4) 1 Producer - 3 Consumers	24.8	27.3	.908
5) 2 process conversation	29.0	29.7	.976
6) Client - Server	28.7	31.5	.911

[HA] - History-saving-mechanism active
[NH] - No history-saving-mechanism

Times in milliseconds per communication activity

Figure 4.5: degradation due to keeping a message history

from the tables, the presence of the history mechanism causes a slight degradation of performance and an active history mechanism can drop performance to 80% of the time taken for communication within a machine. We feel that this program suffers the most performance degradation since both processes are delayed by the other's extra communication overhead.

(2) The same program was run with the producer and the consumer on different machines. Degradation of performance due to the history mechanism

was less here than the degradation on one machine. Here performance dropped to 86% of the time taken for the same program with no history saving mechanism.

(3) This program consisted of three processes on different processors. One process (the producer) sent a message to the other two processes (consumers) then entered a loop. In the loop the producer waited for any send to complete, then sent another message to the corresponding consumer. Since blocking communication was not used, some of the overhead of recording information in the communication history occurred in parallel with other events. Here performance only dropped to 91% of the time taken for the same program with no history saving mechanism.

(4) This program was identical to the previous one except there were three consumers. The results were the same as in the previous example.

(5) This program consisted of two processes on different machines engaged in a conversation. Each process requested both a send to and a receive from the other then entered a loop where they would wait for either activity to complete before rerequesting the completed activity. There was almost no performance degradation here since both processes recorded the necessary history information when they would otherwise have been waiting.

(6) Consisting of two processes on separate machines, the final program followed a client-server model of communication. The client process blocked sending a message to the server then blocked after a receive request to get the reply. The server blocked on a receive request from the client then blocked on a send request to the client. The performance dropped to 91% of the time taken for the same program with no history saving mechanism.

The last four programs were only run with their processes on separate machines because, running on the same machine, their results would have been the same as for the first program since no parallelism is gained from running on the same processor.

These measurements are for communication activity only. A distributed program that does nothing but communicate would have its performance dropped as much as 20%. Programs that communicate little would suffer a much smaller degradation in performance since only communication activity is affected.

Chapter 5

Suggestions

In chapter 2 we have characterized ordering errors and shown necessary conditions for their occurrences. Based on these findings we will now make suggestions for writing distributed programs, designing languages for distributed programming, and writing operating systems for distributed programming.

5.1 Suggestions for Writing Distributed Programs

In section 2.1 we showed that all timing errors can be prevented by the use of proper programming techniques and system design. We also showed that this prevention had a cost in loss of parallelism and flexibility. A programmer writing a distributed program should consider the needs of the program being written and only deviate from the techniques for preventing timing errors if it is important to the program.

When writing distributed programs, a programmer should attempt to adhere to the following six points.

(1) **Avoid asynchronous completion.** The use of asynchronous completion allows single-event timing errors to occur. As shown in chapter 2, either blocking communication or buffering in the communication subsystem

prevents asynchronous completion. The latter allows more parallelism but if the system being used does not provide a proper buffering mechanism then blocking communication should be used.

(2) **Use specific request and report on private channels.** The use of specific request and report on private channels guarantees that all matches are deterministic, thereby avoiding many multi-event timing errors. The use of both receive(any) and wait(any) leads to non-deterministic match and should be avoided.

(3) **Avoid asynchronous report.** If non-blocking communication is being used, then use blocking report (wait) instead of interrupt report. With interrupt report not only is the buffer used for communication open to conflict but so are all of the variables referenced in the interrupt routine. Independent sections would be hard to establish since there is no point (other than the end of the process) by which we can say completion and report have occurred.

(4) **Avoid polling.** If non-blocking communication is being used, then use blocking report (wait) instead of polling. As with interrupt report, with polling we cannot always determine the exact limits as to where completion can occur. Without knowing these limits it is difficult to establish independent sections.

(5) **Limit the sections of the program in which timing errors might occur.** If the other points cannot always be followed, then limit the sections of each process in which timing errors can occur. Since timing errors can only occur in sections of a program where asynchronous communication events can occur, these are the sections that should be limited. Following this advice is helpful for two reasons. First, the program can be manually scanned to see the effect time-dependent behavior will have. Second, if a

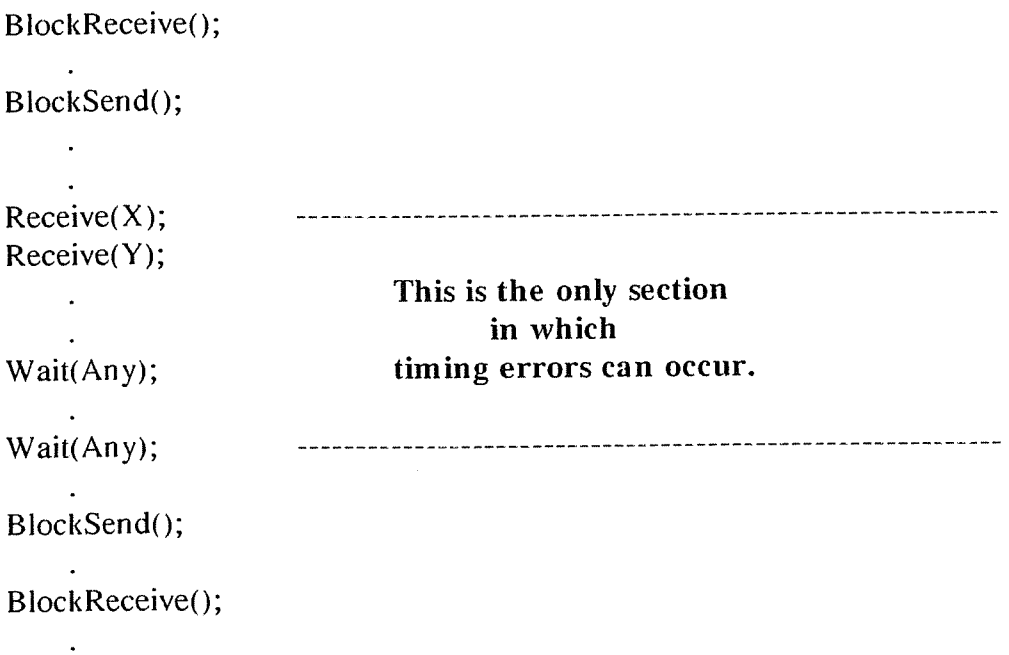


Figure 5.1: limited section where timing errors can occur

timing error does occur then the programmer has a limited area of the program to check to find the cause of the error. For example, see figure 5.1. If blocking communication is used except in the section shown then timing errors can only occur in this section.

(6) **Share as little space as possible.** Each channel or communication request should use a personal buffer. This buffer should only be used by other events that are affected by or that affect communication. Examples include events that fill the buffer to prepare for a send or events that use the information in the buffer after a receive.

By following the first four points, it is possible for a programmer to avoid timing errors. By following the last two points timing errors that cannot be avoided will be contained, thereby making debugging a manageable task. If the programmer structures the program's processes correctly, then for each process, regions can be isolated in which timing errors can occur. For example, nonindependent sections A and B can be analyzed by the user to find the cause of the nonindependence. The user can then either

(1) restructure the process to make the sections independent (if possible)

(2) put in checks to catch unwanted situations

(3) use blocking communication to force the desirable order for the sections

(4) or leave the process alone if the nonindependence is acceptable (for instance, both sections add the received value to a global total whose final value is used after both sections have executed).

5.2 Suggestions for Languages for Distributed Computing

A language for distributed computing should allow the programmer to easily write timing-error-free programs. To accomplish this goal, synchronous completion and deterministic match should be provided. However, a language and a compiler for distributed computing should provide much more than these features. With the right data types and language constructs, a compiler can check a program not using these features to insure timing errors cannot occur.

The tagged buffer method of timing error detection, described in chapter 2, can be implemented by the compiler to detect situations where some timing errors can occur. The programmer can be informed of these situations and even told in which statements the errors might occur. In NIL [Stro83], the compiler provides tags for variables and gives a compile-time error message if communication buffers could be used before completion is guaranteed to have occurred.

A language could also provide the programmer with a mechanism to describe sections believed to be independent. The compiler could then check for this independence and warn the programmer when independence is in question.

Runtime checks could also be provided to discover independence violations. An example of a language construct to declare sections independent is shown in figure 5.2.

Another language construct that would help a programmer avoid timing errors is suggested by Scott in his language LYNX [Scot84]. LYNX, a language for writing distributed programs, provides different threads of control in each process. Variables can be shared between threads of control or can be local to a particular thread. Assuming the programmer uses a different thread of control

---

Independent Sections begin
    guard section 1:
        Action for section 1
    guard section 2:
        Action for section 2
end Independent Sections

Independent sections are contained in a block of code delimited by the statements "Independent Sections begin" and "end Independent Sections." Each section has a guard, which is a logical expression, and an associated action. Once the block of code is entered each section will be executed exactly once before the block is exited. The order in which the sections are executed is undefined but a section can only execute if its guard is "true."

Figure 5.2: Independent sections

---

for each channel, then if no variable is shared between threads, or if the use of every shared variable is independent of the order of events between threads, then no multi-event timing errors can occur.

This use of threads can be modeled with independent sections as shown in figure 5.2. Each thread is divided into subthreads and switching between threads can only occur at subthread boundaries. A process would contain an independent section for each subthread and a guard for each section would contain a boolean set to ''true'' by the preceding subthread. A compiler for this LYNX-like language could warn the programmer if shared variables are used in such a way as to permit multi-event timing errors. All variables shared between threads would have to be read-only, write-before-read or unused in each section.

The use of a LYNX-like language does not prohibit the user from writing programs open to timing errors but it structures the program to reflect the intent of the programmer and makes writing timing-error free programs an easier chore. The programming language is the programmer's vehicle for expressing the program's desired behavior. For communication on a multicomputer, a language should reflect the positive traits of the underlying system to the user, mask the undesirable traits and create the missing necessary traits. With help from a programming language, writing efficient timing-error-free programs can be a manageable task.

5.3 Suggestions for Distributed Operating Systems

An operating system for distributed processing on a multicomputer should have the facilities to allow a programmer to write timing-error-free programs and guarantee that the time-dependent nature of the information it keeps for a process is reflected back to the process appropriately. These facilities allow a process to control the order in which it handles events. To accomplish the goal of providing these facilities, such an operating system should guarantee that messages sent from process A to process B on channel C are delivered in the order sent. When a receive request can match more than one send request, the operating system should guarantee the receive request matches the first send whose message arrives at the receiving processor. Similarly, if there is a choice of events to report (as in wait(any)) then the earliest locally-known event should be reported.

Other facilities such an operating system should provide include those that would potentially give a program more parallelism. These facilities include run-time tagged buffers and message buffering in the communication subsystem. As described in chapter 2, tagged buffers, monitored by the microcode, would allow a program to detect all single-event and some multi-event timing errors without having to use blocking communication. Similarly, as described in chapter 2, message buffering in the communication subsystem would allow a process to avoid timing errors without blocking for receive and only blocking for send until

the message is copied to the local communication subsystem. This added copying has a cost but there would be a benefit since the applications processes would then be free to execute. Both runtime tagged buffers and buffering in the communication subsystem permit greater parallelism than does blocking communication. Proper use of buffering prevents all timing errors while the use of tags detects many timing errors.

An operating system for distributed processing on a multicomputer should also help the user debug distributed programs. To this end, the operating system should keep a trace of the communication between processes and provide an easy way for the user to get at the saved information. One such way would be to provide system calls which allow a process to examine the communication history of another process. The operating system should also provide an efficient way to obtain information concerning the user processes, such as the identity of communication channels in use or a walk-back through the user's stack. These facilities allow a tool such as TAP to be developed and provide an efficient way to gather the information needed by the tool.

Chapter 6

Conclusion

6.1 Summary

This research has investigated ordering errors in distributed programs with an emphasis in five areas.

Characterization of communication

In chapter 1 we characterized communication between distributed processes in three ways. First, we decomposed communication activities into their subparts (communication events) and focused on request, completion, and report. Second, we discussed the different dimensions of blocking communication. Third, we characterized communication channels.

Characterization of timing errors

In the first part of chapter 2 we focused on timing errors. We separated timing errors into two types, single-event timing errors and multi-event timing errors. We used our model of communication to show necessary conditions for each type of timing error. These are asynchronous communication events and shared space for single-event timing errors and non-deterministic order of occurrence and shared space for multi-event timing errors. We then showed

ways to prevent the occurrence of timing errors by ensuring at least one necessary condition was violated for each type of timing error. We discussed reasons for not preventing timing errors and techniques for detecting their occurrences.

Characterization of misorderings

In the latter part of chapter 2 we focused on another ordering error, misorderings. We defined misorderings, showed examples of them, showed how they can be detected using the vector detection method, and showed how they can be prevented using the vector prevention method.

Development of a methodology for finding causes of timing errors

In chapter 1 we described timing graphs and showed how the partial ordering of events inherent in the timing graphs can be exploited to discover the cause of detected timing errors.

Implementation of a tool based on that methodology.

In the last part of chapter 3 we described TAP, a tool we have implemented. TAP uses the ideas about timing graphs to help the user find the cause of timing errors. In chapter 4 we describe the results of the experiments we conducted using TAP.

6.2 Ideas for Future Research

In this thesis we have characterized ordering errors and shown necessary conditions for their occurrences. Future research in this area should concentrate in four areas: expanding the theory to cover more communication primatives, language design, timing-error detection, and user interface.

Expanded Theory

The theory of detecting and preventing timing errors can be expanded to include other communication primitives such as broadcast (send to many) and cancel (telling the communication subsystem not to do the requested action if it has not yet been done). Broadcast almost fits the model now but it has many completion phases. The addition of cancel to the model would introduce new areas of conflict outside of a process's address space and would add complexity to determining request-report matches.

Language Issues

Future research on programming languages for distributed computing should provide the means for programs to exploit the parallelism available on a multi-computer while still avoiding timing errors. An area to concentrate on is methods for automatically guaranteeing sections are independent. Such a method might come from code optimization techniques such as those using data flow

analysis [Aho79]. The idea of independence can also be expanded to include the mutual use of variables in different sections of code such that the behavior of the program is acceptable no matter in which order the sections execute. This version of independence depends on the semantics of the program and would be much more difficult to automate.

Timing Error Detection

As stated earlier, timing-error detection is difficult because it depends on the system knowing when the program's behavior is wrong. In chapter 1 we mentioned some techniques that have been developed to specify the expected behavior of a program. More research needs to be done in this area.

More research is also needed in the area of implementing some of these detection ideas. As in the case of recording the communication history, any timing-error detection mechanism will have to be active all of the time. For these techniques to be useful they have to have a minimal impact on the execution of the program.

User Interface

The user interface to TAP is poor. Research is needed to find ways to make the user's debugging task more efficient. For example, the user might benefit from an overview of the communication. Displaying a large section of the timing graph might help. Another useful feature might be an analysis of the programs

communication patterns by the debugger. Areas where the communication patterns change might be areas following timing errors.

Chapter 7

APPENDICIES

This section contains the transcripts of the three debugging sessions discussed in Chapter 4. The boldface type following prompts is the response tot he promt. Prompts with no response use the default value enclosed in brackets ([]). Comments have been added in boldface type in the right margins.

7.1 Appendix A

The following is the transcript of the session where the dining philosopher program was run.

```
Charlotte(vax), Version 2.5.1 (new nugget), Fri Mar 29 11:50:54 CST 1985

Please give your Configuration File name: /tmp/ajg

charlotte_1 => cd dp2

charlotte_2 => connect C

TAP: going to work                          An error has been detected by process #9

Distress signal from process #9

[HALT] Command: s                           Look at process #9's history
[10] Which Process? (Enter logical process id): 9
[0] Local link number (0 for any):
[0] remote link number (0 for any):
[0] remote machine number (0 for any):
[0] remote process number (0 for any):
[0] link count (0 for any):
[any] Message Type: ([a]ny,[s]nd,[r]ec,[d]es,[R]wait,[S]wait,reqre[c],reqse[n]):
[backward] Direction to search ([f]orward, [b]ackward, [s]ame, [r]everse):

found at 44
WAIT ON SEND:      count = 26,
 loc_link  rem_link  rem_mach  rem_proc
   21        28        1         6
```

```
PC walkback =    b21    c49    2d9   1ed3     0     0     0


[STATUS] Command: p

found at 43
Saved info is    F  o  r  k     #  4  ,     G  E  T
              70 111 114 107  32  35  52  44  32  71  69  84
SEND:              count = 26,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    2d9   1ed3     0     0     0
```

Last message sent was an error message

```
[HISTPREV] Command: p

found at 42
Saved info is    F  o  r  k     #  4  ,     G  E  T
              70 111 114 107  32  35  52  44  32  71  69  84
REQUEST SEND:      count = 25,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    2d9   1ed3     0     0     0


[HISTPREV] Command:

found at 41
WAIT ON SEND:      count = 25,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b21    c49    1f7   1ed3     0     0     0


[HISTPREV] Command:

found at 40
Saved info is    F  o  r  k     #  4  ,     W  o  r
              70 111 114 107  32  35  52  44  32  87 111 114
SEND:              count = 25,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    1f7   1ed3     0     0     0


[HISTPREV] Command:

found at 39
```

```
Saved info is    F  o  r  k     #  4  ,     W  o  r
           70 111 114 107  32  35  52  44  32  87 111 114
REQUEST SEND:      count = 24,
 loc_link  rem_link  rem_mach  rem_proc
   21        28        1         6
PC walkback =    b14    c49    1f7   1ed3     0     0     0


[HISTPREV] Command:

found at 38
WAIT ON RECEIVE:   count = 9,
 loc_link  rem_link  rem_mach  rem_proc
   13        12        3         3
PC walkback =    161   1ed3    134   1ed3     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:
                                        Message received before error detection was a "get"
found at 37                             request from Philo 4
                                        Note this is at location #37 in communication history

Saved info is    g
           103  0  0  0  3  0  0  0  3  0  1  0
RECEIVE:           count = 9,
 loc_link  rem_link  rem_mach  rem_proc
   13        12        3         3
PC walkback =    154   1ed3    134   1ed3     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command: o                                   Go to matching send
Looking for system process 3
Process #4 is:  Philo4, System id is 3

found at 74                          Matching send at 74 in Philo4's communication history
Saved info is    g  n
           103 110  1  0  0  0  0  0  2  0  2  0
SEND:              count = 9,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    36f    724   204f     0     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5
```

```
[OTHEREND] Command: n

found at 75
WAIT ON SEND:      count = 9,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    37c    724   204f     0     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5


[HISTNEXT] Command:

found at 76
Saved info is     n
           110  1  0  0  0  0  0  2  0  1  0  1
REQUEST RECEIVE:   count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    195    333    724   204f     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5


[HISTNEXT] Command:
                                          Response to request is a "yes" from Fork4
found at 77                          BUT Fork4 signalled an error instead of responding??
Saved info is     y
           121  1  0  0  0  0  0  2  0  1  0  1
RECEIVE:           count = 8,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    195    333    724   204f     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5


[HISTNEXT] Command: o                     return to Fork4 to see where
Looking for system process 5              response was issued
Process #9 is:  Fork4, System id is 5

found at 22                          Response recorded at 22 in communication history
Saved info is     y                  This is before request was received
           121  0  0  0  0  0  0  0  2  0  1  0
SEND:              count = 8,
 loc_link  rem_link  rem_mach  rem_proc
   13        12        3         3
PC walkback =    3e4   1ed3    1f7   1ed3     0     0     0
```

```
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3
```

The next 8 pages detail more evidence that the pattern of communication is not synchronized

```
[OTHEREND] Command:  p

found at 21
Saved info is     y
              121  0  0  0  0  0  0  0  2  0  1  0
REQUEST SEND:       count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    3e4   1ed3    1f7   1ed3     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:

found at 20
WAIT ON SEND:       count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    368   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:

found at 19
WAIT ON SEND:       count = 21,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b21    c49    1f7   1ed3     0     0     0


[HISTPREV] Command:

found at 18
Saved info is     F  o  r  k     #  4  ,     W  o  r
              70 111 114 107  32  35  52  44  32  87 111 114
SEND:               count = 21,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    1f7   1ed3     0     0     0
```

```
[HISTPREV] Command:

found at 17
Saved info is    F  o  r  k     #  4  ,     W  o  r
            70 111 114 107  32  35  52  44  32  87 111 114
REQUEST SEND:      count = 20,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    1f7   1ed3     0     0     0

[HISTPREV] Command:

found at 16
WAIT ON RECEIVE:   count = 8,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    161   1ed3   134   1ed3     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:

found at 15
Saved info is    g
            103  0  0  0  3  0  0  0  3  0  1  0
RECEIVE:           count = 8,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    154   1ed3   1ed3     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command: o
Looking for system process 3
Process #4 is:  Philo4, System id is 3

found at 68
Saved info is    g  n
            103 110  1  0  0  0  0  0  2  0  2  0
SEND:              count = 8,
 loc_link  rem_link  rem_mach  rem_proc
   12        13         4         5
PC walkback =    36f    724   204f     0     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5
```

```
[OTHEREND] Command: n

found at 69
WAIT ON SEND:      count = 8,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    37c    724   204f     0     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5


[HISTNEXT] Command:

found at 70
Saved info is     n
            110   1   0   0   0   0   0   2   0   1   0   1
REQUEST RECEIVE:   count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    195    333    724   204f     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5


[HISTNEXT] Command:

found at 71
Saved info is     n
            110   1   0   0   0   0   0   2   0   1   0   1
RECEIVE:           count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    195    333    724   204f     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5


[HISTNEXT] Command: o
Looking for system process 5
Process #9 is:  Fork4, System id is 5

found at 0
Saved info is     n
            110   0   0   0   0   0   0   0   2   0   1   0
SEND:              count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   13        12        3         3
PC walkback =    3e4   1ed3     0     0     0     0     0
```

```
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[OTHEREND] Command:  p

found at 100
Saved info is     n
            110  0  0  0  0  0  0  0  2  0  1  0
REQUEST SEND:      count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    3e4   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:

found at 99
WAIT ON SEND:      count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    368   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:

found at 98
WAIT ON SEND:      count = 17,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b21    c49    1f7   1ed3     0     0     0


[HISTPREV] Command:

found at 97
Saved info is    F  o  r  k     #  4  ,     W  o  r
            70 111 114 107  32  35  52  44  32  87 111 114
SEND:              count = 17,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    1f7   1ed3     0     0     0
```

```
[HISTPREV] Command:

found at 96
Saved info is    F  o  r  k     #  4  ,     W  o  r
            70 111 114 107  32  35  52  44  32  87 111 114
REQUEST SEND:      count = 16,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    1f7   1ed3     0     0     0


[HISTPREV] Command:

found at 95
WAIT ON RECEIVE:   count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    161   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:

found at 94
Saved info is    g
            103  0  0  0  2  0  0  0  2  0  1  0
RECEIVE:           count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    154   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command: o
Looking for system process 3
Process #4 is:  Philo4, System id is 3

found at 62
Saved info is    g  n
            103 110  1  0  0  0  0  0  2  0  1  0
SEND:              count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   12        13         4         5
PC walkback =    28b    724   204f     0     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5
```

[OTHEREND] Command: **n**

found at 63
WAIT ON SEND: count = 7,
loc_link rem_link rem_mach rem_proc
12 13 4 5
PC walkback = 298 724 204f 0 0 0 0
The process on the other end of this link is:
Process #9 is: Fork4, System id is 5

[HISTNEXT] Command:

found at 64
Saved info is n
110 1 0 0 0 0 0 2 0 1 0 1
REQUEST RECEIVE: count = 5,
loc_link rem_link rem_mach rem_proc
12 13 4 5
PC walkback = 195 333 724 204f 0 0 0
The process on the other end of this link is:
Process #9 is: Fork4, System id is 5

[HISTNEXT] Command:

found at 65
Saved info is n
110 1 0 0 0 0 0 2 0 1 0 1
RECEIVE: count = 6,
loc_link rem_link rem_mach rem_proc
12 13 4 5
PC walkback = 195 333 724 204f 0 0 0
The process on the other end of this link is:
Process #9 is: Fork4, System id is 5

[HISTNEXT] Command: **o**
Looking for system process 5
Process #9 is: Fork4, System id is 5

found at 90
Saved info is n
110 0 0 0 0 0 0 0 2 0 1 0
SEND: count = 6,
loc_link rem_link rem_mach rem_proc
13 12 3 3
PC walkback = 3e4 1ed3 0 0 0 0 0

```
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[OTHEREND] Command:  p

found at 89
Saved info is     n
            110  0  0  0  0  0  0  0  2  0  1  0
REQUEST SEND:       count = 5,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    3e4   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:

found at 88
WAIT ON SEND:       count = 5,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    368   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command:

found at 87
WAIT ON SEND:       count = 15,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b21    c49    1f7   1ed3     0     0     0


[HISTPREV] Command:

found at 86
Saved info is     F  o  r  k     #  4  ,     W  o  r
            70 111 114 107  32  35  52  44  32  87 111 114
SEND:               count = 15,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    1f7   1ed3     0     0     0
```

```
[HISTPREV] Command:

found at 85
Saved info is    F  o  r  k     #  4  ,     W  o  r
            70 111 114 107  32  35  52  44  32  87 111 114
REQUEST SEND:      count = 14,
 loc_link  rem_link  rem_mach  rem_proc
   21        28         1         6
PC walkback =    b14    c49    1f7   1ed3     0     0     0

[HISTPREV] Command:

found at 84
WAIT ON RECEIVE:   count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    161   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3

[HISTPREV] Command:

found at 83
Saved info is    g
            103  0  0  0  2  0  0  0  2  0  1  0
RECEIVE:           count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   13        12         3         3
PC walkback =    154   1ed3     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  Philo4, System id is 3


[HISTPREV] Command: o
Looking for system process 3
Process #4 is:  Philo4, System id is 3

found at 53
Saved info is    g  n
            103 110  1  0  0  0  0  0  2  0  2  0
SEND:              count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   12        13         4         5
PC walkback =    36f    724   204f     0     0     0     0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5
```

```
[HISTPREV] Command: s
[4] Which Process? (Enter logical process id):
[0] Local link number (0 for any): 12
[0] remote link number (0 for any):
[0] remote machine number (0 for any):
[0] remote process number (0 for any):
[0] link count (0 for any): 7
[any] Message Type: ([a]ny,[s]nd,[r]ec,[d]es,[R]wait,[S]wait,reqre[c],reqse[n]): s
[backward] Direction to search ([f]orward, [b]ackward, [s]ame, [r]everse):

found at 62
Saved info is    g   n
          103 110  1  0  0  0  0  0  2  0  1  0
SEND:          count = 7,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =   28b   724   204f    0    0    0    0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5
```

the send request below is requesting the fork from Fork4 but this philosopher never read the previous message from Fork4, since the fork has been granted this next request is an error

```
[STATUS] Command: p

found at 61
Saved info is    g   n
          103 110  1  0  0  0  0  0  2  0  1  0
REQUEST SEND:     count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =   28b   724   204f    0    0    0    0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5


[HISTPREV] Command:

found at 60
WAIT ON SEND:     count = 5,
 loc_link  rem_link  rem_mach  rem_proc
   19        23        1         6
PC walkback =   c9d   dc5   701   204f    0    0    0
```

```
[HISTPREV] Command:

found at 59
Saved info is    P  4     g  e  t  t  i  n  g     f
            80  52  32 103 101 116 116 105 110 103  32 102
SEND:            count = 5,
 loc_link  rem_link  rem_mach  rem_proc
   19        23        1        6
PC walkback =    c90    dc5    701   204f     0     0     0


[HISTPREV] Command:

found at 58
Saved info is    P  4     g  e  t  t  i  n  g     f
            80  52  32 103 101 116 116 105 110 103  32 102
REQUEST SEND:    count = 4,
 loc_link  rem_link  rem_mach  rem_proc
   19        23        1        6
PC walkback =    c90    dc5    701   204f     0     0     0


[HISTPREV] Command:

found at 57
WAIT ON SEND:    count = 4,
 loc_link  rem_link  rem_mach  rem_proc
   19        23        1        6
PC walkback =    c9d    dc5    12b    749   204f     0     0


[HISTPREV] Command:

found at 56
Saved info is    P  4     t  h  i  n  k  i  n  g

            80  52  32 116 104 105 110 107 105 110 103  10
SEND:            count = 4,
 loc_link  rem_link  rem_mach  rem_proc
   19        23        1        6
PC walkback =    c90    dc5    12b    749   204f     0     0


[HISTPREV] Command:

found at 55
Saved info is    P  4     t  h  i  n  k  i  n  g
```

```
            80  52  32 116 104 105 110 107 105 110 103  10
REQUEST SEND:      count = 3,
 loc_link  rem_link  rem_mach  rem_proc
   19        23        1         6
PC walkback =    c90    dc5    12b    749   204f     0      0


[HISTPREV] Command:

found at 54
WAIT ON SEND:      count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    37c    724   204f     0      0      0      0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5
```

the send below (#53) requests a fork but the philosopher never makes a receive request to get the reply

```
[HISTPREV] Command:

found at 53
Saved info is    g   n
            103 110   1   0   0   0   0   0   2   0   2   0
SEND:              count = 6,
 loc_link  rem_link  rem_mach  rem_proc
   12        13        4         5
PC walkback =    36f    724   204f     0      0      0      0
The process on the other end of this link is:
Process #9 is:  Fork4, System id is 5

[HISTPREV] Command:  q
charlotte_3 = >
```

With the above information we can see that Philo4 made two consecutive requests to Fork4 without looking at the intervening response until later. Knowing where in the philosopher's code the sends come from helps us limit our search for the cause of the problem. The walkback shows the send request was made when the PC = 36f.

```
        procedure GetFork (forklink : LinkId; var timeout: integer) : boolean;
        begin
024e    msg.Action := GET;
0256    retcode    := Wait(Send(forklink, adr(msg), size(msg), NOLINK), Sent, result);
0297    if retcode < SUCCESS then
029f        sprintf(Printstring, "Philo%d: error on GET, retcode=%d ",
```

```
                                                    rank, retcode);
02be        Print(Printstring);
02d9        SignalParent(2);
02ea        dummy := Suspend(-1);
            end;
0306  while (timeout > 0) and (not GetAns(forklink)) do
0332        msg.Action := GET;
033a        retcode  := Wait(Send(forklink, adr(msg), size(msg), NOLINK),
                                                    Sent, result);
037b        if retcode < SUCCESS then
0383            sprintf(Printstring, "Philo%d: error on GET #2, retcode=%d ",
                                                    rank, retcode);
03a2            Print(Printstring);
03bd            SignalParent(3);
03ce            dummy := Suspend(-1);
                end;
03ea        timeout := timeout - 1;
            end; (* while *)
03f9  if timeout > 0 then
0401        GetFork := true
0407  else GetFork := false          end;
end GetFork;
```

7.2 Appendix B

The following is a transcript of the session for the process which did two receives into the same buffer.

```
Charlotte(vax), Version 2.5.1 (new nugget), Fri Mar 29 11:50:54 CST 1985

%1: Please give your Configuration File name: /tmp/ajg

charlotte_1 => cd rcv2

charlotte_2 => connect A                                    the program is started here

                                                    The error is detected and reported here
3: Z> num = 2, msg[1] = 1002, msg[2] = 1002, msg[3] = 1002, msg[4] = 1002
TAP: going to work

Distress signal from process #1

[HALT] Command: s
[-1] Which Process? (Enter logical process id): 1
[0] Local link number (0 for any):
[0] remote link number (0 for any):
[0] remote machine number (0 for any):
[0] remote process number (0 for any):
[0] link count (0 for any):
[any] Message Type: ([a]ny,[s]nd,[r]ec,[d]es,[R]wait,[S]wait,reqre[c],reqse[n]): r
[backward] Direction to search ([f]orward, [b]ackward, [s]ame, [r]everse):
                                    The following is a look at the messages received
                                    by process 1 (Named AAA)

found at 28
Saved info is
          -22  3  0  0 -22  3  0  0 -22  3  0  0
RECEIVE:          count = 2,                                          PC = 244
 loc_link  rem_link  rem_mach  rem_proc
    8        33        2         8
PC walkback =   7c1   8e9   244   1b73     0     0     0
The process on the other end of this link is:
Process #3 is: CCC, System id is 8


[REPEAT] Command:
found at 23
Saved info is
            2  0  0  0  2  0  0  0  2  0  0  0
RECEIVE:          count = 2,                                          PC = 127
 loc_link  rem_link  rem_mach  rem_proc
```

```
   1       1       4       3
PC walkback =    127   1b73     0     0     0     0     0
The process on the other end of this link is:
Process #2 is:  BBB, System id is 3


[REPEAT] Command:
found at 20
Saved info is
            -23   3   0   0 -23   3   0   0 -23   3   0   0
RECEIVE:            count = 1,                                  PC = 2ed
 loc_link  rem_link  rem_mach  rem_proc
    8       33        2        8
PC walkback =    2ed   1b73     0     0     0     0     0
The process on the other end of this link is:
Process #3 is:  CCC, System id is 8


[REPEAT] Command:
found at 15
Saved info is
              1   0   0   0   1   0   0   0   1   0   0   0
RECEIVE:            count = 1,                                  PC = 167
 loc_link  rem_link  rem_mach  rem_proc
    1       1        4        3
PC walkback =    167   1b73     0     0     0     0     0
The process on the other end of this link is:
Process #2 is:  BBB, System id is 3


[REPEAT] Command:
found at 6
Saved info is
              2   0   0   0   9   0   0   0   0   0   0   0
RECEIVE:            count = 3,
 loc_link  rem_link  rem_mach  rem_proc
    4       30        2        5
PC walkback =    c94     0     0     0     0     0     0


[REPEAT] Command:  W


proc  valid first wrap dir  histadr  histpntr  mach  sid  name
 1     T     F     F    B   38556     33        3     3   AAA
 2     T     T     F    F   38048     21        4     3   BBB
 3     T     T     F    F   38076     21        2     8   CCC
```

[REPEAT] Command: q

When the messages were received the PC was at 244, 127, 2ed and 167. As can be seen from the section of code below with corresponding addresses, all but the last receive occurred with a wait. The last receive occurred while the process was using buffer and led to the error.

charlotte_4=>

```
==========================================================

PC    Statement              Process AAA

      module main;
          .
          .
      begin
00d5      Setup;
00e0      i    := 1;
00e7      while i < 1000 do
0102      err  := Receive(b,adr(msg),size(msg));
0126      err  := Receive(c,adr(msg),size(msg));
014a      err  := Wait(ALL_LINKS,Received,result);
0166      num := msg[1];
0171      if (msg[2] < > num)  or (msg[3] < > num) or (msg[4] < > num) then
0198          sprintf(printmsg,
                  "Y> num = %d, msg[1] = %d, msg[2] = %d, msg[3] = %d,
                  msg[4] = %d0, num, msg[1], msg[2], msg[3], msg[4]);
01c9          Print(printmsg);
01e0          SignalParent(1);
01ed          dummy    := Suspend(-1);
          end;
0201      sprintf(printmsg,"first num = %d0, num);
021a      num := msg[1];
0225      Print(printmsg);
023c      if (msg[2] < > num)  or (msg[3] < > num) or (msg[4] < > num) then
0263          sprintf(printmsg,
                  "Z> num = %d, msg[1] = %d, msg[2] = %d, msg[3] = %d,
                  msg[4] = %d0, num, msg[1], msg[2], msg[3], msg[4]);
0294          Print(printmsg);
02ab          SignalParent(1);
02b8          dummy    := Suspend(-1);
          end;
02cc      err  := Wait(ALL_LINKS,Received,result);
02ec      inc(i);
          end; (* while *)
      end main.
```

7.3 Appendix C

The following is a transcript of the debugging session for the program which had a process (global) receiving values from three other process, one value from each per round. Somehow global received values which indicated one process had gotten two rounds ahead of another.

```
Charlotte(vax), Version 2.5.1 (new nugget), Fri Mar 29 11:50:54 CST 1985

%1: Please give your Configuration File name: /tmp/ajg

charlotte_1=> cd round

charlotte_6=> connect A

3: Global: 20 rounds done
3: Global: 20 rounds done
3: Global: 20 rounds done
3: Global: 20 rounds done
3: Global: underflow error,  i = 36, total = -6          Error detected here

TAP: going to work

Distress signal from process #1

[HALT] Command: W                                        Who is everybody?


proc  valid first wrap dir  histadr  histpntr  mach  sid  name
1     T     T     T    F    38700    78        4     6    global
2     T     T     T    F    38816    36        2     14   first
3     T     T     T    F    38816    25        2     15   second
4     T     T     T    F    38816    30        3     6    third

[HALT] Command: s                        error detected by process 1 so look at it
[-1] Which Process? (Enter logical process id): 1
[0] Local link number (0 for any):
[0] remote link number (0 for any):
[0] remote machine number (0 for any):
[0] remote process number (0 for any):
[0] link count (0 for any):
[any] Message Type: ([a]ny,[s]nd,[r]ec,[d]es,[R]wait,[S]wait,reqre[c],reqse[n]): r
[backward] Direction to search ([f]orward, [b]ackward, [s]ame, [r]everse):

                                         Look at values received by process 1
                                                    -6 from first
found at 73
Saved info is
          -6 -1 -1 -1  0  0  0  0  0  0  0  0
```

```
RECEIVE:          count = 93,
 loc_link  rem_link  rem_mach  rem_proc
   13        36        2        14
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #2 is:  first, System id is 14


[REPEAT] Command:

found at 71                                                    -6 from first
Saved info is
            -6  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 92,
 loc_link  rem_link  rem_mach  rem_proc
   13        36        2        14
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #2 is:  first, System id is 14


[REPEAT] Command:

found at 69                                                    -4 from third
Saved info is
            -4  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 92,
 loc_link  rem_link  rem_mach  rem_proc
   18        13        3        6
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6


[REPEAT] Command:

found at 67                                                    -6 from first
Saved info is
            -6  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   13        36        2        14
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #2 is:  first, System id is 14


[REPEAT] Command:
```

-4 from third

```
found at 65
Saved info is
            -4  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   18        13        3         6
PC walkback =     d6   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6
```

[REPEAT] Command:

10 from second

```
found at 63
Saved info is
           10   0   0   0   0   0   0   0   0   0   0   0
RECEIVE:          count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   17        70        2         15
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #3 is:  second, System id is 15
```

[REPEAT] Command:

-4 from third

```
found at 61
Saved info is
            -4  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 90,
 loc_link  rem_link  rem_mach  rem_proc
   18        13        3         6
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6
```

[REPEAT] Command:

10 from second

```
found at 59
Saved info is

           10   0   0   0   0   0   0   0   0   0   0   0
RECEIVE:          count = 90,
 loc_link  rem_link  rem_mach  rem_proc
   17        70        2         15
PC walkback =     f2   1a3f   1b8   1a3f     0     0     0
The process on the other end of this link is:
```

```
Process #3 is:  second, System id is 15


[REPEAT] Command:

found at 57                                                    -6 from first
Saved info is
            -6  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 90,
 loc_link  rem_link  rem_mach  rem_proc
    13        36         2        14
PC walkback =     f2   1a3f    1b8   1a3f     0     0     0
The process on the other end of this link is:
Process #2 is:  first, System id is 14


[REPEAT] Command:

found at 55                                                    10 from second
Saved info is

            10   0   0   0   0   0   0   0   0   0   0   0
RECEIVE:          count = 89,
 loc_link  rem_link  rem_mach  rem_proc
    17        70         2        15
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #3 is:  second, System id is 15


[REPEAT] Command:

found at 53                                                    -6 from first
Saved info is
            -6  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 89,
 loc_link  rem_link  rem_mach  rem_proc
    13        36         2        14
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #2 is:  first, System id is 14


[REPEAT] Command:

found at 51                                                    -4 from third
Saved info is
            -4  -1  -1  -1   0   0   0   0   0   0   0   0
```

```
RECEIVE:          count = 89,
 loc_link  rem_link  rem_mach  rem_proc
   18        13        3         6
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6
```

[REPEAT] Command:

-6 from first

```
found at 49
Saved info is
            -6  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 88,
 loc_link  rem_link  rem_mach  rem_proc
   13        36        2         14
PC walkback =     d6   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #2 is:  first, System id is 14
```

[REPEAT] Command:

-4 from third

```
found at 47
Saved info is
            -4  -1  -1  -1   0   0   0   0   0   0   0   0
RECEIVE:          count = 88,
 loc_link  rem_link  rem_mach  rem_proc
   18        13        3         6
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6
```

[REPEAT] Command:

10 from second

```
found at 45
Saved info is
             10   0   0   0   0   0   0   0   0   0   0   0
RECEIVE:          count = 88,
 loc_link  rem_link  rem_mach  rem_proc
   17        70        2         15
PC walkback =     f2   1a3f     0     0     0     0     0
The process on the other end of this link is:
Process #3 is:  second, System id is 15
```

[REPEAT] Command:

found at 43 **-4 from third**
Saved info is
-4 -1 -1 -1 0 0 0 0 0 0 0 0
RECEIVE: count = 87,
loc_link rem_link rem_mach rem_proc
18 13 3 6
PC walkback = f2 1a3f 0 0 0 0 0
The process on the other end of this link is:
Process #4 is: third, System id is 6

[REPEAT] Command:

found at 41 **10 from second**
Saved info is
10 0 0 0 0 0 0 0 0 0 0 0
RECEIVE: count = 87,
loc_link rem_link rem_mach rem_proc
17 70 2 15
PC walkback = f2 1a3f 0 0 0 0 0
The process on the other end of this link is:
Process #3 is: second, System id is 15

[REPEAT] Command:

found at 39 **-6 from first**
Saved info is
-6 -1 -1 -1 0 0 0 0 0 0 0 0
RECEIVE: count = 87,
loc_link rem_link rem_mach rem_proc
13 36 2 14
PC walkback = f2 1a3f 0 0 0 0 0
The process on the other end of this link is:
Process #2 is: first, System id is 14

[REPEAT] Command:

found at 37 **10 from second**
Saved info is

10 0 0 0 0 0 0 0 0 0 0 0
RECEIVE: count = 86,
loc_link rem_link rem_mach rem_proc
17 70 2 15
PC walkback = d6 1a3f 1b8 1a3f 0 0 0
The process on the other end of this link is:

Process #3 is: second, System id is 15

[REPEAT] Command:

-6 from first

found at 35
Saved info is
-6 -1 -1 -1 0 0 0 0 0 0 0 0
RECEIVE: count = 86,
loc_link rem_link rem_mach rem_proc
13 36 2 14
PC walkback = f2 1a3f 1b8 1a3f 0 0 0
The process on the other end of this link is:
Process #2 is: first, System id is 14

[REPEAT] Command:

-4 from third

found at 33
Saved info is
-4 -1 -1 -1 0 0 0 0 0 0 0 0
RECEIVE: count = 86,
loc_link rem_link rem_mach rem_proc
18 13 3 6
PC walkback = f2 1a3f 0 0 0 0 0
The process on the other end of this link is:
Process #4 is: third, System id is 6

We can see the pattern of values received by global is irregular

Process 2 (first) is the process that got ahead
Now let us look at first

[REPEAT] Command: s
[1] Which Process? (Enter logical process id): **2**
[0] Local link number (0 for any):
[0] remote link number (0 for any):
[0] remote machine number (0 for any):
[0] remote process number (0 for any):
[0] link count (0 for any):
[any] Message Type: ([a]ny,[s]nd,[r]ec,[d]es,[R]wait,[S]wait,reqre[c],reqse[n]): **r**
[backward] Direction to search ([f]orward, [b]ackward, [s]ame, [r]everse):

Received synchronization message from third

found at 27
Saved info is
1 0 0 0 0 0 0 0 0 0 0 0
RECEIVE: count = 92,

```
 loc_link  rem_link  rem_mach  rem_proc
   40        17         3         6
PC walkback =    279   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6


[STATUS] Command:  .

found at 21                                  Received synchronization message from third
Saved info is
                1  0  0  0  0  0  0  0  0  0  0  0  0
RECEIVE:          count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   40        17         3         6
PC walkback =    2c5   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6


[REPEAT] Command:  .

found at 12                                  Received synchronization message from third
Saved info is
                1  0  0  0  0  0  0  0  0  0  0  0  0
RECEIVE:          count = 90,
 loc_link  rem_link  rem_mach  rem_proc
   40        17         3         6
PC walkback =    20a   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6


                                             Go forward from here and see what the pattern
                                             of communication looks like
[REPEAT] Command:  n

found at 13
WAIT ON RECEIVE:   count = 90,
 loc_link  rem_link  rem_mach  rem_proc
   40        17         3         6
PC walkback =    20a   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6


[HISTNEXT] Command:
```

```
found at 14
Saved info is
            -6 -1 -1 -1  0  0  0  0  0  0  0  0
REQUEST SEND:      count = 90,
 loc_link  rem_link  rem_mach  rem_proc
   36        13         4         6
PC walkback =    248   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #1 is:  global, System id is 6


[HISTNEXT] Command:

                                                          Send -6 to global
found at 15
Saved info is
            -6 -1 -1 -1  0  0  0  0  0  0  0  0
SEND:              count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   36        13         4         6
PC walkback =    248   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #1 is:  global, System id is 6


[HISTNEXT] Command:

found at 16
WAIT ON SEND:      count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   36        13         4         6
PC walkback =    255   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #1 is:  global, System id is 6


[HISTNEXT] Command:

found at 17
REQUEST RECEIVE:   count = 90,
 loc_link  rem_link  rem_mach  rem_proc
   40        17         3         6
PC walkback =    279   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6


[HISTNEXT] Command:
```

```
found at 18
WAIT ON SEND:      count = 90,
 loc_link  rem_link  rem_mach  rem_proc
   74        77        2         15
PC walkback =    29a   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #3 is:  second, System id is 15


[HISTNEXT] Command:

found at 19                                          Send synchronization message to second
Saved info is
              1   0   0   0   0   0   0   0   0   0   0   0
REQUEST SEND:      count = 90,
 loc_link  rem_link  rem_mach  rem_proc
   74        77        2         15
PC walkback =    2c5   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #3 is:  second, System id is 15


[HISTNEXT] Command:

found at 20
Saved info is
              1   0   0   0   0   0   0   0   0   0   0   0
SEND:              count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   74        77        2         15
PC walkback =    2c5   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #3 is:  second, System id is 15


[HISTNEXT] Command:

found at 21                                          Received synchronization message from third
Saved info is
              1   0   0   0   0   0   0   0   0   0   0   0
RECEIVE:           count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   40        17        3         6
PC walkback =    2c5   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6
```

```
[HISTNEXT] Command:

found at 22
WAIT ON RECEIVE:   count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   40        17        3         6
PC walkback =    20a   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #4 is:  third, System id is 6


[HISTNEXT] Command:

found at 23                                                    Send -6 to global
Saved info is
            -6  -1  -1  -1  0  0  0  0  0  0  0  0
REQUEST SEND:      count = 91,
 loc_link  rem_link  rem_mach  rem_proc
   36        13        4         6
PC walkback =    248   1b43     0     0     0     0     0
The process on the other end of this link is:
Process #1 is:  global, System id is 6


[HISTNEXT] Command: q

charlotte_7=>
```

The pattern of communication between first and the other processes shows that between sending values to global first tries to synchronize with the other two processes. First tries to receive a message from third and waits for it to complete. First also tries to send a message to second but does not wait for the completion until the next round. By then it is too late; first is too far ahead of second.

Chapter 8
REFERENCES

[Aho79] **Aho, Alfred V. and Jeffrey D. Ullman**: *Principles of Compiler Design*, Addison-Wesley, 1979.

[Alme80] **Almes, G. T., M. J. Fischer, H. Golde, E. D. Lazowska, and J. D. Noe**: "Eden project proposal." TR 80-10-01, University of Washington, October, 1980.

[Andr82] **Andrews, Gregory R.**: "The Distributed Programming Language SR - Mechanisms, Design and Implementation," *Software-Practice and Experience*, 12 1982, pp. 719-753.

[Arts84] **Artsy, S., H-Y Chang, and R. Finkel**: "Charlotte: Design and Implementation of a Distributed Kernel." Technical Report 554, University of Wisconsin-Madison Computer Sciences, September 1984.

[Baia83] **Baiardi, F., N. DeFrancesco, and E. Matteoli**: "Development of a Debugger For a Concurrent Language," *ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High Level Debugging*, March, 1983, pp. 6-22.

[Balz69] **Balzer, R. M.**: "EXDAMS - EXtendable Debugging and Monitoring Systems," *Proceddings of AFIPS FJCC*, 34 1969, pp. 567-580.

[Bask77] **Baskett, F., J. H. Howard, and J. T. Montague**: "Task communication in Demos," *Proc. 6th Symposium on Operating Systems Principles*, November 1977, pp. 23-31.

[Bate81] **Bates, Peter and Jack C. Wileden**: "Event Definition Language: An Aid to Monitoring and Debugging Complex Software Systems." COINS Technical Report 81-17, University of Massachusetts, 1981.

[Card83] **Cardell, James R.**: "Multilingual Debugging with the SWAT High-Level Debugger," *ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High Level Debugging*, March, 1983, pp. 79-99.

[Cher83] **Cheriton, David R. and Willy Zwaenepoel**: "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating System Principles*, October, 1983, pp. 129-140.

[Cook83] **Cook, R., R. Finkel, D. DeWitt, L. Landweber, and T. Virgilio**: "The crystal nugget: Part I of the first report on the crystal project." Technical Report 499,

Computer Sciences Department, University of Wisconsin, April 1983.

[DeWi84] **DeWitt, D., R. Finkel, and M. Solomon**: "The Crystal multicomputer: Design and implementation experience." Technical Report 553, University of Wisconsin-Madison Computer Sciences, September 1984.

[Ditz78] **Ditzel, David R.**: "Interactive Debugging Tools for a Block Structured Programming Language." MCS72-03642-CL7802, Cyclone Computer Laboratory, Iowa State University, 1978.

[Evan66] **Evans, T. G. and D.L. Darley**: "On-Line Debugging Techniques: A Survey," *Proceddings of AFIPS*, 29 Fall, 1966, pp. 37-50.

[Fink81] **Finkel, R. A. and M. H. Solomon**: "The Arachne Distributed Operating System." Technical Report 439, University of Wisconsin--Madison Computer Sciences, July, 1981.

[Fink83a] **Finkel, R. A., M. H. Solomon, and et. al.**: "Charlotte: Part IV of the First Report on the Crystal Project." Technical Report 502, University of Wisconsin--Madison Computer Sciences, 1983.

[Fink83b] **Finkel, R. A., Robert Cook, David DeWitt, Nancy Hall, and Lawrence Landweber**: "Wisconsin Modula." Technical Report 501, University of Wisconsin--Madison Computer Sciences, April, 1983.

[Garc84] **Garcia-Molina, H., F. Germano, and W. H. Kohler**: "Debugging a Distributed Computing System," *IEEE Transactions on Software Engineering*, SE-10 , 2 March 1984, pp. 210-219.

[Gilm57] **Gilmore, J. T.**: "TX-O Direct Input Utility System." Lincoln Laboratory Memo 6M-5097, MIT, April, 1957.

[Gros83] **Gross, Thomas and Willy Zwaenpoel**: "System Support for Multi-Process Debugging," *ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High Level Debugging*, March, 1983, pp. 192-196.

[Hodg80] **Hodgson, L. I. and M. Porter**: "BIDOPS - A Bi-Directional Programming System," *Australian Computer Science Comm.*, 2 , 3 June, 1980, pp. 349-355.

[Kish83] **Kishimoto, Zen**: "An Experimental Debugger in a Limited Programming Environment," *ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High Level*

Debugging, March, 1983, pp. 242-243.

[Kuls69] **Kulsrud, H. E.**: "HELPER: An Interactive Extensible Debugging System," *Second Symposium on Operating Systems Principles*, 1969, pp. 105-111.

[Lamp78] **Lamport, Leslie**: "Time, Clocks, and the ordering of Events in a Distributed System," *CACM*, 21 , 7 July, 1978, pp. 558-564.

[LeBl83] **LeBlanc, Richard J.**: "Position Statement: Interactive Debugging of Distributed Programs," *ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High Level Debugging*, March, 1983, pp. 250-253.

[Lisk83] **Liskov, Barbara and Robert Scheifler**: *TOPLAS*, 5 , 3 July, 1983, pp. 381-404.

[Mcda77] **Mcdaniel, Gene**: *METRIC: A Kernel Instrumentation System for Distributed Environments*, Proceedings of Sixth ACM Symposium on Operating System Principles, ACM, Nov. 1977.

[Mill85] **Miller, Barton Paul**: "Performance Characterization of Distributed Programs." UCB/CSD 85/197, Computer Science Division (EECS), University of California, January, 1985.

[Owen81] **Owen, David and Allan Ramsay**: "An Environment for Distributed Computing," *The 2nd International Conference on Distributed Computing Systems*, 1981, pp. 173-179.

[Pars79] **Parsley, B. L., H. G. Lehtman, and S. Kahn**: "On-Line Programmer's Management System, Addendum I & II, User's Guide to JOVIAL Debugger." Rome Air Defense Report RADC-TR-79-205, August, 1979.

[Phil82] **Philips, Doug**: "Black-Flag." Draft, CMU Department of Computer Science, June, 1982.

[Poul78] **Poulsen, S. E.**: "A High Level Symbolic Debugger for Pascal," *Proceedings of the Digital Equipment Computer Users Society*, 4 , 4 Spring, 1978, pp. 953-956.

[Reis75] **Reiser, John F.**: "BAIL - A debugger for SAIL." STAN-CS-75-523, Stanford Computer Science Department, October, 1975.

[Rose85] **Rosenburg, Bryan S.**: *Dissertation in Progress*, University of Wisconsin - Depart-

ment of Computer Sciences, December, 1985.

[Schi81] **Schiffenbauer, Robert D.**: "Interactive Debugging in a Distributed Computational Environment." MIT/LCS/TR-264, MIT Computer Science Department, September 1981.

[Scot84] **Scott, M. L. and R. A. Finkel**: "LYNX: A Dynamic Distributed Programming Language," *1984 International Conference on Parallel Processing*, August, 1984,

[Smit83] **Smith, Edward T.**: "Debugging Techniques for Communicating, Loosely-Coupled Processes," *ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High Level Debugging*, March, 1983,

[Stro83] **Strom, Robert E. and Shaula Yemini**: *NIL: An Integrated Language and System for Distributed Programming*, Proceedings of SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, ACM, June, 1983.

[Swin74] **Swinehart, D. C.**: "COPILOT: A multiple Process Approach to Interactive Programming Systems." STAN-CS-74-412, Stanford University, August, 1974.

[Unit83] **United States Department of Defense,**: "Reference Manual for the Ada Programming Language." ANSI/MIL-STD-1815A-1983, February 1983.

[Webe83] **Weber, Janice Cynthia**: "Interactive Debugging on Concurrent Programs," *ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High Level Debugging*, March, 1983, pp. 357-358.