DPD: A Distributed Program Debugger for the REM Environment

by

Robert S. Side

B.Sc., University of Victoria, Victoria, B.C., Canada, 1986

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the Department
of
Computer Science

ACCEPTED
FACULTY OF GRADUATE STUDIES
[redacted]
DEAN
DATE 1990-08-31

We accept this thesis as conforming
to the required standard

[redacted]

Dr. Gholamali C. Shoja, Supervisor

[redacted]

Dr. Hausi A. Müller, Departmental Member

[redacted]

Dr. Kin F. Li, Outside Member

[redacted]

Dr. Warren D. Little, External Examiner

Supervisor: Dr. Gholamali C. Shoja

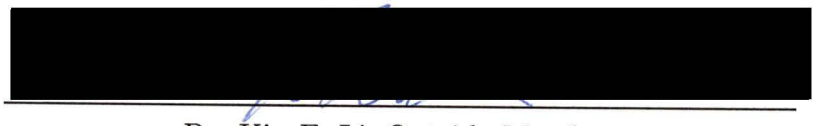
Abstract

Developing a distributed debugger is much more complex than developing a sequential or even a concurrent debugger. This added complexity is mainly due to the non-determinism introduced by the communication delays in distributed systems. This thesis explores the problems that must be addressed when designing a distributed program debugger, and then describes our design and implementation of DPD (Distributed Program Debugger). Problems addressed include non-determinism of events, finding consistent system states, stopping processes, setting breakpoints, recording events, and checkpointing. Important features of DPD include dynamic roll back and replay, as well as a graphical user interface. DPD has been tested successfully in debugging distributed programs within a distributed facility called REM (Remote Execution Manager). The results for actual use of DPD are presented. Scope for future work is also discussed.

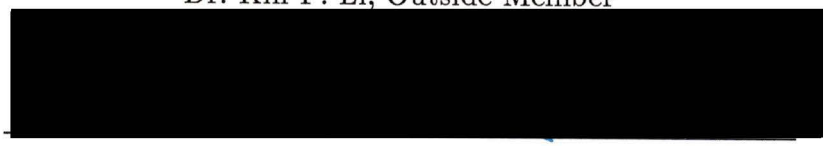
Examiners:

Dr. Gholamali C. Shoja, Supervisor

Dr. Hausi A. Müller, Departmental Member



Dr. Kin F. Li, Outside Member



Dr. Warren D. Little, External Examiner

Contents

List of Figures

List of Tables

Acknowledgements

I wish to thank my supervisor, Dr. Gholamali Shoja, for all his patience, support, and advice throughout this thesis. My committee for their patience waiting for me to finish. My wife, Alice, for her support and for her enduring patience when I was working those late nights. Acquired Intelligence Incorporated and especially Dr. Brian Schaefer for allowing me the freedom and flexibility to work on my thesis. My fellow graduate students, Brian Corrie, Eric Davis, Daniel Liew, Francis Liu, Craig Sinclair, and Jim Uhl, for helping me when I was stuck. The staff of the Department of Computer Science for their support and paper work, and the University of Victoria for its financial support. For proof reading my thesis, I like to thank Rick Wagstaff and Charles (Chuck) Jamieson. Finally, I wish to thank my family for encouraging me to continue my education.

Chapter 1

Introduction

This thesis describes our implementation of DPD, a *Distributed Program Debugger* for the REM environment. Through DPD, we explored the problems associated with distributed debugging. In turn, DPD provided REM[Sho90, SCTT88, SCT88] with the necessary tools to debug its distributed applications. Included in the implemented tools are a graphical user interface and a dynamic roll back and replay facility. These tools are lacking or have not been fully developed in other distributed debuggers.

Debugging distributed programs is much more difficult than debugging sequential programs[GMGK84] or concurrent ones[LM89]. A sequential program consists of a single process with a single instruction thread running on a single processor with a single memory. The deterministic behavior of sequential programs is heavily exploited when debugging. Given the same input data, a sequential program produces the same results thus allowing cyclic debugging[MC89, LMC87]. Debugging a concurrent program in a tightly coupled system is similar to debugging a distributed program. A tightly coupled concurrent program is comprised of multiple processes with multiple instruction threads running on a single processor, or on multiple tightly coupled pro-

cessors with shared memory. Concurrent processes may communicate with each other to transfer information. Absence of significant communication delays in single processor systems or in tightly coupled systems makes it easier to debug tightly coupled concurrent programs. However, a distributed program[Ens78, TR85] is comprised of multiple processes with multiple instruction threads running on multiple and remote processors each with their own separate memories. Distributed systems suffer from unknown and variable communication delays. The possibility of remote execution and communication delays makes it hard to debug distributed programs and makes it hard to implement a distributed program debugger. Furthermore, distributed programs, especially faulty ones, may not be deterministic[LMC87]. This non-determinism may be caused by synchronization errors which are brought on by communication delays and by changing processor loads.

The same reasons that make it difficult to debug a distributed application also make the job of implementing a distributed debugger more complex. A distributed debugger must be capable of controlling multiple processes, working with communication delays, maintaining consistent global system states, setting and detecting distributed breakpoints, and collecting multiple trace streams. Furthermore, a distributed debugger must provide the facilities to display temporal relationships between distributed processes and to analyze the volume of trace information that can be produced by distributed applications.

Sequential debugger technology is not sufficient to debug distributed applications. Sequential debuggers do not provide the tools to control and synchronize multiple processes, maintain consistent global system states, or display the temporal relationships between processes. Debuggers for tightly coupled systems are also not sufficient to debug distributed programs because they are not designed with the possibility of remote execution and the varying communication delays in mind. There

has been recent advances in the development of distributed debuggers, for instance [CW82, CP83, HHK85, Coo87, Els89]. Some of these debuggers lack useful facilities such as a graphical user interface to display temporal relationships and a dynamic roll back and replay to analyze event sequences. Those debuggers that have implemented a replay facility provide either limited dynamic replay[Wit89, FB89] or provide *Instant Replay*[LMC87] which can replay only a static sequence of events[MC89, PL89, Els89]. Some "distributed debuggers" are no more than postmortem analyzers[LR85]. These debuggers only supply the tools to analyze the effects of the bug, but not the tools to find the cause of the bug. Some distributed debuggers provide some facility to set sequential style breakpoints, that is breakpoints that effect only one process. There are others that provide the ability to set distributed breakpoints[HHK85, MC88, HW88].

Developing a distributed application also means developing a debugging strategy. There should be at least two main testing phases. In the first phase, one should test each process separately. This is to find sequential logic and programming errors. In the second phase, one should test the program in its distributed environment. This is to find the communication and synchronization errors in the program. Trying to debug a program for sequential problems and for distributed problems concurrently just makes the task of debugging more difficult.

DPD was developed as a debugging facility for REM. REM, *Remote Execution Manager*, is a tool to build fault tolerant distributed programs. It provides facilities for process management, load sharing, and inter-process communication. The main objectives of REM are: (1) utilize the idle processing power of a local network of workstations, (2) provide a simple user interface, and (3) provide transparent fault tolerance in case of a workstation failure. REM requires no modifications to the UNIX kernel and is implemented entirely at the application layer.

1.1 Approach

Our main goals were to explore distributed debugging techniques and to implement a useful distributed debugging facility for REM. We also wanted to develop DPD so that it would follow the REM paradigm, that is, it had to be extensible, portable, and easy to use.

DPD is concerned with debugging the communication and synchronization errors of distributed programs. To provide this, DPD uses a graphical user interface that displays an abstract representation of the program being debugged. This abstract representation displays only processes, events, and the communication between processes. Because of this, there is no need to debug using the source code of the program. DPD allows the user to inspect events, view the contents of messages, and set breakpoints at events. Virtual time[Lam78] is used to show the temporal relations among processes and events.

In its present form, the DPD prototype is not capable of debugging the system's remote execution manager, i.e., REM. We chose to limit our efforts to debugging application programs that run under REM. However, the ability to debug REM itself can be added to DPD at a later point in time.

We assume that DPD has some effect on the performance of the application programs being debugged. Implementing the tools planned for DPD makes it difficult to write a non-impact debugger. However, with the tools that we provide and the techniques that we describe, the impact of the debugger should be minimal.

We do not want to modify the operating system kernel. If we remove this constraint then we should be able to improve on performance. However, we would loose portability and furthermore the administration of the debugger would be more difficult. Most other debuggers and their corresponding distributed environments are em-

bedded into the operating system kernel[HHK85], or need direct kernel support[FB89].

We wanted to use as many existing tools as possible and not to "re-invent the wheel." Existing tools for sequential process control, checkpointing, and system state maintenance are used. We also use REM's facilities to distribute DPD.

The main tool that we wanted to implement and the tool that other debuggers lack is a dynamic roll back and replay feature. Some have implemented a replay feature that allows the user to continually replay the same static sequence of events. This is usually facilitated by keeping a *history log* of events[CC89, LR85]. A history log implies that a continual log of events be kept even when the application program is being used in the field. Also, the history log approach does not provide the facilities to dynamically analyze the execution behavior of application programs. We want to provide this dynamic roll back and replay facility, so that the user can roll back to any given event and interactively control the replay of the program.

1.2 Overview

In Chapter 2, we present the works that have influenced the design and development of DPD. We review monitors for distributed systems, debuggers for concurrent systems, and debuggers for distributed systems.

In Chapter 3, we describe what makes debugging distributed programs and implementing distributed debuggers more difficult than its sequential and concurrent counterparts. We also describe the problems associated with maintaining global system states, collecting trace information, displaying temporal relationships, providing roll back and replay, and breakpointing in a distributed system.

In Chapter 4, we describe the enhancements and extensions that were required

to REM. We present a brief overview of REM and its terminology. We discuss the enhancements that were required to bring REM to a state capable of handling complex distributed applications such as DPD. All the extensions, except for a specific module to handle the DPD interface, are usable outside the scope of distributed debugging.

In Chapter 5, we give a complete and detailed description of the implementation of DPD. We describe the processes that make up the nucleus of DPD. We describe how application processes record their events and how these events are recognized by DPD. We show how our graphical interface interacts with the user, how processes and events are displayed, and how commands are entered. We describe how DPD controls target application processes. Finally, we describe the implementation of our roll back and replay feature.

In Chapter 6, we present the usability of DPD by using it to debug a well-known distributed application. We then measure the intrusiveness of DPD by analyzing specific features of the debugger.

In Chapter 7, we state our findings and contributions. We conclude the thesis by describing future work and possible enhancements to DPD.

In Appendix A, we provide a brief user's guide for DPD.

Chapter 2

Background

There has been a considerable amount of work both on monitors for distributed systems and on debuggers for concurrent programs. The development of debuggers for distributed systems, however, has only recently begun. Monitors aid in the development of distributed programs, especially system programs like REM, by allowing the programmer to view state changes as the distributed program executes. They also allow the programmer to access a postmortem trace. Monitors do not control distributed programs nor should they influence the run-time behavior of the program. Concurrent program debuggers developed for shared memory environments have laid the foundations for distributed program debuggers. Such systems have some means of synchronization, be it either a single clock or a synchronous bus. This synchronization is not available to distributed systems. Some of the early works on distributed debuggers have met with some success. They all have the ability to control distributed programs and to inspect and change system states; however, each one has its strong and weak points.

2.1 Monitors

Radar[LR85] is a monitor that provides a graphical postmortem view of the execution of a distributed program. Event collection is controlled by a library which is combined at link time with the application programs. The purpose of this library is to save every event to disk. Since recording all events can use a lot of disk space, there exists an option that ignores the contents of messages when events are being saved. They offer no mechanisms to filter events or to reduce the number of events being saved. Their graphical interface is based on the partial order of events. It allows the user to run or step through the replay of execution and to inspect the contents of messages. It also provides an instant replay of events but only within a limited time frame. The authors dismiss the resulting intrusiveness of the library by stating that the intrusiveness does not affect the semantics of a program from reaching its goal. This, however, fails to realize that any influence, albeit small, on a faulty distributed program may cause results of successive runs to be different.

Miller *et al.*[MMS86] describe a methodology for monitoring distributed programs running under Berkeley UNIX. Their main focus is on increasing the transparency of the monitor. To achieve this they have modified the UNIX kernel rather than placing the necessary code within a library or having it generated by a compiler. Their monitor provides a postmortem dump of the execution and supplies the mechanisms to record, filter, and analyze events. Users can add their own filters to aid in the analysis. Finally, the analysis procedure totally orders events. Global partial orders must be deduced by the user.

2.2 Concurrent Program Debuggers

LeBlanc and Mellor-Crummey[LMC87] describe a mechanism, called Instant Replay, for reproducing the execution behavior of parallel programs. Their mechanism requires monitoring a program and keeping a history log of events. They can repeatedly reproduce the same behavior in a parallel program by using this history log and by keeping the program's input the same. Instant Replay allows new trace statements to be added to the program and breakpoints to be set. However, when a process reaches a breakpoint, the best they can do is to keep the program in a consistent state by blocking any other process from continuing past a state which depends on the stopped process. Stopping a parallel program in this manner does allow a global state to be inspected and this global state does contain the current state of the process being subjected to breakpointing. However, this global system state may be so out of date, with respect to the time the breakpoint occurred, that the information contained in this global system state may not be beneficial to detect possible bugs. Unless a system can be stopped instantaneously, viewing the state in which the breakpoint occurred may only be possible if roll back and replay is implemented, and they do not describe such a mechanism.

Baiardi *et al.*[BMV83, BFV86] describe a debugger for a concurrent language. Their debugger is based on comparing the actual program behavior with its expected behavior. To do this, they have a specification model that allows the programmer to formally define what the program's behavior should be. Their behavior specification model allows the programmer to define the order and number of iterations of events, and to state assertions. They do not require a notion of global time to establish the order of events. They do have a mechanism to delay processes so that the order of events can be maintained even when the event order is non-deterministic. When

running the debugger and the program diverts from the specification model, the debugger stops the necessary processes and returns control to the user. The problem with this approach is that the user must correctly specify the model. An incorrect model and a faulty program may not cause the debugger to stop the program in the wanted state.

2.3 Distributed Program Debuggers

Even though Jeff[Car83, Car85], the debugger for the Blit[1] terminal, is a sequential program debugger, it is of interest because it is an early example of a distributed debugger. The target environment has limited resources, namely memory, and to use these resources as efficiently as possible, the debugger was implemented as two processes running on separate processors. A slave process along with the process being debugged reside on the Blit terminal. The master process resides on the host processor. The master and slave processes communicate through an asynchronous communication line. When implementing Jeff, the problem of communication delay was encountered which caused the master process on the host processor to become out of synchronization with the target process on the terminal. They found that it was relatively easy to solve this problem, the slave process simply ignored out-of-date commands from the master process. This solution was possible because the slave process, which had direct control over the target process, was always synchronized with the target process.

Garcia-Molina *et al.*[GMGK84] do not describe an implementation but describe the issues involved in debugging a distributed computing system. They argue that when debugging distributed programs one should make generous use of traces, use a

[1]The Blit is a bitmap terminal developed at Bell Laboratories.

two-phase debugging process, and proceed in a bottom-up fashion, i.e., test modules separately first and then as a whole. Their main tool for debugging distributed programs is the logging of traces and the ability to examine the trace data in a coherent manner. They also describe the need for controlled execution, the need to set breakpoints, and the need to monitor distributed processes. The first phase of their two-phase debugging process consists of running the faulty program and by using trace data identify the faulty process or processes and the series of events that led up to the fault. Phase two consists of running the faulty processes in an artificial environment and then debugging until the bug is found. The problem with this approach is that artificial environments may not only be hard to develop but may not model the real-world realistically enough so that bugs that occur because of real-world influences will not show up in the artificial environment.

Smith[Smi84, Smi85] provides the ability to debug the inter-process communication between distributed processes. He provides typical process and message control as well as the ability to test assertions and record events. Daemon processes are used to provide assertion detection with effects. Each daemon has a trigger and a set of commands which can manipulate inter-process events. When a system call occurs, each daemon has its trigger tested. If the trigger fires then its set of commands is executed. Events are recorded with a transcriber and the recorded events can be used to replay a process. To implement and test his debugger, a kernel was written that extended the existing operating system to provide inter-process communication and control. Extensions were made to this kernel to provide a daemon trigger evaluator and event recording.

Bugnet[CW82, WC85, JBW87] is a portable distributed debugger that can reproduce the real-time event sequences of the distributed program. Bugnet monitors a distributed program by giving information on inter-process communication,

input/output events, and trace data. It has the ability to exactly roll back and replay the events that occurred prior to the error while maintaining their real-time characteristics. To debug using real-time event sequences, a checkpoint algorithm was developed that required processes to synchronously start and be checkpointed together. This form of checkpointing and roll back does not have to address the problems of identifying consistent states during roll back, since all checkpoints are synchronous and thus define a consistent global state. However, this checkpointing scheme requires all clocks to be synchronized. Bugnet also implements a graphical user interface and allows process groups to be created for easier process management.

IDD's[HHK85] strategies for debugging distributed programs include state examination, stepping, and tracing; however, its main strategy is assertion monitoring by using a powerful assertion language. The complexity of defining assertions is reduced by using a graphical interface and incremental assertion generation. They claim that a distributed debugger must be embedded deeply into the operating system. This has been challenged by Kenniston[Ken86] and by the implementation for a distributed debugger that follows in Section 5. They also state that it is important to stop a set of processes as quickly as possible to preserve the relative state of the processes for programmer inspection. They, however, give no mechanism to do this and leave it to the users to set an assertion prior to a required state and then step the program to the state in question. A roll back and replay mechanism would alleviate this problem.

Bates and Wileden[BW83b, BW83a] describe an approach to debugging distributed programs called Behavior Abstraction. They view a distributed program as a stream of events that can be filtered and clustered so that unnecessary events can be ignored and groups of events can be abstracted into higher level events. Clustering can be repeatedly used to build a hierarchy of abstractions. Their Event Definition Language[BW82] allows definitions of abstract events to be comprised of both ordered

or unordered sequences of events. Their ordering operator depends on synchronous clocks across processors so events are forced to be totally ordered.

The debugger for Amoeba[Els89] is modeled after Bates[BW83a]. Programs being debugged generate a stream of events, but their debugger goes beyond the capabilities in [BW83a] by providing access to the state of the program. Replay is provided and is similar to that of Instant Replay[LMC87], but cannot always guarantee reproducible behavior. A checkpoint and roll back mechanism also exists. When a checkpoint occurs, a snapshot is taken for the whole program and because communication is synchronous they did not have to implement a consistent global state snapshot algorithm as in [CL85]. Distributed breakpoints are not described but sequential breakpoints do exist. The debugger is built on the philosophy that a distributed debugger does not have to be transparent to the target program because the delays imposed by the debugger, although apparent, are not significant. They also feel that their debugger should not be a monitor for production code so if the program crashes while in production, a postmortem analysis tool should be used to find why the program crashed.

Cheung's dissertation[Che89] presents the use of abstraction for coping with the complexities of debugging distributed programs. The main problems addressed are process abstraction and event abstraction. For process abstraction two views are used. The first is the process view. It provides programmers with the ability to focus on the process structure of a distributed program. This allows the programmer to detect unexpected process death, unexpected process creation, or incorrect process interaction. The second is the temporal view. It allows programmers to focus on the relative ordering of the events among the processes. This view shows the order of events. For event abstraction, the problem of maintaining the precedence relation is studied. Cheung did not implement a debugger. He did, however, implement an

event recording mechanism to study its run-time behavior. This mechanism resides between the application process and the kernel and it simply needs to be linked into the application program at compile time. The implementation of the event recording mechanism required no modifications to the kernel or to the application programs. Analyzing the run-time behavior of the event recording mechanism found that its overhead was small; however, the volume of information being recorded was also small. He only recorded the occurrence of an event and any necessary information regarding that event, for instance its time. He did not record the contents of messages. Finally, he proposes a replay mechanism but does not present the details of one.

2.4 Summary

The work on distributed debuggers is still very immature as compared with the maturity of sequential debuggers. An example of this immaturity is that it is still a challenge to view the *real* state in which the program crashed. Most distributed debuggers supply the sequential debugger's equivalents of tracing, stepping, and stopping, but the specialized tools for distributed debuggers, such as the detection of distributed breakpoints and execution replay, are weak or non-existent. Furthermore, the representation of a program's system state has to be much more sophisticated than what is currently available. For this last point there has been recent work on trying to find a graphical representation for distributed programs[SBN89, Sto89]. The existing debuggers for distributed programs, however, lack some essential facilities such as dynamic roll back and replay which has been included in DPD.

Chapter 3

Problems in Distributed Debugging

The environment in which one debugs sequential programs contains a single processor and a single memory. Programs that run in this environment have only a single execution thread. A major feature when debugging sequential programs is that replaying a program does not usually change the behavior of the program thus allowing programmers to cyclically debug their programs. Distributed debuggers, on the other hand, are burdened with an environment that contains multiple processors, multiple memory spaces, and a relatively low-speed asynchronous communication network connecting the processors. Programs that run in this environment may have multiple execution threads, may not be deterministic given the same input data, and may suffer from uncontrollable communication delays among processes running on different processors. These unfavorable features cause delays when stopping and starting a program, makes it difficult to set and detect breakpoints, causes delays before a system state can be inspected, and makes it difficult to cyclically debug. A distributed

debugger must also synchronize the execution of processes, must collect events from remote processors, must not seriously interfere with the behavior of the program, and must provide a user interface to represent program behavior. These problems make the development of a debugger for distributed programs more complex than the development of a debugger for sequential or concurrent programs. Because of these problems, distributed program debuggers are not as widely developed or used as sequential or concurrent debuggers.

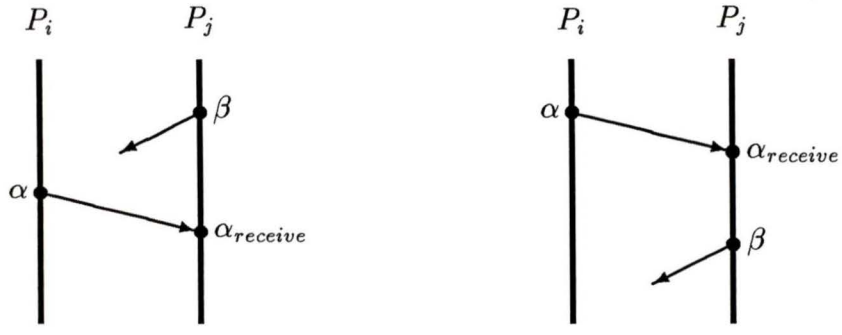
3.1 Communication Delays

Most of the problems associated with distributed debugging are directly related to communication delays such as the non-determinism of events, the failure to synchronize, the difficulty of maintaining consistent states, the inability to display the current global state of a running program, and the inability to stop processes in a timely manner. The remaining sections touch on all of these problems in more detail.

3.2 Non-determinism of Events

The human debugger relies on the determinism of events when debugging sequential programs. This dependence on determinism is not valid in a distributed system. Non-determinism of events is caused by constantly changing communication delays and by the differing speeds among processors in the distributed system. In an environment such as the above it is possible for a program to fail even though it is thought to be correct; it is also possible for a known faulty program to run correctly occasionally.

For example, assume the execution of a program where event β occurs before event α. This order of events always holds in a sequential environment even if the



Desired event sequence Unexpected event sequence

Figure 3.1: Desired and unexpected event orderings

environment changes, (e.g. the load of the processor). In a distributed system where the events α and β occur on processors P_i and P_j, respectively, there may be different event orderings in different runs[Lam78]. Suppose α is the event *send message to processor* P_j and β is the event *send message to processor* P_i and there are the corresponding events $\alpha_{receive}$ and $\beta_{receive}$ on processors P_j and P_i, respectively. Suppose β is expected to occur prior to $\alpha_{receive}$, but no checks were made to assure this. Suppose also that the programmer assumed that β would always occur prior to $\alpha_{receive}$, because of the expected short computation time required to reach β as compared to the expected long computation time to reach α. If processor P_j is exceptionally slow or processor P_i is exceptionally fast then event $\alpha_{receive}$ could occur prior to β causing unexpected results. Furthermore, if the combination of a fast P_i or a slow P_j is a rare occurrence, then this bug may be very difficult to track down.

There are two methods for debugging such a situation. One method is to continually monitor a distributed program even when it is in production. When the program crashes, a trace dump is analyzed and the order of events is determined. Since the se-

quence of events shows that α and $\alpha_{receive}$ occurred before β, the reason for the crash can be determined. The problem with this approach is that it is not usually feasible to continually monitor a production system especially when bugs, such as the above, may occur only rarely. The other approach is to have a debugger that can change the order of events so that different event sequences can be tested. The problem with this approach is that it may not be feasible to test every event sequence; however, those critical event sequences that may lead to a failure can be rigorously tested.

3.3 Synchronization

Synchronization problems arise because the state the distributed debugger perceives the application program to be in might not be the state the application program is actually in. For instance, a command might be given to stop a running process; however, before the command arrives the application process stops. The other problem with a distributed debugger is that all the parts of the debugger must be kept synchronized with one another.

3.4 Consistent States

Distributed debuggers must be capable of identifying and maintaining consistent system states. A system state of a program is inconsistent if the said state cannot be reached during the normal execution of the program. An example of an inconsistent state is where one process has received a message that has not yet been sent. Identifying and finding consistent states in a distributed system is discussed in [CL85, SK86, KT87, JZ88]. There are two situations where maintaining consistent system states becomes important for distributed debuggers. The first is displaying

the system states to the user. If delays prevent event information from arriving at the client processor, the user may see *receive message* events without the corresponding *send message* events. The second situation is when roll back and replay becomes a factor. Suppose that we have a program consisting of two processes, P_i and P_j. A message is sent from P_i and is received by P_j. If the program is stopped and P_i is rolled back to a state prior to the sending of the message and P_j is not rolled back to a state prior to receiving the message, the program is in an inconsistent state. In this example, not only a message appears to have been received prior to it being sent, but continuing the program will cause P_j to receive a duplicate message.

3.5 Stopping Processes

Stopping a process in a distributed environment is much more time consuming than in a sequential environment. The reason is communication delays. When stopping a process, one expects it to be stopped in the state in which the command is given. This is not possible in a distributed environment for two reasons. The first being the latency in sending the stop command from the client processor to the remote processor. The second being that the current state being viewed by the human debugger may be out-of-date with respect to the actual state the program is in.

These two reasons make it impossible to stop a process in a distributed program instantaneously. For example, suppose there exists an application process executing on a remote processor P_j. The command to stop the process is given from processor P_i at time t. It takes time s for the command to be sent through the network and to be performed on P_j. Before the stop command reaches the application process, the latter has executed an additional s time and is, thus, in a different state from what it was at time t. It is possible that important information available only at time t is

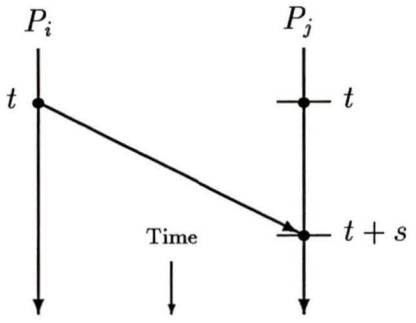


Figure 3.2: The latency in stopping a distributed program

not available at time $t + s$. It is, therefore, extremely difficult, if not impossible to stop all of the processes of a distributed program instantaneously.

3.6 Replay

Normally, one is not concerned with roll back and replay in a sequential environment because of the deterministic behavior of its programs. If one needs to replay a program, one can always replay from the program's initial starting point. If one does want to implement roll back with replay, for those long running sequential programs where it would not be feasible to re-start from the beginning, then checkpointing is possible without much difficulty. This is because (1) the state of the process does not change during a checkpoint, (2) there is only one memory space to be saved, (3) checkpoints can occur at almost any time during execution, and (4) checkpointing has no effect on a sequential program other than the delay involved in performing the checkpoint. To roll back a sequential program simply requires the restoration of a checkpoint. Since sequential programs are deterministic, roll back and replay can occur many times and the program's behavior does not change during replay.

To produce a snapshot in a distributed environment requires all processes to be checkpointed. One minor drawback when checkpointing in a distributed system is the extra storage space required to save every process's system state. A more serious drawback is that it is impossible to take an instantaneous snapshot of a distributed system. To find a consistent global state to roll back to, requires either implementing a snapshot algorithm that guarantees a consistent global state[CL85, SK86], or implementing a sophisticated algorithm to find a consistent state during replay[JZ88]. When implementing the latter, the *domino effect*[JZ88, KT87] can occur. This is where a consistent state cannot be found in the set of all checkpoints causing the program to be rolled back to its initial state.

When a roll back occurs in either a sequential or a distributed program, the effects to the outside world cannot normally be rolled back. Examples of side-effects are the sending of text to a printer or the launching of a missile. This problem can have an effect on a program especially if the program updates a database which may make it impossible for the same sequence of events to occur again. For example, suppose a process checks the existence of a record in a database. If the record does not exist, the process performs the event sequence E_1 and adds the record. If the record does exists, the process only performs event sequence E_2. Now suppose the process is running and finds the non-existence of the record. It performs the event sequence E_1 and adds the record. If the process is rolled back prior to checking for the existence of the record, the process finds that the record does exist and, thus, performs event sequence E_2. This problem is beyond the scope of this thesis.

3.7 Trace Information and the Representation of Events

In a sequential environment there is a single process producing a single stream of trace data. All events are totally ordered and the time between events can be used as a measure for the computation between events. In a distributed environment there may be a large number of processes producing multiple trace streams. If the trace streams are merged into a single stream, a total order is forced on a set of events where only a partial order is possible. The time the user sees among events cannot be used as a measure for computation because of varying communication delays. Furthermore, the amount of trace information produced in a distributed environment can easily overwhelm the user.

The *happened before* relationship[Lam78], written A $\rightarrow$ B, can be used to partially order events in a distributed system. This relation states:

1. If a and b are events in process P_i, and a comes before b, then $C_i\langle a\rangle < C_i\langle b\rangle$, where C_i is the clock for process P_i.

2. If a is the sending of a message by process P_i and b is the receipt of that message by process P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$.

Those events that do not fall into the above relation cannot be ordered. For instance, in Figure 3.3 one can state $P_1.e_1 \rightarrow P_2.e_1$, $P_1.e_1 \rightarrow P_1.e_2$, $P_1.e_2 \rightarrow P_1.e_3$, $P_1.e_3 \rightarrow P_1.e_4$, $P_2.e_1 \rightarrow P_2.e_2$, $P_2.e_2 \rightarrow P_2.e_3$, and $P_2.e_3 \rightarrow P_1.e_4$. No relationships can be drawn between $P_1.e_2$ and $P_2.e_1$, $P_1.e_2$ and $P_2.e_2$, $P_1.e_2$ and $P_2.e_3$, $P_1.e_3$ and $P_2.e_1$, $P_1.e_3$ and $P_2.e_2$, and $P_1.e_3$ and $P_2.e_3$. Thus, for those events where no relationship can be drawn no ordering can be imposed. Because there has to be some means to

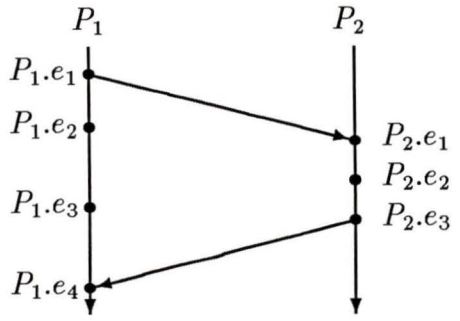


Figure 3.3: Event Order

present the order of events in a distributed system, a sophisticated event viewing facility must exist.

Since events are recorded on remote machines, the time between events, especially for events occurring on different processors, cannot be represented in real-time. This is because the clocks in a distributed system may not be synchronized. To represent time in a distributed system one must use logical clocks[Lam78]. Logical clocks are maintained by the following rules:

1. Each process P_i increments C_i between any two successive events.

2. (a) If event a is the sending of a message m by process P_i the message m contains a timestamp $T_m = C_i\langle a\rangle$.

   (b) Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

This means that if on one processor the real-time between two events is large, the logical clock is only incremented once; however, if on another processor many events occur rapidly, the logical time between the first and last event is large.

A distributed program may be made up of many processes distributed across the network. There may be an overwhelming volume of trace data that the user cannot analyze. An ability to abstract events is required to assist the user in analyzing the event information. Usually a graphical representation to view the abstract events is chosen.

3.8 Breakpoints

To set a breakpoint in a sequential program one states an assertion with respect to a single time reference. This allows a sequential program to be stopped immediately. When a breakpoint occurs it leaves the program in the exact state in which the breakpoint is detected. There are two problems with breakpointing in distributed systems:

1. Detecting the occurrence of a breakpoint.

2. Stopping the system in both a consistent state and in a state where that breakpoint still holds.

In a distributed system there are multiple time references. This means that one needs to set breakpoints based on the *happened before* relation and the ability to detect simultaneous events without the advantage of a single time reference. If one were to state a simple assertion based on one process in a distributed system, then as soon as that assertion is met the whole system can be stopped. This situation leaves the breakpointed process in the exact state in which the breakpoint occurred. If we were to set an assertion based on two events occurring in different processes on different processors and there is no *happened before* relation between the two events, then it is

difficult to stop the system in a state where both breakpointed processes are in the state in which their individual breakpoint is reached[MC88, page 318]. Furthermore, trying to stop a distributed program when a breakpoint is reached has the same difficulties as trying to stop a distributed program in general. This is compounded by wanting to stop the system in a state in which the breakpoint occurred. Miller[MC88] uses Chandy and Lamport's algorithm[CL85] to stop a distributed program in a consistent state after the breakpoint has occurred. Fowler and Zwaenepoel[FZ89] use roll back and replay to stop the distributed program in the largest consistent state that has happened before the state of the processes in which the breakpoints occurred.

3.9 Summary

This chapter has presented the hard problems inherent in both debugging distributed programs and in designing a debugger for distributed systems. Most problems such as non-determinism, synchronization, maintaining consistent system states, and stopping processes are caused by the communication delays that exist in distributed systems. Other problems that are not based on communication delays are the multiple execution threads and the volume of trace information.

Chapter 4

REM

DPD provides debugging tools for programs developed for REM[Sho90, SCTT88, SCT88]. REM, *Remote Execution Manager*, is a tool to build fault tolerant, distributed programs. It provides facilities for process management, load sharing, and inter-process communication. The main objectives of REM are to utilize the idle processing power of a local network of workstations, provide an easy to use user interface, and achieve transparent fault tolerance in case of a workstation failure. REM is modular and is written in the C Programming Language[KR78]. It currently runs on a network of Sun workstations running SUN OS 4.0.[1] An important feature of REM is that it requires no modifications to the UNIX kernel and is implemented entirely at the application layer. Because of this, REM is portable across UNIX systems that provide a *socket* facility.

An overview of REM is presented in Figure 4.1. The REM environment consists of two components, a set of communicating daemon processes and REM's Application Interface Library. The daemon processes are distributed across the network of

[1]SUN OS 4.0 is based on BSD UNIX 4.3.

workstations and provide the means to create, communicate with and terminate remote application processes. These daemon processes are fully distributed without any centralized control. The second component, REM's Application Interface Library, is combined at link time with the processes of an application program and provides the means by which an application program can interface with REM.

A socket[San90] provides the facilities for inter-process communication. Protocols understood by sockets are ARPA Internet protocols (internet) and UNIX system internal protocols (interunix) among others. There are three socket types: stream sockets, datagram sockets, and raw sockets. REM uses the stream socket which provides bidirectional, sequenced, and unduplicated flow of data.

Before continuing, we introduce the terminology used in the REM model.

Definition 1 An *application program* is a set of distributed, communicating processes that work together to reach a common goal. Each application process uses the routines in REM's Application Interface Library to communicate with and control other processes of the application program.

Definition 2 The *parent application process* is the first process invoked when execution of the application program begins. There exists at most one parent application process per application program and all other application processes are offsprings of this process.

Definition 3 A *child application process* is a remote process created by the parent application process through the facilities supplied by REM. There may exist many child processes per application program.

Definition 4 A *suite* is a collection of daemon processes. Each REM daemon process has knowledge of only the other daemon processes in the suite.

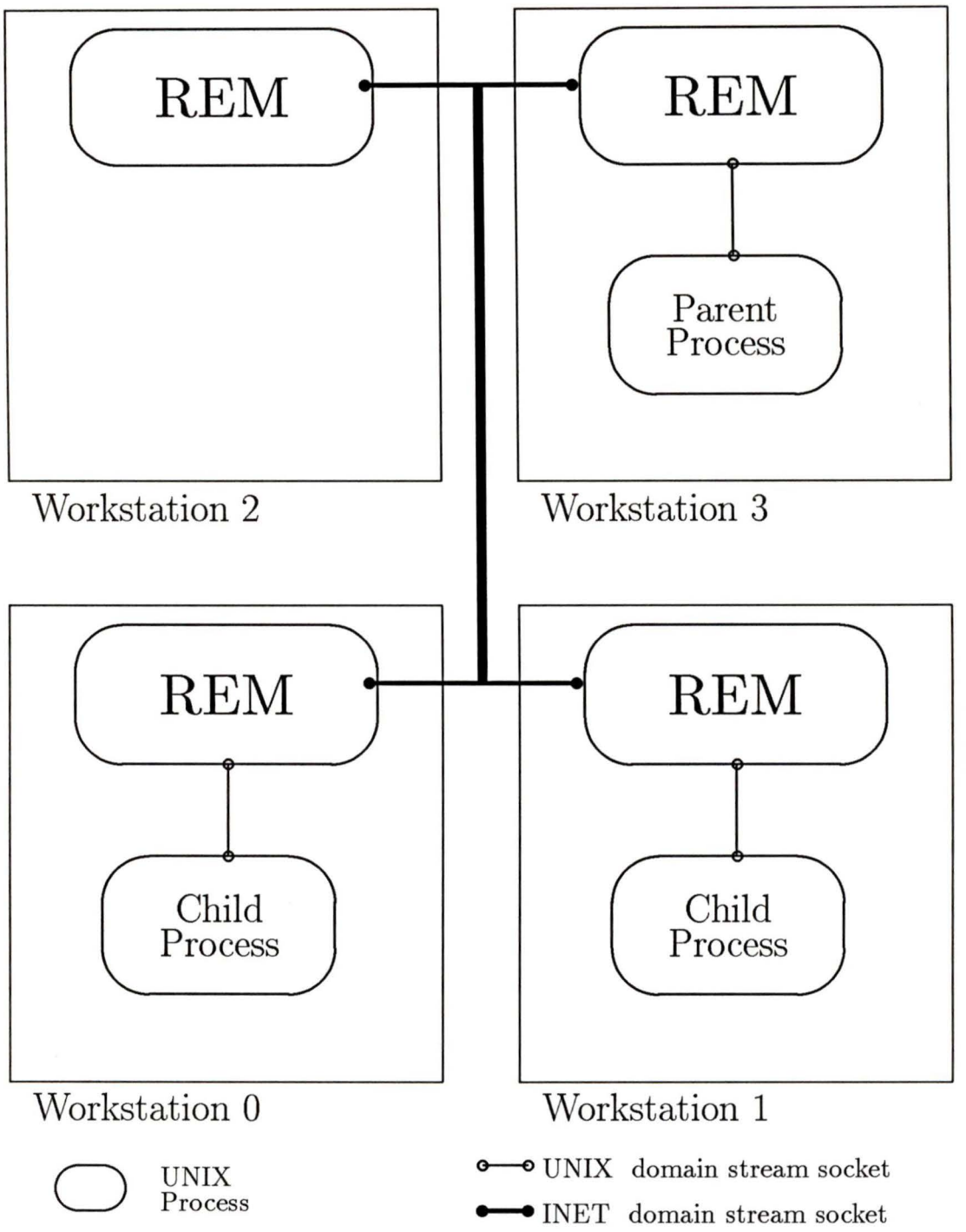


Figure 4.1: An Overview of the REM Architecture

Definition 5 The *client workstation* refers to the workstation where an application program is invoked (the parent process). *Remote workstations* are all other workstations in the suite and are where the child application processes reside.

In the REM model, an application program consists of a parent process and a set of child processes. The parent process resides only on the client workstation and child processes reside only on the remote workstations. More than one child process may occupy the same remote workstation if there are more child processes than workstations or if REM's load-balancing algorithm decides that it is favorable to double-up a workstation because the load average of all other workstations is too high. Before the parent process can create a child process, it must connect to REM through the interface routines supplied by REM's Application Interface Library. All communications are parent-to-child or child-to-parent, there are no facilities to provide child-to-child communication. Before a child process can communicate with the parent process or vice-versa, it must connect to REM. Before each application process terminates it must close its connection with REM. Each process is assumed to be deterministic between receive events. This means that if a program is re-run, and if all messages arrive at the processes in the same order and contain the same information, then the program produces the same result. This is not only important for roll back and replay, as described in Section 5.4.7, but also for fault tolerance.

A suite is a collection of daemon REM processes. Daemons of a particular suite know only of other daemons in the suite and correspondingly know nothing of the ones belonging to other suites. There can be more than one active suite on the network and an individual workstation may have more than one suite running on it. An application program can belong to only one suite and its processes can reside only on workstations that have an REM daemon of that suite. The purpose of suites is to provide varying degrees of fault tolerance which may be needed for execution of

different programs.

REM attempts to create child replicas on different workstations to maximize fault tolerance. A workstation may contain at most one replica of a child process. All child replicas run in unison. This means that when the parent sends a message to one of its child processes, that same message is sent to all replicas of that child process. When a child process sends a message back to the parent process, each replica of that child sends that same identical message back to the parent process because all application processes are assumed to be deterministic between receive events. The parent process receives only the first message to arrive from that child or one of its replicas, all other identical messages are ignored. This means that the parent process receives messages from the fastest running replica. If a workstation containing a child replica fails, the parent process is unaware of this failure, because the parent continues to receive messages from one of the other child replicas. This means that the degree of fault tolerance is dependent on the number of child replicas.

4.1 Extensions

Extensions to REM were needed for the development of DPD. The original functionality of REM was sufficient for many distributed applications; however, implementing DPD would have been difficult, if not impossible, without these following extensions. Furthermore, the following extensions increase the usefulness of REM by allowing a larger class of distributed applications to be built with less work by the programmer. REM is now being used as the target environment for a distributed ray-tracer[Cor90] and as the target of an automatic parallelization compiler[Sin90]. The following section discusses the important extensions to REM including asynchronous receives, message queues, and socket flushing.

4.1.1 Asynchronous Receives

REM was originally implemented with only a blocking receive primitive. This primitive, `us_recv_dist`[Sid90], requires the receiving process to specify the source process of the message. When invoked, the receiving process blocks waiting for a message to arrive from that source process. If the sending process has not sent the message, the receiving process remains blocked even if messages arrive from other processes. This prevents messages from being received and processed in a first-come first-serve manner. This could be considered a non-efficient use of the available processing power. To allow messages to be received in a first-come first-serve manner asynchronous receives were implemented to extend REM's existing receive primitive.

To implement asynchronous receives, the programmer is required to declare a function which is invoked whenever the process is in asynchronous receive mode and a new message arrives. This function is called the *asynchronous handler* (see Appendix A). Within the asynchronous handler, the programmer is required to call `us_recv_dist` to actually receive the message. When the asynchronous handler is invoked, the sender's process identifier, the length of the message, and the type of the message are passed as arguments to it. This is sufficient information for the programmer to call `us_recv_dist`. Because the asynchronous handler is invoked only when a message is available, `us_recv_dist` does not block when called with the arguments passed to the asynchronous handler.

For the programmer to register the asynchronous handler with REM, a new routine was added to REM's Application Interface Library. This interface function is called `us_s_asynchandler` and its only parameter is a function pointer to the asynchronous handler. Another effect when `us_s_asynchandler` is called is that all messages from this time forward are received asynchronously. If the programmer no longer wants to

receive messages asynchronously, a call to `us_s_asynchandler` with a NULL function pointer causes the application process to return to receiving messages synchronously.

We now describe in detail how messages are received asynchronously, but before we continue, we first define two new terms.

Definition 6 A message is considered to have *arrived* at an application process when the internal routines in REM's Application Interface Library have read the message from the socket. Since the message is not addressable by the application process, the application process has no knowledge that a message has been read from the socket.

Definition 7 A message is said to be *received* by the application process when the message is transferred from the memory controlled by REM's Application Interface Library to the working memory of the application process.

When `us_s_asynchandler` is called with a valid asynchronous handler, all arriving messages are handled asynchronously in a first-come first-serve manner. However, there is a possibility that prior to `us_s_asynchandler` being called, there could be messages that have arrived but are not received yet. When this occurs, each message that has arrived but is not received causes the asynchronous handler to be invoked. This means that if there is a backlog of messages, all these messages are serviced before any newly arriving messages are serviced.

An important feature of asynchronous receives is that if a process is compute bound, it is able to receive messages without having to poll or wait. Sometimes, however, a process may want to receive messages asynchronously although it is not compute bound. In this situation, there exists an interface function called `us_s_waitasync` which causes the process to block waiting for new messages. `us_s_waitasync` returns

only when the application process changes from asynchronous mode to synchronous mode.

Asynchronous receives were implemented by catching the SIGIO[Sun88] signal. The communication socket used by REM is configured, through a call to `ioctl`[Sun88], to raise this signal when input is available. A signal handler is declared which, when invoked by an IO signal, first calls REM's low-level routine to read from the socket and then call the user's asynchronous handler. When the signal handler is active all further SIGIO signals are blocked. This last point means that while receiving an asynchronous message, processes are not interrupted to service another message. When the asynchronous handler is finished with its current message, any blocked signals are then handled.

In keeping with REM's modular style, a new module was created to hide how a socket is prepared for asynchronous receives. This module was added to REM's Application Interface Library.

4.1.2 Queues

In REM's skeletal prototype, messages are queued only within REM's Application Interface Library. Furthermore, messages are queued only when a process attempts to receive from a specified source process and a message arrives from another process prior to the one from the specified sender. The REM daemon processes, however, never queued messages. If a socket is busy, because the receiving daemon is not processing as fast as the sending daemon, the sender waits until the receiver has read and serviced all of its pending messages. Originally, this did not affect REM, since it is guaranteed that the receiving daemon reads the messages in short order and the sending daemon need not wait indefinitely. The major reason for not implementing

message queues within REM daemon processes is because of the additional overhead involved in checking and maintaining queues. In a networked system, the bottle-neck is not the transmission time to send a message but the overhead in preparing the message to be transmitted. One REM goal is to keep this message preparation time to a minimum. See [SCT88] for benchmarks.

Not queueing messages became a problem in the following scenario. There existed two application processes, α and β. α continually sent messages, one after another, to β. β could not read the messages as fast as they were being sent, because of long computations being performed between receiving new messages. This caused a backlog of messages to build from α. This backlog caused the REM daemons involved in forming the communication channel from α to β to block for an extended period of time due to full sockets. This situation caused other application processes that were attempting to communicate with these daemon processes to block until α and β finished communicating. Under normal conditions this is not a problem, since other processes attempting to communicate with those REM daemons would behave as if the communication network is slow.

During the development of DPD this situation occurred causing DPD to loose control over the processes it controlled. When DPD stopped β, α was left running. Very quickly, the socket buffers became full and the REM daemon processes became blocked. This made all communications with these blocked REM daemons impossible. Since the REM daemons were blocked there was no way to send a message from DPD to β to start it again. This meant DPD and the REM daemon processes became deadlocked: β was waiting on DPD to start it, the REM daemons were waiting on β to start receiving messages, and DPD was waiting on the REM daemons to except new messages.

The queue module was added to REM to solve the above dilemma. The actual performance decrease is minimal if the socket buffers never become full[Sho90]. When

messages are queued, REM polls every two seconds[2] to see if the socket can accept new messages. If during this time the socket can accept new messages, queued messages are written to the socket. Not all the queued messages are sent during this time even if the socket can accept all the messages. The reason for this is that if the sender is continually sending messages and the receiver becomes capable of receiving all of them, then this situation prevents any other process wanting to communicate with REM to block until the sender and receiver are finished communicating. Restricting the number of queued messages that can be sent at any one time prevents two communicating processes from starving other processes that are trying to communicate with REM.

4.1.3 Socket Flushing

Replay requires the communication channels to be clear of any messages so that no spurious messages arrive at a process after it is rolled back. This implies all message queues are to be emptied and all sockets are to be flushed. Emptying the queues is straight forward; however, there is no means to flush a socket. To clear sockets, each process that has just been rolled back sends a *flush communication channel* message to REM. This message is received by REM causing the corresponding message queues for the process to be emptied and an *end of socket* token to be sent back to the process. The application process reads the socket until the *end of socket* token is read, discarding any messages that precede the token. Once the token is read, the communication channel is considered to be flushed. Since flushing the communication channel may be used for purposes other than debugging, it was added to REM's Application Interface Library.

[2]It is more efficient for REM to be signaled when the socket can accept new messages; however, no mechanism was found to provide this capability.

Chapter 5

Implementation of DPD

The *Distributed Program Debugger*, DPD, was inspired by the need for sophisticated but easy to use debugging facilities for distributed systems. A paradigm for debugging sequential programs with a standard sequential debugger is analogous to debugging that program microscopically, i.e., the human debugger can inspect every detail of the program. Debugging with most distributed debuggers can be thought of as debugging macroscopically, i.e., the human debugger is trying to abstract the behavior of the program. When debugging distributed programs macroscopically, one is more interested in the communication behavior, the temporal relationships, and the overall global state of the program. Trying to debug a distributed program microscopically is futile, because one cannot possibly comprehend the overall global system state at this level. However, to debug complex distributed programs only macroscopically may leave the human debugger without the ability to explore the reasons as to why a program is behaving incorrectly. DPD tries to balance the micro and macro needs of a distributed debugger by supplying the ability to control and probe the behavior of distributed programs while allowing the human debugger to view the distributed program as abstract events and processes.

In this chapter, we present a detailed description of the development of DPD. We first present an overview of the debugger and then describe the intrinsics of the debugger including event recognition and collection, user interface, process control, and checkpointing. Finally, we describe the roll back feature.

5.1 Overview

DPD is comprised of three types of processes: *User Interface Process* (UIP), *Remote Debugger Process* (RDP), and *Event Collection Process* (ECP). An overview of a DPD configuration is presented in Figure 5.1. The User Interface Process resides on the client processor. It is responsible for coordinating all other processes in the DPD environment, controlling the parent application process of the target application program and presenting a graphical interface to the user. A Remote Debugger Process resides on every remote processor that contains an REM daemon of the same suite in which the User Interface Process is invoked. Each RDP is responsible for controlling the remote child application processes running on its processor. The RDPs are also responsible for conveying to the UIP the state information of the child application processes that they control. The Event Collection Processes are responsible for collecting event information from the application processes and for initiating the transfer of this information to the UIP. Since the ECP for the parent application process resides on the same processor as the UIP, the ECP has a direct communication channel to the UIP. The ECP for a child application process has an indirect communication channel to the UIP through the local RDP. The interunix communication channels between the Event Collection Process and the Remote Debugging Process and the User Interface Process are through UNIX's `pipe`[1] facilities.

[1]A pipe is a means for two processes to communicate in UNIX built on top of sockets.

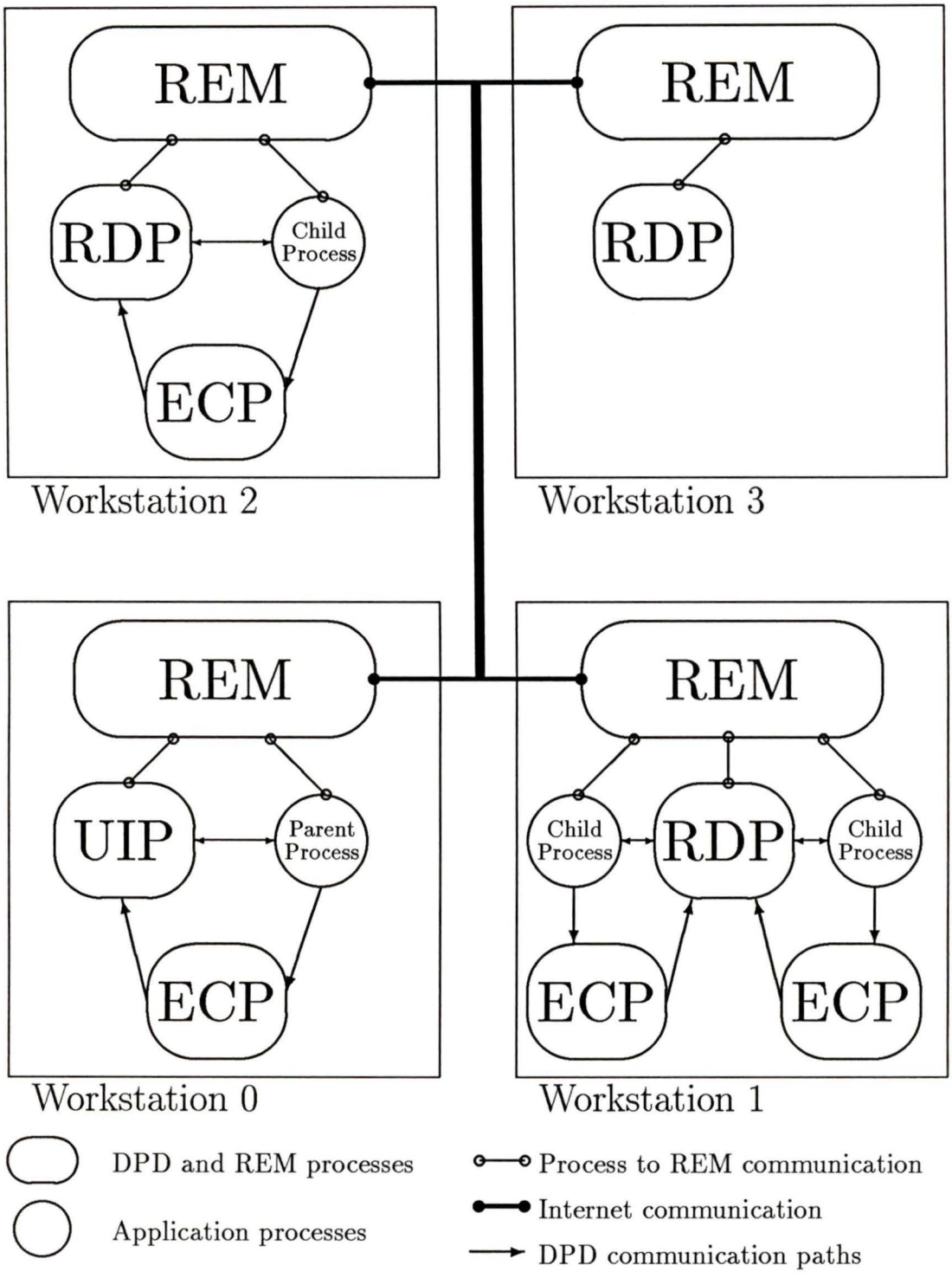


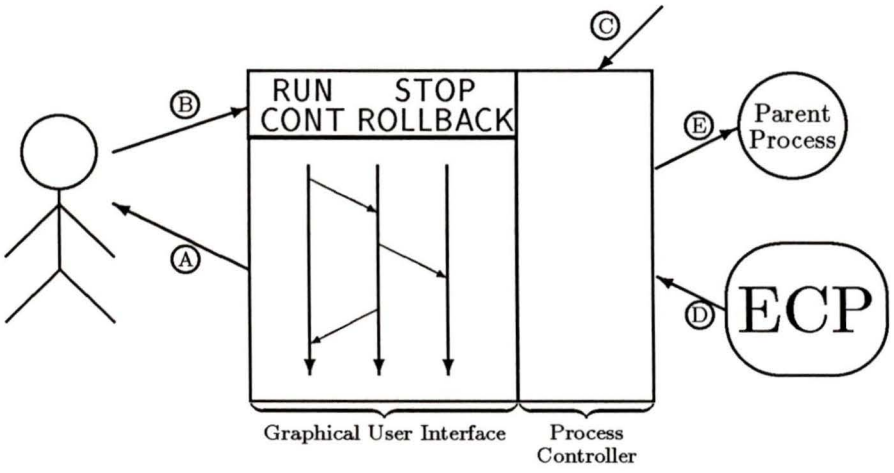
Figure 5.1: An Overview of DPD

DPD was designed to be a standard REM application program within the REM paradigm. The UIP is the parent process and the RDPs are the child processes. All communications between the UIP and RDPs are through the facilities supplied by REM. All communications are point-to-point between the UIP and individual RDPs. DPD required no modifications to the UNIX kernel, but did DPD require changes to the REM nucleus as described in Chapter 4. The remainder of this section describes the individual processes that comprise DPD.

5.1.1 User Interface Process

The main responsibilities of the User Interface Process are to control and coordinate the Remote Debugging Processes, directly control the application program's parent process, graphically display the application program's system state, and accept commands from the user. An overview of the UIP is presented in Figure 5.2. The UIP is comprised of two main components. The first is the Graphical User Interface which depicts a graphical representation of the progress of an application program. The second component is the Process Controller which controls all application processes either directly in the case of the parent process or indirectly via the RDPs in the case of remote child processes. The process controller component handles commands after they are entered by the user. If the command affects a child application process, it is packaged into a message and sent to the necessary RDP. If the command affects the parent process, the command is passed to the low-level routines that handle process control.

Since the UIP is the parent process of the DPD, it is the process invoked by the user. The first item on the UIP's agenda when invoked is to connect with REM. Following a successful connection, the UIP sends a message to REM announcing that



Ⓐ Event information is displayed in the Display Window to the user
Ⓑ User issues commands from the Command Window
Ⓒ Event information arrives from the RDPs
Ⓓ Event information arrives from the parent's ECPs
Ⓔ The UIP controls the parent process

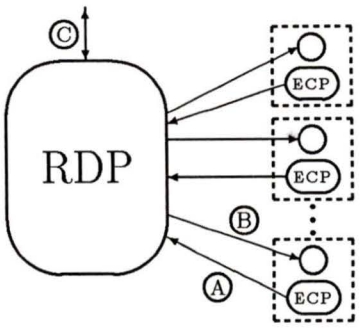
Figure 5.2: An overview of the UIP

it is a debugger and that it requires the ability to create one RDP on every remote processor of the invoked suite. Following the successful creation of all the RDPs, the UIP displays its graphical user interface and is ready to accept commands from the user.

The underlying graphical user interface is the SunView windowing environment. This environment provides a bit-mapped display, a facility to manage windows, and enhanced features such as menus and browsers. A feature of the SunView windowing environment is that user input is event-based and asynchronous. This means that the UIP never blocks waiting for user input. SunView also provides the ability to build bit-mapped displays larger than the window the image resides in. This is necessary for our application since long-running programs may produce many events that cannot be displayed in the area limited by a window. A detailed description of the user interface is presented later in Section 5.3.

The User Interface Process is responsible for creating and controlling the parent application process. Since REM allows at most one parent process to exist per application program, the UIP directly controls at most one application process. Controlling the parent process is handled identically to controlling a child process by the RDPs. A detailed description of process control is presented later in Section 5.4. Since the UIP controls an application process, it collects event information arriving from that application process's Event Collection Process. Information arriving from the ECP is handled asynchronously and, thus, no processing is required by the UIP until information is available. A detailed description of event handling is presented later in Section 5.2.

The UIP is also responsible for controlling all the Remote Debugging Processes. Since REM provides only parent-to-child communication there are no mechanisms to provide RDP-to-RDP communication. Therefore, the UIP acts as the central con-



Ⓐ Event information is collected from the ECP
Ⓑ RDP controls the child application processes
Ⓒ Commands arrive and results are sent to the UIP

Figure 5.3: An overview of the Remote Debugging Process

troller and synchronizer to all the RDPs. This model of a single parent process controlling several child processes conforms to the REM paradigm. All information arriving from the RDPs is received and handled asynchronously through the asynchronous receive facility added to REM.

5.1.2 Remote Debugger Process

A *Remote Debugger Process* is responsible for managing all application processes that are being debugged on its remote processor, as shown in Figure 5.3. One Remote Debugger Process exists for every remote processor in the suite for which the UIP is invoked. The RDP's responsibilities are to carry out commands received from the UIP and to transfer application process's event information and state changes to the UIP. Since REM allows more than one application process to reside on a single remote processor, an RDP may control more than one application process at a time.

RDPs are also responsible for creating the remote child application processes that

are going to be debugged. When an application program is not going to be debugged, the REM daemons are responsible for creating the child processes. The reason for creating the child application processes is to gain debugging control over them. When an REM daemon receives a message to create a child application process, it checks to see if the application process is going to be debugged. If the application process is not going to be debugged, it is created by the daemon. If the application process is going to be debugged, the daemon sends a message containing the child's name and arguments to the RDP. The RDP attempts to create the child process and sends back to the daemon either a positive or negative acknowledgement depending on whether the child process is successfully created or not. Once an acknowledgement is received by the daemon, it continues processing as if it had just created the child process itself.

Another responsibility of the RDP is to transfer to the UIP the event information collected by the Event Collection Processes from child application processes. The ECPs on the remote processors do not have a direct link to the UIP. Thus the RDPs provide the means in which the event information can be transferred to the UIP.

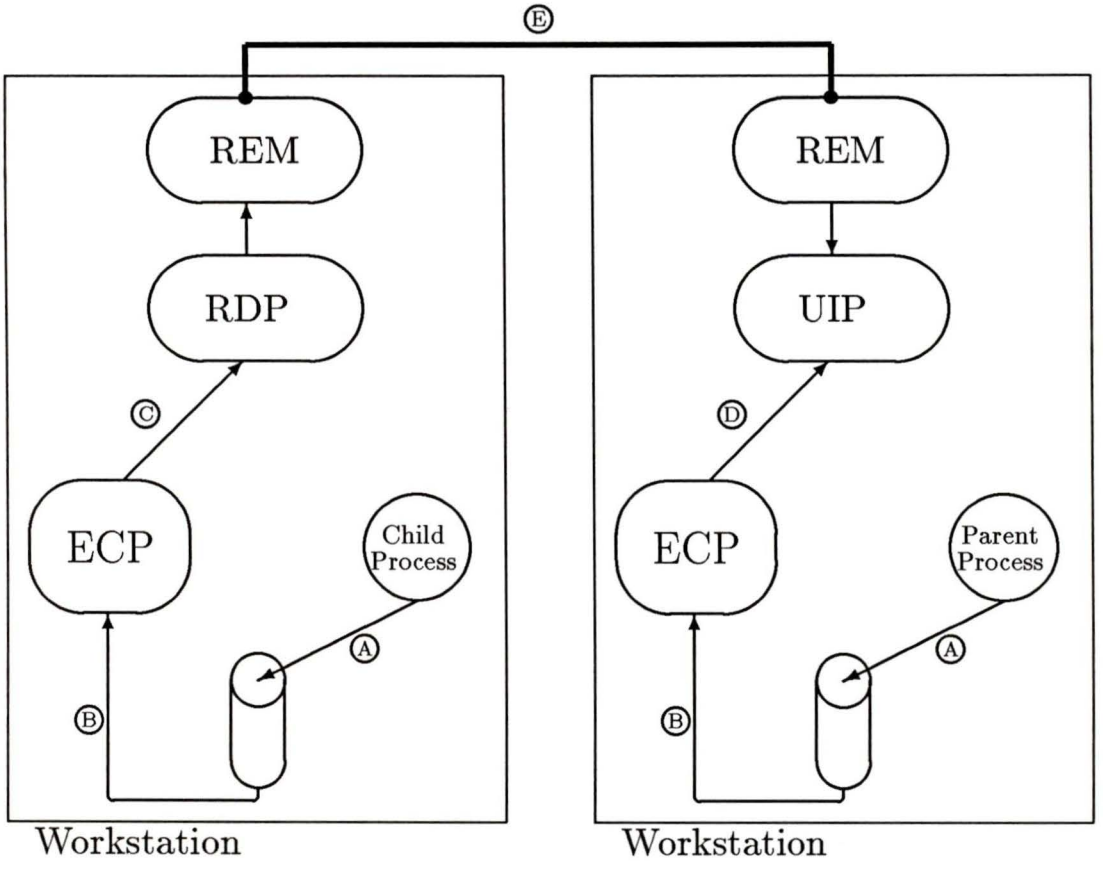
5.1.3 Event Collection Process

Event Collection Processes are responsible for collecting application process's event information and for initiating the transfer of this information to the UIP. There exists an ECP for every application process. To generate their event information, all application processes record their events to an event file when they are being debugged. This is facilitated by an *Event Record Module* which is transparently embedded in the application process. There are two communication paths which may be used to transfer event information to the UIP as shown in Figure 5.4. The first is a direct path from the ECP to the UIP. This path is only possible when the

ECP and its application process reside on the client processor. The second is an indirect path through an RDP. This is required when the ECP and its application process reside on a remote processor. The ECP communicates with the UIP or an RDP through a directed `pipe`. The ECPs do require some initialization information to be sent to them. This information includes the identifier of the process from which the events are to be collected, and the file descriptor with which the pipe is associated. This information is conveyed through command line arguments of ECP.

An application process writes its event information to a specially named event file through the facilities supplied by the *Event Record Module*. This event file is named DBG_<*process_id*>, where <*process_id*> is the UNIX identification number of the application process. Once an ECP is created and initialized, it blocks attempting to read an *event header* from the event file. When a process records an event, the event record module writes an event header to the event file. This event header contains information such as what type of event occurred and the virtual time of the event. The event header may be followed by additional information such as the contents of a message or the name of a newly created child process. Once the event header is written to the event file, the ECP unblocks by reading the header. If there is additional information following the header, the ECP records the size of this information so that it can be read later. The event header is then written to the pipe. The ECP then reads the additional information if any and writes it to the pipe. Finally, the ECP blocks again waiting for the next event header to be written.

In addition to event information being written to the event file, the event record module also writes information about checkpoints. When a checkpoint occurs, the process first checkpoints itself to a separate checkpoint file and then writes a standard event header to the event file. This event header is followed by the name of the file used to checkpoint the process. The ECP reads the event header and the name of



Ⓐ Event information is written to disk file from application process
Ⓑ ECP reads event information from file
Ⓒ Ⓓ ECP transfers event information to either the RDP or the UIP
Ⓔ RDP uses REM to transfer event information to the UIP

Figure 5.4: An overview of the Event Collection Process

the checkpoint file from the event file and transfers this information to the UIP just as if this information describes a standard event.

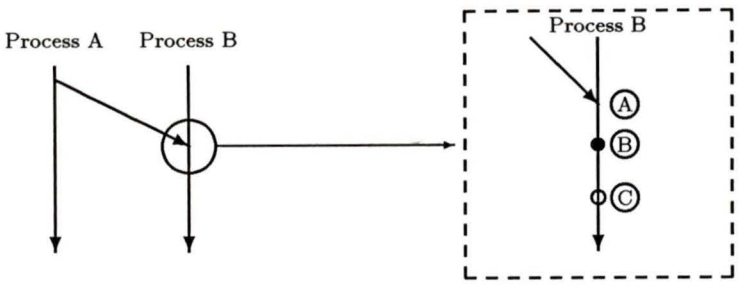
Within the application process resides an *Event Record Module* which is responsible for recording event information to the event file. The hooks to this module reside in REM's *dist*[2] module and, thus, the event record module is transparent to the application programmer. Furthermore, this module is never invoked if the application process is not being debugged. Obviously, writing to an event file may not be the most efficient mechanism for logging events; however, it is assumed that a process being debugged has additional delays. If a more efficient mechanisms for recording events is implemented, only the ECP and the event record module need to be changed.

5.2 Recognition of Events

Events have to be recorded in a reliable manner in order for the DPD to display the execution of a distributed program and to maintain consistent system states during roll back. This requires writing the necessary event information to disk, as discussed in Section 5.1.3, and guaranteeing that the program cannot be in an inconsistent state with respect to the recorded event information. This section describes when events are recognized and when they are recorded. This section makes use of Definition 6 and Definition 7.

Events that are currently recognized are: *receive message*, *send message*, *create child process*, *initialization*, and *termination*. The following rules describe how events are recorded.

[2]The *dist* module contains the user interface routines for REM.

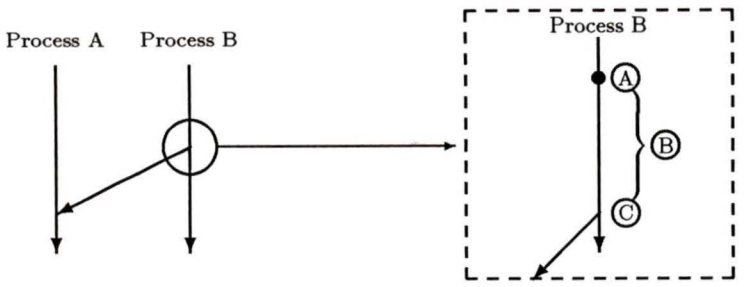


Ⓐ Event has *arrived*; it is read from the socket
Ⓑ Event is recorded
Ⓒ Event is *received*; message is in memory addressable by the application process

Figure 5.5: Recording of a *receive message event*

Rule 1 A *receive message event* is recorded after the message has arrived, but before it is received, as shown in Figure 5.5. If the program is interrupted after the message has arrived, but before the message is recorded, or during the time the message is being recorded, the DPD does not show the message as being received and does not display the event to the user. If the process is interrupted after the message is recorded, but before the message is received, the DPD shows the message as being received and displays the event to the user. This situation does not cause problems because the message is addressable by DPD and is not affected by the REM libraries.

Rule 2 A *send message event* is recorded before it is written to the socket, as shown in Figure 5.6. If the DPD records the event after it is sent, it is possible that the sender can be interrupted before the message is recorded. If the receiver records the message, this may lead to a situation where a message seems to be received but not yet sent. This situation cannot occur because if the process is interrupted before the message is recorded, or during the time the message is being recorded, the DPD does not recognize the event. Furthermore, since no write has occurred to the socket no message is sent. If the process is interrupted after the message is recorded, but



Ⓐ Event is recorded
Ⓑ Message is considered to be in the communication channels even though it is not in the socket
Ⓒ Message is written to socket

Figure 5.6: Recording of a *send message event*

before the message is sent, then DPD recognizes the event and displays it to the user. This is a consistent state where the message is considered to be somewhere in the communication channels.

Rule 3 The *create child process event* is recorded after the *child creation acknowledgement* is returned from REM. When creating a child process, the parent process is blocked until the child process is successfully created and connected to REM. Once this occurs, REM sends the *child creation acknowledgement* to the parent process. This event cannot be recorded earlier than the receipt of the acknowledgement, because one cannot guarantee that a child process is created successfully until the child process connects to REM.

Rule 4 The *initialization event* is recorded after the application process has successfully connected with REM. When a successful connection occurs, a *connection acknowledgement* is sent to the process from REM. This event cannot be recorded prior to the receipt of this acknowledgement, because one cannot guarantee a successful connection to REM until the acknowledgement is received.

Rule 5 The *termination event* is recorded prior to the process closing its connection with REM. If the event is recorded after the process closes its connection, the process cannot be restarted, because the communication channel to REM is lost.

An event is not displayed to the user until the following occurs: the event is written to disk by the application process, read by the *Event Collection Process*, and transferred to the UIP. In the case of a child process, the event information has to be sent from the Remote Debugger Process to the User Interface Process. There may be a considerable lag between the time the event occurred and the time it is viewed by the user. In the DPD, this time lag is not a problem because of roll back and replay.

5.3 User Interface

Most widely used sequential debuggers are considered source-level debuggers. By using the source code as a reference for execution, programs can be controlled, system states can be inspected, and breakpoints can be set. Furthermore, there is no need to represent temporal relations for sequential programs, since there is only a single thread of execution. One needs only to have a text-based user interface to be able to present the program's execution, to set breakpoints, and to view the program's system state. Sophisticated sequential debuggers, such as DBXTOOL[AM86], Microsoft's CodeView[Mic87], and Borland's Turbo Debugger[Bor88] make use of *windows* to aid the human debugger; however, the information they represent in the windows is only the program's source code or the textual representation of its current state. The information one wants to present when debugging a distributed program is the temporal relationships among events, global system states, and distributed breakpoints. Displaying a distributed program's execution textually is possible; however the clearest and most understandable method is through graphical representation. A goal of a

distributed debugger, or *any* debugger for that matter, is to aid the human debugger in finding the cause of a bug. A graphical representation of a distributed program's execution gives the user the maximum information with the least complexity while debugging. This section describes the graphical interface used by the DPD.

DPD's graphical interface consists of a Command Window and a Process Display Window as shown in Figure 5.7. The Command Window is where the user controls the DPD and the application processes. The Process Display Window is responsible for presenting the application program's global system state graphically. All input is through *menus, browsers*, and *slide-bars*. A menu is a static list of items from which the user can select one item. Browsers are lists of items that can change over time from which the user can select one or multiple items. Slide-bars allow the user to adjust quantities.

5.3.1 Process Display Window

In the Process Display Window, application processes are represented as vertical lines, events are represented as bullets (•) and process communication is represented as diagonal lines connecting two events between processes. No direction of communication is indicated by the process communication lines; however, one can always tell which process is the sender and which is the receiver because a *send event* always occurs before its corresponding *receive event*. Temporal relationships are drawn using the virtual time of events. Each process starts with a virtual time of zero and has its virtual clock incremented according to the rules in [Lam78]. If the user clicks on an event bullet, a window pops up with a description of the event. This window includes in its display the virtual time for the event and the event type. If the event is a *receive message* or a *send message*, the contents of the message is displayed in

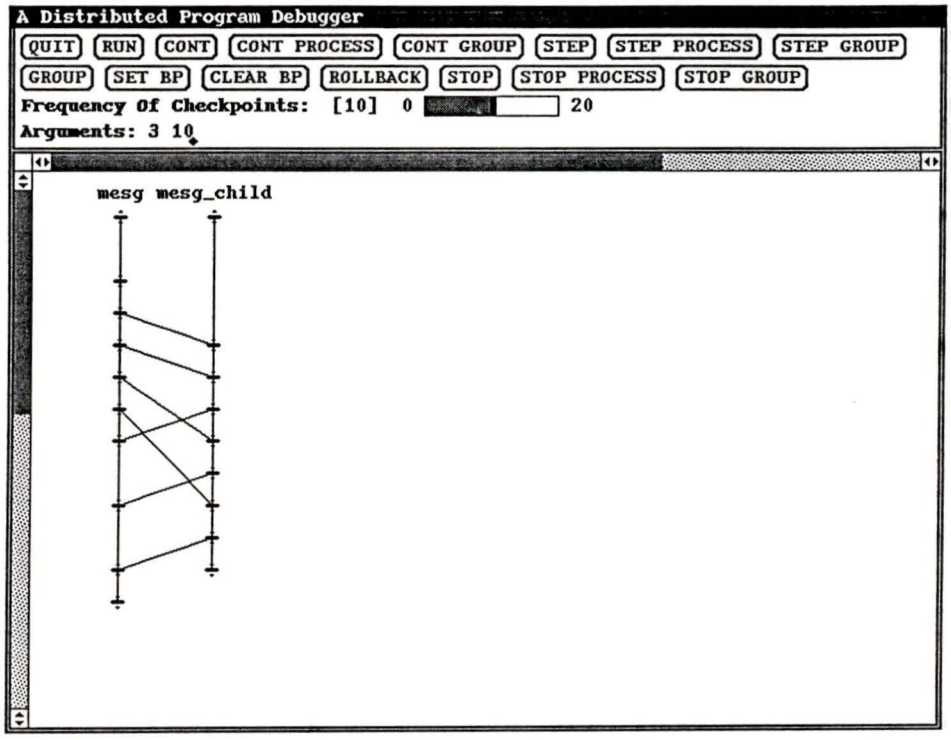


Figure 5.7: The Command and Process Display Window

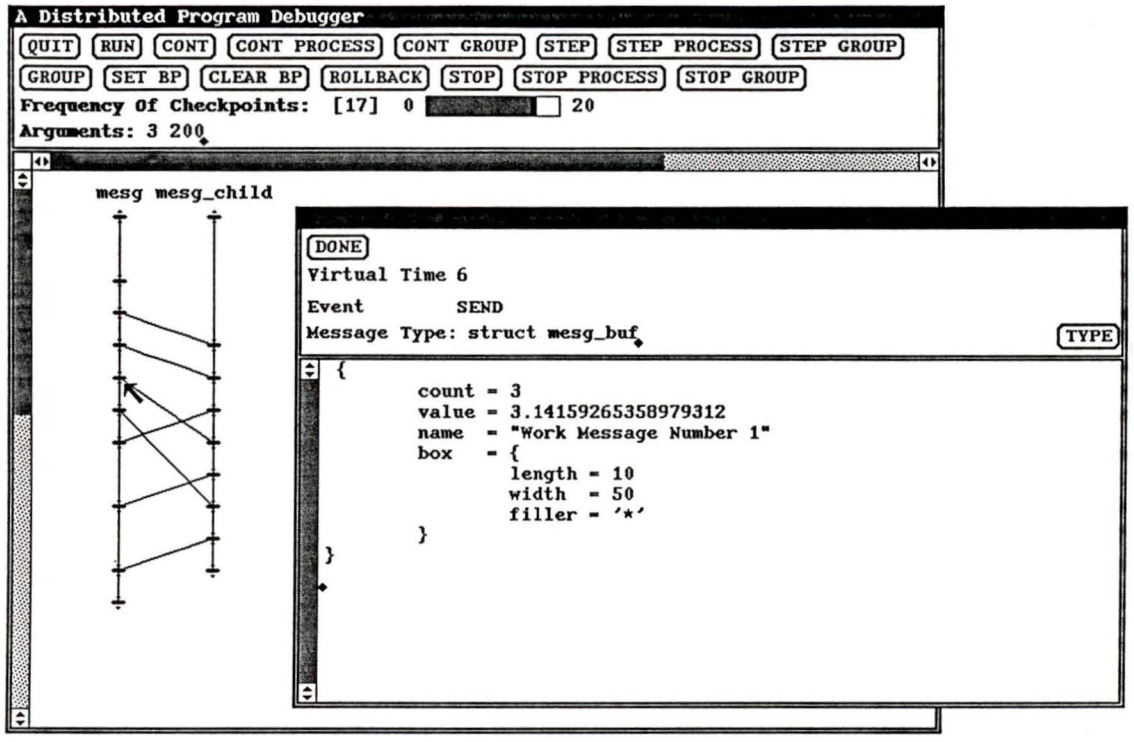


Figure 5.8: The display of a complex data structure

a character string format. Since the message is more likely to be a data type such as an integer, a floating-point number, or even a complex C-language style structure, the user can enter the name of the data type and the message is displayed in a form that represents that data type. See Figure 5.8 for an example showing a complex C-language structure.

5.3.2 Command Window

All DPD commands are entered through the interface supplied by the Command Window. This window consists of a menu from which the user can select commands

by the *point-and-click* of a mouse. This section itemizes the commands and state their effects.

Argument: Before starting a distributed program under the DPD, the user may want to pass arguments to the parent process through the command line. The user can enter command line arguments after the **argument** prompt. The entered text is passed to the parent process when it is **run**. This prompt does not operate in the same manner as entering command line arguments from a UNIX shell, since wild cards are not expanded.

Run: This selection invokes an application program. The effects of this command are that all active processes from the previous run are terminated, the Process Display Window is cleared, the UIP and all the RDPs are reset to handle the new run, the parent process is created and loaded into DBX, text entered at the **argument** prompt is passed through to the parent process, and then the program is run.

Stop: This command stops all active processes. Because of the latency in stopping a distributed program, the Process Display Window may change after this command is issued. This behavior is caused because there may be messages in the communication channel prior to this command being issued.

Stop Process: This command allows the user to stop individual processes. Once selected, a browser of all processes is displayed and the user selects which process to stop. Inactive processes are shown dimmed to signify that the process is currently stopped and is not selectable.

Cont: This command resumes all stopped processes.

Cont Process: This command allows the user to resume individual processes. Once selected, a browser of all processes is displayed and the user selects which process to resume. Active processes are shown dimmed to signify that the process is currently running and is not selectable.

Step: This command allows the user to step the entire distributed program. If there are active processes, they are stopped since a stepping breakpoint must be set in them. Once all processes are stopped, the command to step is sent to all processes. When the first process reaches its next event, all the other processes are stopped.

Step Process: This command steps an individual process. If the process is active then it is stopped so that a stepping breakpoint can be set in it. As soon as the process reaches its next event, it is stopped.

Group: This command allows the user to create sets of processes so that they can be controlled collectively thus allowing the user to abstract processes. When this command is selected, a browser of all processes is displayed from which the user selects which processes are to be in the new group. The user then names the group. A process is only allowed to be in one group. If a process is already in another group, it is dimmed in the browser to signify that it is already grouped.

Stop Group: This command stops all the processes in a named group. Upon issuing this command, a browser of groups is displayed from which the user selects one. All processes in the selected group are then stopped. It may be the case that some processes in the selected group are already stopped. These processes are not affected.

Cont Group: This command resumes all the processes in a named group. A browser of groups is displayed from which the user selects one. All processes in the selected group are resumed. It may be the case that some processes in the selected group are already active. These processes are not affected.

Step Group: This command steps all the processes in a named group. A browser of groups is displayed from which the user selects one. If there are active processes in the selected group, they are stopped so that a stepping breakpoint can be set in them. When the first process reaches its next event, all the other processes in the group are stopped.

Set Breakpoint: This command allows the user to set breakpoints in individual processes or globally for all processes in the distributed program. Upon selecting this command, a browser consisting of **global breakpoints** and the names of all the processes is displayed as shown in Figure 5.9. If the user selects **global breakpoints** then a menu of events is displayed from which the user selects one. A breakpoint is then set for that event in all the processes of the application program. If the user selects a process, the menu of events is once again displayed. If the selected process is active, it needs to be stopped prior to the breakpoint being set. The process remains stopped after the breakpoint is set. A breakpoint cannot be set in an individual process until it is created. Global breakpoints, on the other hand, are inherited by newly created processes. This means a global breakpoint can be set prior to the execution of the program and all of the processes for that program inherit that breakpoint.

Clear Breakpoint: This command allows the user to clear a breakpoint. Upon selecting this command, a browser consisting of **clear global breakpoints** and the names of

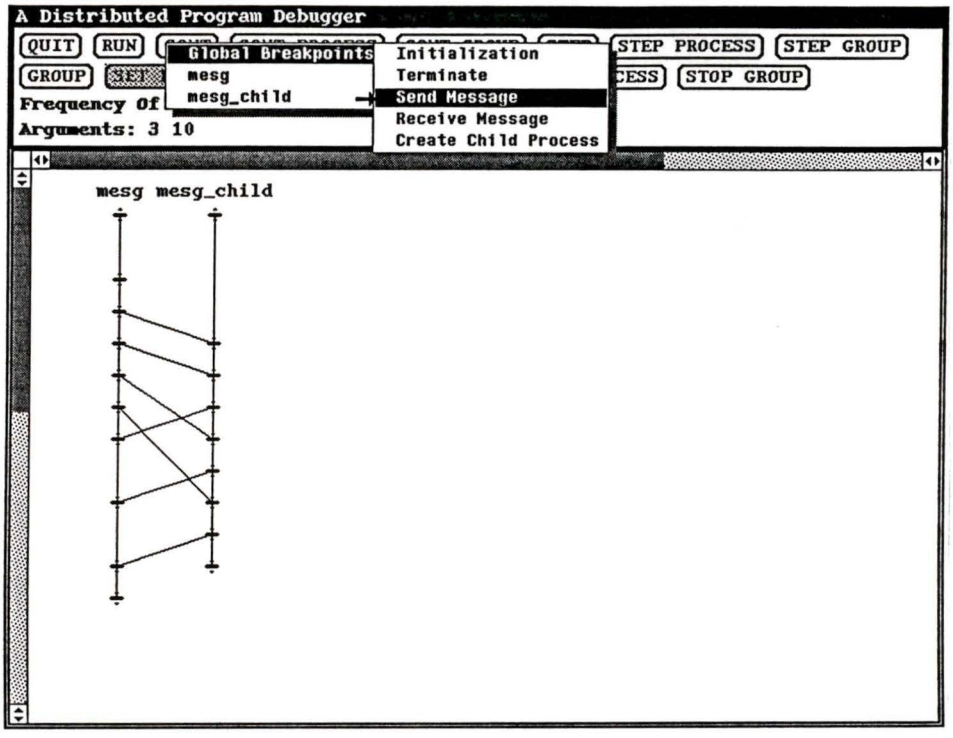


Figure 5.9: Sequential Breakpoint Selection

all the processes is displayed. If the user selects **clear global breakpoints**, a menu of events is displayed from which the user selects one. That breakpoint is then cleared from all the processes of the application program. If the user selects a process, the menu of events is once again displayed. If the selected process is active, it needs to be stopped prior to the breakpoint being cleared. The process remains stopped after the breakpoint is cleared. A global breakpoint can only be cleared by selecting **clear global breakpoints** and an individual process breakpoint can only be cleared by selecting that process.

Checkpoint Frequency: This command consists of a slide bar that allows the user to change the frequency of checkpoints. The checkpointing interval initially defaults to zero, meaning no checkpoints occur during the run. It is under user control to set the checkpointing interval to a non-zero number if checkpoints are desired. The number selected at this prompt represents the number of receive events between checkpoints. It is recommended that the checkpointing interval be set to a large number until the program reaches the point where unexpected behavior occurs. The checkpointing interval should then be decreased so that more checkpoints are produced. This allows greater control over a process at the point when unexpected behavior occurs.

Rollback: This command causes the program to be rolled back to a previous state. Upon selecting this command, the entire program is stopped. The user then selects from the Process Display Window the event to roll back to. The processes are then rolled back to the selected event. It should be noted that the processes are in fact rolled back to the greatest consistent global system state prior to the selected event. The program is then stepped forward until the selected event is reached. If the checkpointing frequency is low, it may take considerable time before the desired event

is correctly rolled back to.

5.4 Process Control

The heart of a distributed debugger is process control. Distributed debuggers are required to control multiple remote processes. This is one of the fundamental reasons why the development of a distributed debugger is more complex than the development of a sequential debugger. The concept of execution, stopping, stepping, and system states differ from sequential programs.

This section describes the process control mechanisms used in the DPD. First, we discuss how an existing sequential debugger is used to ease the burden of sequential process control. Next, we discuss how process initialization is handled. A discussion follows on how distributed programs are stopped and stepped by the DPD. The use of sequential breakpoints is discussed next. Finally, a discussion on the checkpointing mechanism is presented.

5.4.1 Interface to DBX

A distributed debugger needs the facilities for individual process control. This facility can be implemented from scratch; however, we chose to use an existing tool to perform this task. DBX was chosen to be the underlying sequential debugger. DBX, as supplied by Sun Microsystems, is well suited for being an underlying debugger, since it already has the built-in mechanisms to be controlled by another process. In Sun's case they developed DBXTOOL[AM86] which is a graphical front end to DBX.

The virtues of using a debugger such as DBX are many. First and foremost it provides a clean and well-tested user interface. Second, DBX works and there are

no major bugs to work around. Third, DBX provides powerful sequential process control with sequential breakpoints. Finally, DBX supplies a mechanism for displaying complex data structures in an easy to read format.

Communication with DBX is through a pipe and its standard input.[3] To initialize DBX to be controlled by another process, one sends it special command line arguments. These arguments are `-P` followed by a file descriptor. The `-P` tells DBX that it is going to be controlled by another process, and the file descriptor tells DBX where to output its control codes. The control codes produced by DBX are described in [Muc88]. DBX is controlled by writing to its standard input. The input format is the same as if entered by a user from a terminal. DBX also writes output to its standard output, however, this output is not needed by the DPD and since the user should not know that DBX is the underlying debugger this output is redirected to UNIX's bit bucket (i.e., `/dev/null`).

5.4.2 Initialization

Application process initiation for distributed debuggers is more complex than for sequential debuggers since mechanisms to collect events and to create checkpoints must be initiated. This section describes the steps involved in preparing an application process to be debugged.

Step One: Load DBX and the application process into memory. This step consists of creating a DBX process and sending it a command to load the application process into memory. If the load is successful, two commands are sent to DBX, one to set a breakpoint at the entry point of the process and another to set a breakpoint at the

[3]standard input can be thought of as UNIX's term for terminal base input for a program.

exit point of the roll back mechanism. The reason for this second breakpoint is that when the process is rolled back it needs to be stopped immediately after the restore takes place or else the process continues to run.

Step Two: Start the application process running. This consists of sending the command line arguments to the application process and issuing the DBX command to run the process. The application process immediately stops since it reaches the entry point breakpoint as set in step one.

Step Three: Query the application process for its UNIX process identifier. This is performed by sending a command to DBX that causes the application process to execute a `getpid` system call. The process identifier returned by `getpid` is, unfortunately, not sent through the pipe used by DBX to communicate with DPD, rather it is written to a disk file. By reading this file, the identifier is available to DPD. This identifier is required by DPD so that it can send signals directly to the process.

Step Four: Initialize the checkpointing algorithm for the application process. To perform this, a command is issued that writes the name of the application process and a path to its executable image to an area of memory readable by the checkpointing algorithm. The checkpointing interval is also set at this time. Initializing the checkpointing interval is performed by setting a variable in the application process's memory which is again readable by the checkpointing algorithm.

Step Five: Create the *Event Collection Process* so that the application process's events can be collected. After this step, the application process is ready to be run.

These steps make the preparation for an application process time consuming. The

time to start an application process under DPD control is tabulated and described in Chapter 6. It is obvious that initializing a process under DPD takes at least as long as to initialize the process under DBX and then some for DPD overhead. Performance can be improved if one were to implement the facilities of DBX within DPD, however, the effort to implement such a facility is beyond the scope of this thesis.

5.4.3 Stopping

When the user issues the command to stop an application program, DPD cannot possibly stop all of the application processes instantaneously. This is due to the communication delays involved in sending the stop command to all the remote processors. The best one can expect is to stop the application program *as soon as possible.* Timing results for stopping a distributed program by DPD are given in Section 6.2.3. Maintaining a consistent system state is another problem that complicates stopping a distributed program. If a program is stopped in a state such that a received message is recorded but its corresponding send message is not, the program is in an inconsistent state. This section describes how processes are stopped and how consistent system states are maintained by the DPD.

DPD stops a distributed program in a relatively simple manner. When the stop command is issued, a *stop child* message is sent to the necessary RDP for every child application process. This could mean that more than one *stop child* message may arrive at the same RDP. The parent process is stopped only after the messages to all the child processes are sent. To physically stop a process, DPD uses the `kill` system call with the `STOP` signal. The inefficiency of stopping a distributed program in this manner is the sending of almost the identical message to every RDP. A more efficient means would be to broadcast a single message that is received by all the

RDPs; unfortunately, the REM prototype does not currently include this facility. It will, however, be added in the near future[Liu90].

Stopping a distributed program in this simple manner does maintain a consistent global system state since at no time can a process receive a message that is not sent. The reason why an inconsistent state cannot occur is because of the manner in which events are recorded. This is described in Section 5.2. DPD may display an inconsistent system state for a short period of time; however, it does in time display the program's consistent system state. For example, it is possible that the UIP receives a *receive message event* before its corresponding *send message event* is received by the UIP. Since the *receive message event* is recorded, its corresponding *send message event* is also recorded, as discussed in Rules 1 and 2 in Section 5.2. Since the *send message event* is recorded it arrives at the UIP in some finite time and, thus, a consistent system state is displayed.

DPD can also stop an individual process while keeping the remaining processes active. Stopping a process can affect the execution of a distributed program. In the REM paradigm, if a process is stopped and the other active processes continue to send messages to it, the processes that send messages to it never blocks because of the queues implemented within REM. If a process attempts to receive a message from a stopped process then this receiving process blocks unless this message is previously sent to it by the stopped process. A blocked process attempting to receive from a stopped process remains blocked until the stopped process is resumed and sends a message to it.

Stopping a process under UNIX does have its difficulties. One problem is that the REM libraries make use of the `select` system call. This system call is not restartable and, thus, if interrupted by a signal, it causes an error to be raised. This error causes routines within the REM libraries to fail. To prevent `select` from raising errors, all

signals[4] are masked before the `select` call is issued. This prevents `select` from ever being interrupted by a signal. Since this prevents signals from being handled by the process, the process is now unstoppable and, thus, uncontrollable by DBX and DPD. To prevent this from occurring, `select` is initialized to return after a timer elapses regardless of there being something available at a socket. Once `select` returns, all signals are unmasked. A check is then performed to find the reason why `select` returns. If it returns because something is at a socket, the process continues normally. If `select` returns due to an elapsed timer, the timer is reset, the signals are masked and the `select` routine is called again. The processing overhead involved in the loop when the timer elapses is negligible. It is during the time in which the signals are unmasked that the stop signal can be handled. In this way, `select` does not cause the REM libraries to fail and the process can be stopped even if it has called `select`. If the process is in the `select` system call, there may be a delay before a stop signal is handled. This delay does not affect the overall system state, since the process is already blocked and thus its state does not change.

5.4.4 Stepping

When debugging a sequential program, stepping means to execute one statement at a time. This statement may be one machine instruction or one source code statement depending on the functionality of the debugger being used. For distributed debuggers, events are stepped instead of machine instructions or source statements. To complicate matters further, the time it takes to reach the next event may be undefined because of the dependencies among processes. This section defines stepping under DPD and describes how stepping is implemented.

[4]Technically, not all signals are maskable. The unmaskable signals are used to forcefully terminate a process.

To step a single process of a distributed program requires the process to be run until its next event is recorded. This requirement specifies the granularity of stepping a distributed process. When stepping a sequential process, the time duration of a step is small, usually the time to execute one or several statements. When stepping a distributed process, the time duration may be large if events are few and computation is intensive. This requirement also states that the process is not considered stepped until its next event is reached and is recorded. This means if the next event is a *receive message event* then the process does not conclude its step until the expected message is received and then recorded. This also implies that the time in which a process finally reaches its stepping point may be dependent on other processes.

We now describe the procedure used to step a process. When the step command is issued by the user, the process is stopped if it is not already stopped. A breakpoint is set in the process at the point immediately following where events are recorded. The process is then made active. The process concludes its step when the breakpoint is reached.

When stepping a distributed program, all processes are made active. The first process to reach its next event causes all the other processes to stop. Furthermore, each process of the distributed program executes only until its next event and no further. When stepping a distributed program, not all processes necessarily reach their next event. A process whose next event is a *receive message event* may not conclude its step because the process which is supposed to send the message may not reach the event from which the message is sent. When stepping a distributed program, more than one process may reach their next events simultaneously and each of these processes attempts to stop the entire program. This situation has to be handled correctly.

If the parent application process is the first process to reach its stepping break-

point, the UIP sends a stop message to all other processes. If a child is the first application process that reaches its stepping breakpoint first the RDP sends a *has stepped* message to the UIP and the UIP then sends a stop message to all other processes. If two or more application processes reach their next event simultaneously, more than one *has stepped* message arrives at the UIP. The UIP serializes these messages and executes only one of them.

5.4.5 Sequential Breakpoints

DPD provides the facilities to set sequential style breakpoints in application processes. Sequential breakpoints operate on a single process and only affect that one process. Most sequential debuggers have the capability to set breakpoints in a sophisticated manner. For example, breakpoints can be set at specific source lines, when functions are invoked, and when memory changes. DPD's sequential breakpoints can only be set at events. This section describes DPD's implementation of sequential breakpoints.

Sequential breakpoints are implemented using the breakpoint facilities supplied by DBX. Each event has a special breakpoint function associated with it. These breakpoint functions are located within the *Event Record Module*. This implies that breakpoints can only be set at events recognized by DPD. The calls to the breakpoint functions are located after the point in which events are recorded. When the user sets a breakpoint for an event, a DBX breakpoint is set in that event's corresponding breakpoint function. For events such as receive message, create child, and initialize process, it does not matter when the breakpoint is reached. But for the send message event and terminate process event, it does matter when the breakpoint is reached. For the send message event, the breakpoint must be reached before the message is written to the socket. This is to ensure that the receiving process of the message does not

progress beyond the point in which it tries to receive the message. For the terminate event, the breakpoint must be reached before the process closes its connection with REM so that all necessary connections to REM remain open.

Breakpoints can be set individually for a single process or globally for the entire program. An individual breakpoint can only be set for a specific process and only after the process is created. Global breakpoints are inherited by all newly created processes and, thus, global breakpoints can be set prior to execution. Global breakpoints and individual breakpoints are maintained separately. A global breakpoint can only be removed by the **clear global breakpoint** command and an individual breakpoint can only be removed for that process.

Global breakpoints are useful when one wants to stop a program in a specific state. For example, suppose a global breakpoint is set for the initialize process event. When run, the parent process reaches its initialize process breakpoint and stops. If the parent is resumed and allowed to create all of its child processes, then each one of the child processes stops as soon as they are created. Once all the child processes are created, the parent process can be stopped by the user. In this state the program has all of its child processes created, but they do not execute beyond connecting to REM. At this time, the user can resume all the processes simultaneously or the user can step an individual process to analyze its behavior. Another useful place to set a global breakpoint is at the terminate process event. This allows all the processes to be run at full speed, but stop before they terminate. Once every process has stopped, the program can be rolled back and different event sequences can be analyzed.

5.4.6 Distributed Breakpoints

The framework for distributed breakpoints is described by Haban and Wiegel[HW88] and by Miller and Choi[MC88]. Sequential breakpoints deal with one process and one time reference. Distributed breakpoints deal with multiple processes and multiple time references. Distributed breakpoints, for example, can be set to stop a distributed program when one event happens before another event, when two events occur simultaneously, or when a single event is reached out of a set of events.

Distributed breakpoints are not currently implemented in DPD. The definitions in [HW88] and [MC88] along with the algorithm described by Fowler and Zwaenepoel[FZ89] provide the basis to which distributed breakpoints can be facilitated in DPD.

5.4.7 Checkpoints

There are two common methods to checkpoint a process under UNIX. The first method simply `forks`[5] a new process thus creating an in-memory checkpoint of the original process. The forked process is blocked and the original is resumed. To replay, the blocked process is simply allowed to continue. This method is used in [PL89]. The second method writes the information that needs to be checkpointed to a file. To replay, the process's memory must be overwritten with the information that resides in the file. Both methods have problems. The problem with the first one is the high usage of limited system resources, namely main memory. It is conceivable that a large distributed process that produces many events could use up all available memory when checkpointing. The advantages of this method is that it is more faster than the other

[5] `forking` a process means making an exact in memory duplicate of a process. The duplicate process can either continue running or become blocked.

method to produce and restore checkpoints. The problem with the second method is that it is difficult to restore a process to the exact state in which it is checkpointed. For example, if a file that is open at the time of a checkpoint is later closed, that file has to be reopened during the restore. Another problem with this method is that checkpointing is slow since it has to write to a file. This method is used in [FB89]; however, they have optimized it by only saving dirty pages when checkpointing. For DPD, we chose the second method because of the limited resources available.

The checkpointing algorithm is derived from one developed by Curry[Cur88]. Curry's checkpointing algorithm was originally developed for a Digital Equipment Corporation's VAX computer. This meant that the format used to make system calls had to be changed to be compatible with the format used by Sun Microsystem computers. Furthermore, since the architecture of the computers differ, the method in which the registers and the heap are written to disk also changed.

The algorithm is comprised of two parts. The first is a library of modified system calls. Certain systems calls such as `open`, `close`, and `socket` have to be monitored so that information about when and how these routines were called can be saved. There are a total of 15 system calls being monitored. As an example of what information needs to be collected when monitoring these system calls, we examine the `open` system call. For this system call, we need to save the name of the file being opened, the flags and the mode used in opening the file, the file's current position, the number of locks on the file, and the file type. Saving the name of the file is important since there is no easy way under UNIX to get a file name from a file descriptor.

The second part to the checkpointing algorithm is the code to produce and restore checkpoints. To produce a checkpoint means saving information on all open files, saving into the heap the program's limits, timers, priorities, signal stack, user and group identifiers, and finally writing to disk the heap, stack, and registers. Restoring

a checkpoint is almost the reverse of saving a checkpoint. The only major difference is that all files which are open during the time of the roll back are closed and only those files that are open during the checkpoint are reopened.

There are serious limitations to this checkpointing algorithm. First, there is no way to checkpoint the contents of files. This means that if a file changes between two checkpoints and the program is rolled back to the first checkpoint, then the file contains the information written at the second checkpoint. This can have a devastating affect on a program especially if the file contains a random access database. It is possible to checkpoint files but this is beyond the scope of this thesis. The second limitation is that the checkpointing algorithm cannot monitor or understand all calls to the kernel. Even though the algorithm monitors system calls, the state of the kernel can be changed by the data passed through these system calls. For example, suppose one opens a device and between checkpoints the state of this device changes. The checkpointing algorithm cannot revert this device to the state in which the checkpoint occurred.

When restoring a checkpoint, one cannot close the socket file descriptor that is used to connect with REM. If the socket is closed, REM shuts down the connection to that process and removes that process from REM's internal tables. Thus, when restoring a checkpoint, the physical connection to REM remains open. To implement this, an interface routine that prevents the algorithm from closing file descriptors was added to the checkpoint module. Not closing the socket does not affect the communication between REM and the process as long as the socket is flushed to remove any pending messages. Socket flushing is described in Section 4.1.3.

It should be noted that checkpointing is very system dependent, especially since the registers of the processes have to be saved. The current checkpointing algorithm is, however, written entirely in the C Programming Language. The current imple-

mentation is not portable to a segmented architecture, for example some of the 80x86 family of processors, since the implementation expects a linear addressing scheme. To save and restore registers, we made use of the `setjmp` and `longjmp` system routines. This is the most portable method found to save registers. The only non-portable aspect of `setjmp` and `longjmp` is that there is no specific order in which the registers are maintained. This causes problems because the top of stack needs to be retrieved and set using this method and its position in the `setjmp` and `longjmp` data structure is not fixed.

5.5 Roll back and Replay

In this section we describe the roll back and replay mechanism used in DPD. We used the recovery algorithm described by Johnson and Zwaenepoel[JZ88] to implement our roll back feature. Their algorithm was designed to provide the facilities for fault tolerance in a distributed system using checkpoints and optimistic message logging. If a processor fails, a distributed program can be recovered from its checkpoints. If a checkpoint for a process does not form part of the recoverable system state, logged messages are used to roll forward that process to its recoverable system state.

We have enhanced the algorithm so that it can be used by DPD for its roll back feature. The algorithm is used by DPD because it incrementally finds recoverable system states and because it does not suffer from the domino effect. In this section we present the background for Johnson and Zwaenepoel's algorithm and our roll back and replay implementation.

5.5.1 Algorithm Background

Every time a process receives a message it begins a new *state interval.* Each process of the distributed system maintains a *state interval index* which is a count of the number of received messages. When a process receives a message from another process, it becomes *dependent* on the state of that other process. Each process maintains a *dependency vector* which contains the maximum state intervals of the other processes it depends on. This means that when a process sends a message, it must include in the message its current state interval. If a process has not received a message from another process, that process' element of the dependency vector is denoted by $\perp$ which has a value less than any possible state interval. A *system state* is represented by an $n \times n$ *dependency matrix* where row i is the dependency vector of process i. The diagonal of this matrix contains the current state interval index of each process. A dependency matrix describes a consistent system state if no element of a column is larger than the diagonal element of that column. Figure 5.10 shows two system states and their dependency matrix.

Since Johnson and Zwaenepoel describe an algorithm to recover from the failure of a distributed system, logged messages and checkpoints are written to stable storage. We are only interested in debugging programs not systems and assume a hardware fault free environment so logging messages and writing checkpoints to stable storage is not needed. Due to our pessimistic message logging their definition of a *recoverable system state* can be relaxed. By their definition, a system state is recoverable if and only if all component process states are stable and the resulting system state is consistent. We can relax this constraint and define a system state to be recoverable if the resulting system state is consistent. This means all consistent system states are recoverable. They also maintain the *current recovery state* which is the largest recoverable system state. The current recovery state is maintained incrementally and

$$A = \begin{bmatrix} 2 & 0 & \perp & \perp \\ \perp & 0 & \perp & \perp \\ \perp & 0 & 2 & 0 \\ \perp & \perp & \perp & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & \perp & \perp \\ \perp & 0 & \perp & \perp \\ \perp & 0 & 2 & 0 \\ \perp & \perp & 2 & 1 \end{bmatrix}$$

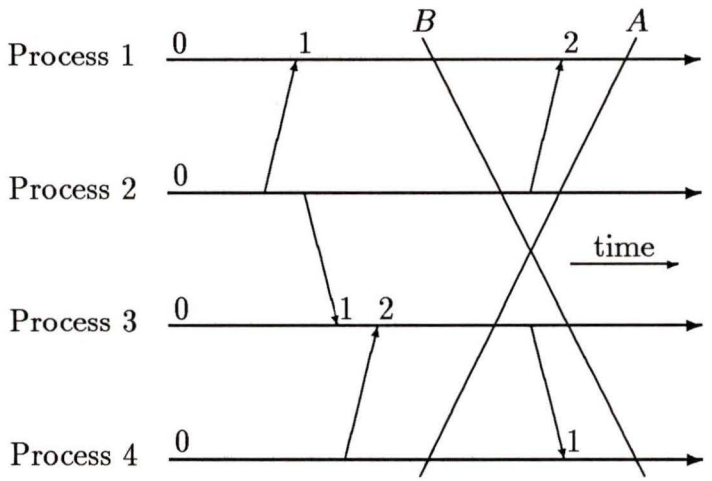


Figure 5.10: Two system states and their dependency matrices

thus can never decrease. This prevents the domino effect.

There are other features to this recovery algorithm. Processes are assumed to be deterministic between received messages. No coordination is required between the checkpoints of different processes. Checkpoints need only occur after a message is received. Since the state interval index for the process only changes after a message is received. Every process is required to checkpoint immediately after it is created so that there is at least one checkpoint in which to roll back to.

5.5.2 Implementation of Algorithm

We have implemented a modified version of Johnson and Zwaenepoel's recovery algorithm. The algorithm resides in the User Interface Process. As described in Section 5.1.3, the Event Collection Processes sends all event information for an application process to the User Interface Process. Every time a *receive message event* arrives at the UIP, it increments that application process's state interval index and then the recovery algorithm is invoked to determine if there is a current recovery state. If there is one, it is saved in a list of recovery states. When an application process checkpoints, the UIP records the process's state interval and the file to which the checkpoint is written.

To roll back the application program, the user selects a process and an event to roll back to. The state interval containing that event is found first and then a recoverable system state for the event's state interval is found next. The recoverable system state is the one that contains a state interval which is less than or equal to the state interval of the selected event. Finally, the checkpoints used to roll back each process with are found and are the ones that occurred at or prior to the state for that process in the recoverable system state. These checkpoints are called the *effective checkpoints* for

the processes.

When the checkpoints are found, each process is then restored to their effective checkpoint. It is likely that the state interval in which the checkpoint occurred for a process are not the same state interval as its recoverable system state interval. This means that the process may have to be run forward until the recoverable system state interval is reached. If the process is run forward, any message that it sends is discarded. If the process expects to receive messages, those messages must be retrieved from the message log and sent to the process. If the process attempts to create a child process and that child process is still alive, that child process is not recreated and REM's *create child* interface routine returns the virtual process identifier of that active child process. If the selected event is not a *receive message event*, the selected process is run forward until the selected event is reached. If there are only *send message events* and *create process events* up to the selected event, the processes can be run forward to the selected event. If there is a *receive message event* between the recoverable event and the event selected to roll back to, this *receive message event* is not part of a consistent state and, thus, the process has to be stopped before executing this event. Once the selected process is run forward, the system is rolled back.

Chapter 6

Results

This chapter shows the usability and effectiveness of DPD when used to debug a distributed program. The effects of DPD on running distributed applications and some performance figures are also presented.

6.1 Usability

To show the usability of DPD, we have chosen a distributed application that is simple but effectively tests all of the tools supplied by DPD. Through this application, we show how state inspection, process control, and roll back and replay can be used to inspect, change, and analyze the behavior of a program.

6.1.1 PAR

The Positive Acknowledgement with Retransmission protocol (PAR)[Tan81] is our test-bed for DPD usability. The PAR protocol is meant to work in a networked

environment where messages may be lost or damaged. To detect out of sequence messages caused by lost or damaged messages, the sender and receiver make use of a one bit sequence number. The protocol works in the following manner; for every message the sender sends, it embeds its current sequence number in the message. After sending the message, the sender sets a timer and then blocks waiting for an event to occur. If the event is the arrival of an acknowledgement, the sender increments its sequence number and sends a new message. If the event is a time-out, or the arrival of a damaged acknowledgement, the original message is retransmitted. The receiving process simply waits for messages to arrive. If a damaged message arrives, it is discarded. If an undamaged message arrives, the receiver checks its own sequence number against the sequence number embedded in the message. If the two numbers match, the receiver's sequence number is incremented and the message is transferred to its host. The receiver always sends an acknowledgement when an undamaged message arrives even if the message is out of sequence.

The following experiment illustrates a distributed program consisting of two processes that implements the PAR protocol. The parent process of the program was configured to be the sender and the child process was configured to be the receiver. To simulate the arrival of damaged messages, randomly chosen messages were flagged as being damaged. This simulation was necessary because REM presents an error free communication channel. Lost messages cannot be simulated; however, since the protocol behaves identically when messages are lost or damaged, simulating only damaged messages suffices. Time-outs were implemented through UNIX's `setitimer` system call. Both the percentage of damaged messages and the time-out interval are adjustable at run-time. Finally, since the protocol never halts, a small modification was made so that there is an upper bound of the number of successfully acknowledged messages in order to terminate the protocol.

For our tests, the PAR program was set to send 10 messages and, thus, the sender is to receive 10 acknowledgements. It should be noted that it actually sends 11 messages; the first message is an initialization message used for our implementation. The contents of the message is the string, "The quick brown fox jumps over the lazy dog." It is rotated one character after a message is sent. Using this strings and then rotating it one character shows when the protocol fails.

6.1.2 State Inspection

To simulate a faulty program, a bug was introduced that caused the sender to incorrectly increment its sequence number. The effect of this bug caused the receiver to correctly receive, acknowledge, and transfer to its host the first message that arrived. All subsequent messages were not being transferred to the host. To complicate matters further, the sending process was terminating normally. By analyzing the run-time behavior of the program without the aid of a debugger, one was able to deduce that either all of the messages except for the first one were not being received correctly, or that they were somehow being received correctly but were being mishandled by the receiver.

DPD was invoked and the program executed under its control. The behavior of the program looked normal when viewed through the graphical interface. See Figure 6.1 for DPD's graphical representation of this event sequence. The graphical interface showed that the correct send-receive-acknowledge pattern was occurring. This ruled out the possibility that the second and subsequent messages were being lost. By inspecting the message at the receiver showed that the user's part of the message was correct. See Figures 6.2, 6.3, and 6.4 for the displays showing the contents of the user's messages. This probably meant that the receiver was mishandling the

second and subsequent messages. However, closer inspection of the system's part of the message showed the real reason why the protocol was failing. The system's part of the message contains the sequence number and every message had an embedded sequence number of 0 in it. Figures 6.5, 6.6, and 6.7 show the system's part of the messages. This caused the incorrect behavior to occur since only the first message and its embedded sequence number matched with the receiver's sequence number. Because the receiver correctly incremented its sequence number and because the second and subsequent messages always had a sequence number of 0 embedded in them, the receiver never transferred these messages to the host. Furthermore, since the receiver always sends an acknowledgement for every undamaged message that arrives, the sender was always receiving an acknowledgement for every message it sent. This meant that the sender terminated normally when its allotted number of messages were sent.

To fix this problem one had to find out why the embedded sequence numbers were zero and this can only be done by viewing the source code of the program; however, DPD was able to isolate the problem to the sending process. This example showed how, by viewing the program behavior and by inspecting the contents of the messages, a bug in the program can be detected. This example also shows how one of the most commonly used techniques in sequential debuggers, namely state inspection, can also be used in a distributed debugger.

6.1.3 Process Control

We now show how the debugger can be used to control an application program. To show process control, we cause the PAR protocol to fail on its own accord. In this example, we use the notation M_{j_k} to represent a message, where j is the sequence

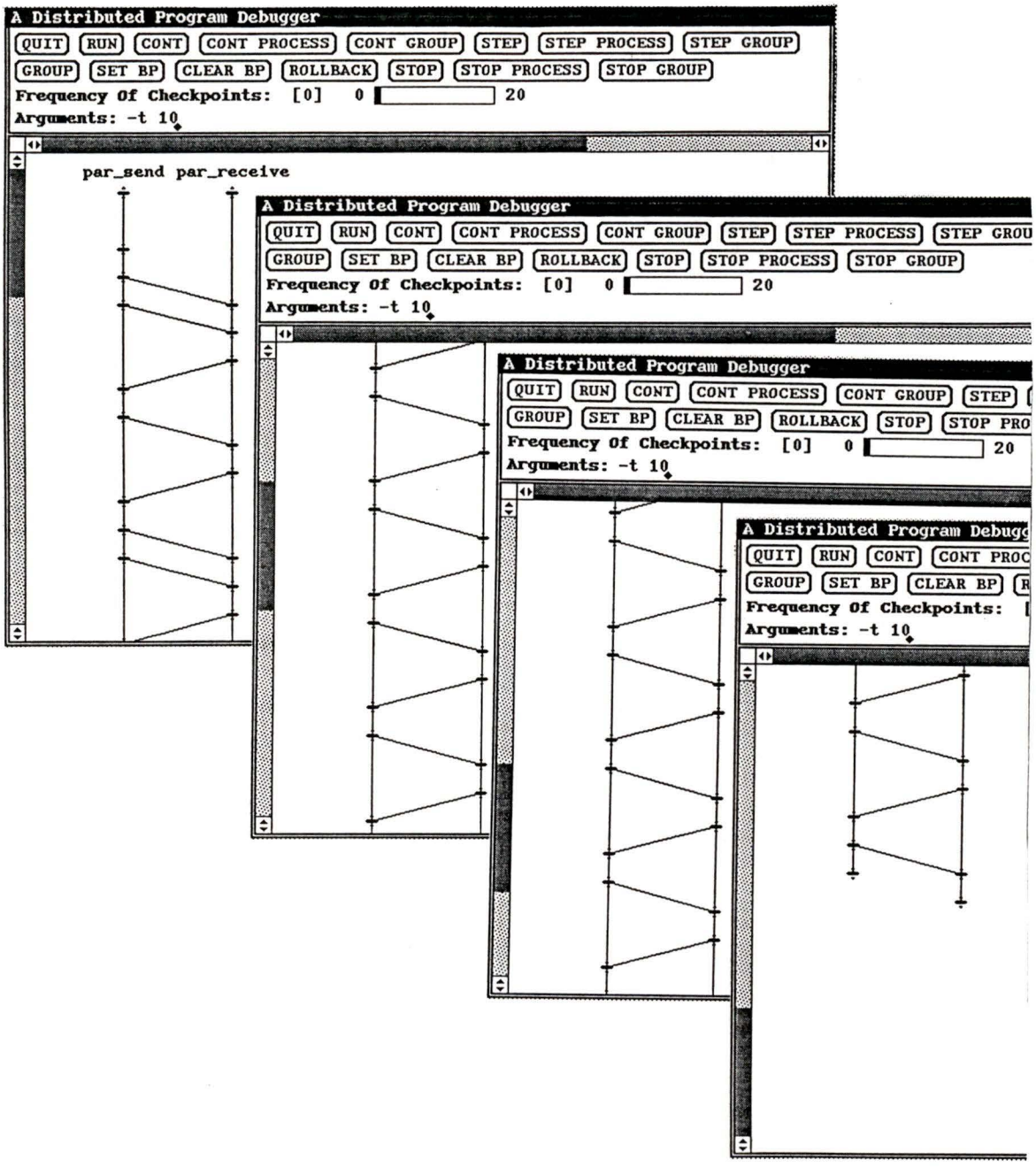


Figure 6.1: The expected behavior of the PAR protocol (Screens 1, 2, 3, and 4)

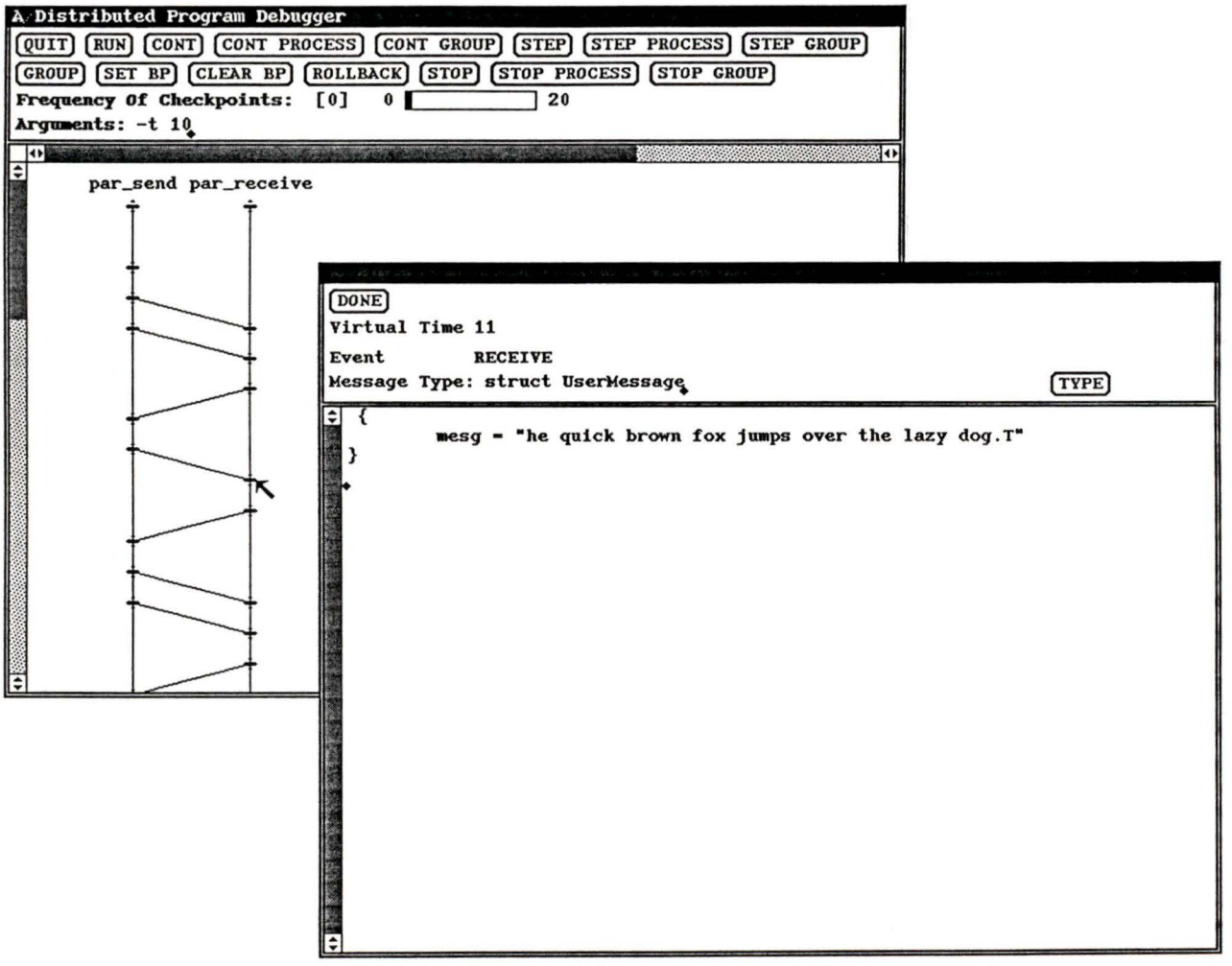


Figure 6.2: The user's message contents (Screen 1)

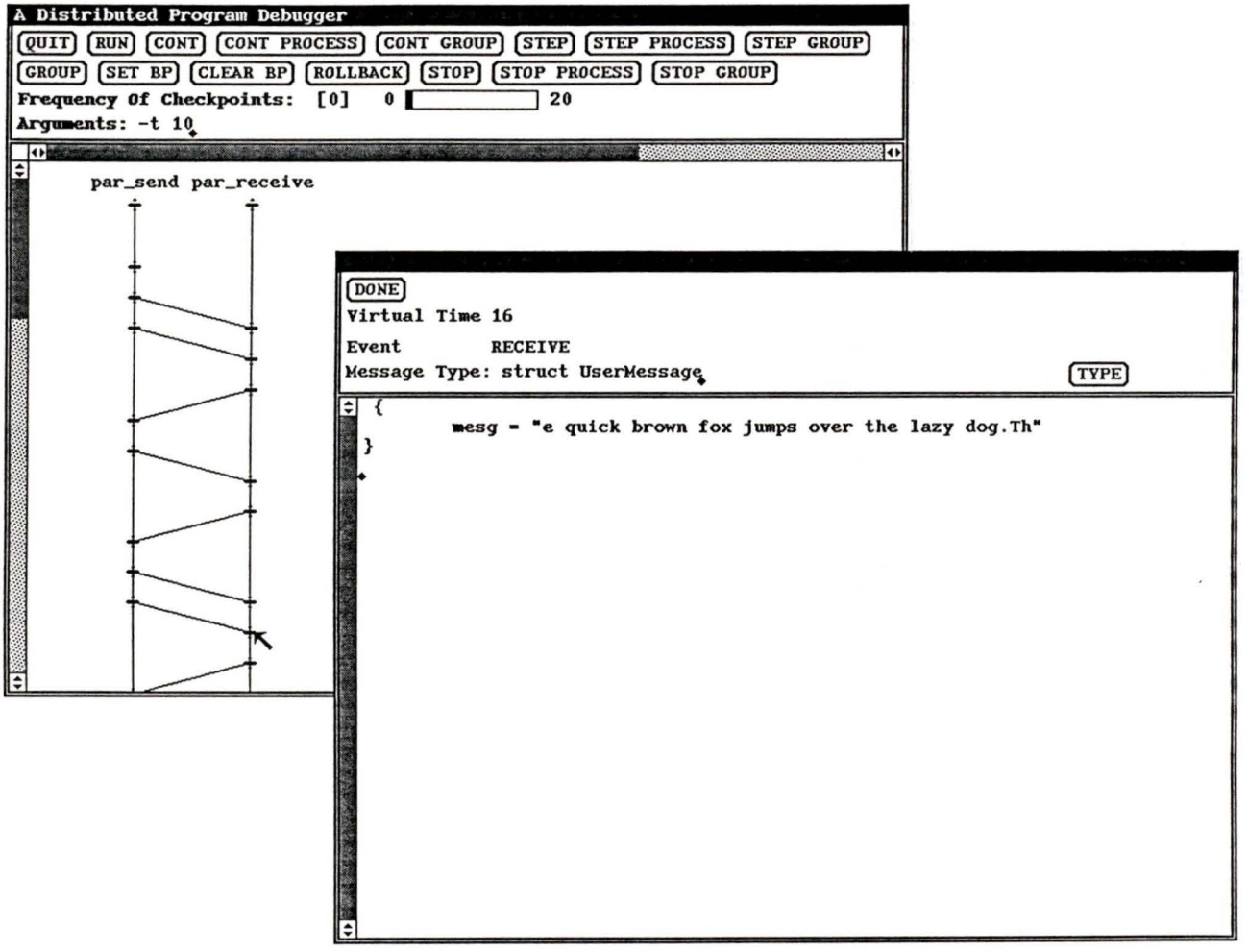


Figure 6.3: The user's message contents (Screen 2)

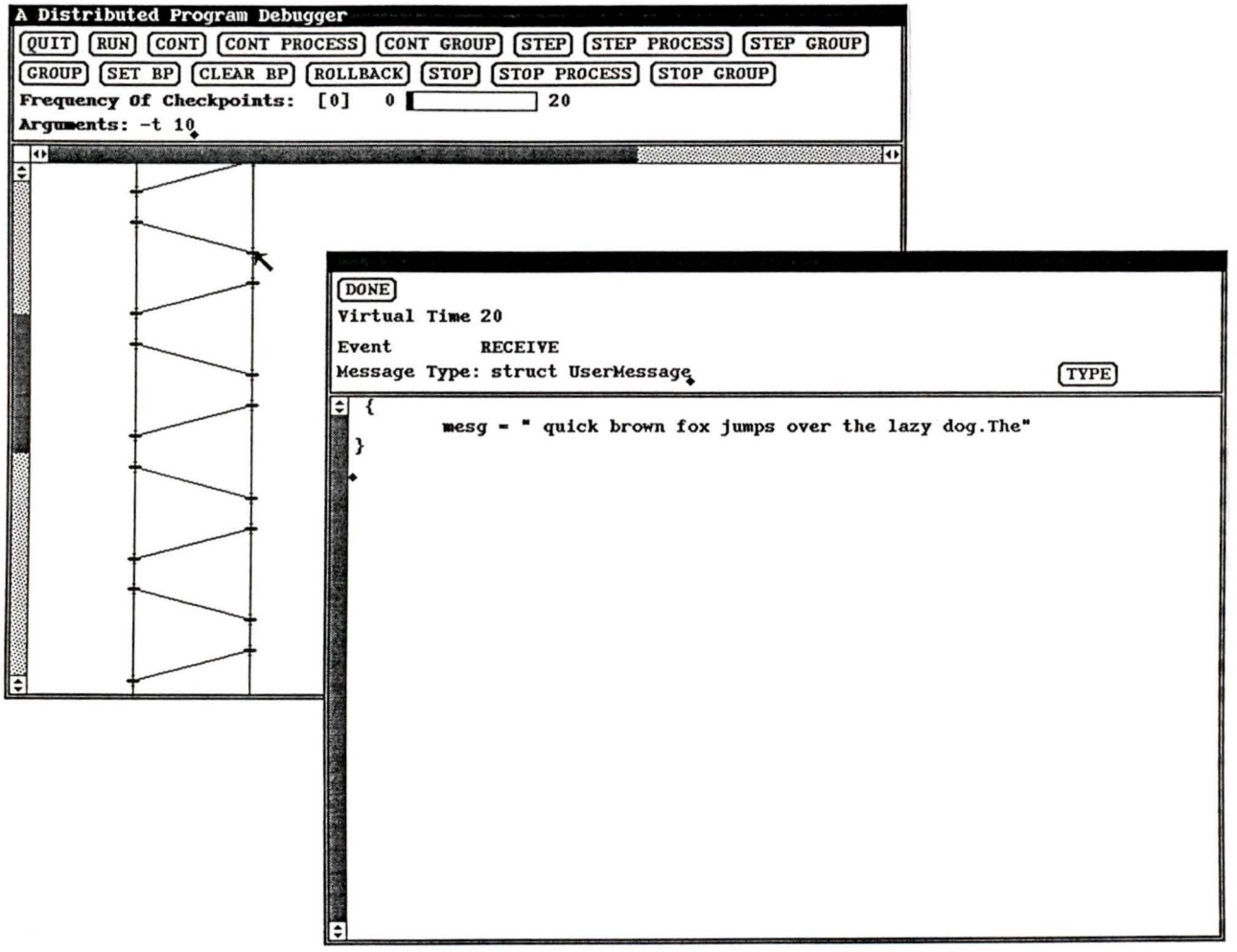


Figure 6.4: The user's message contents (Screen 3)

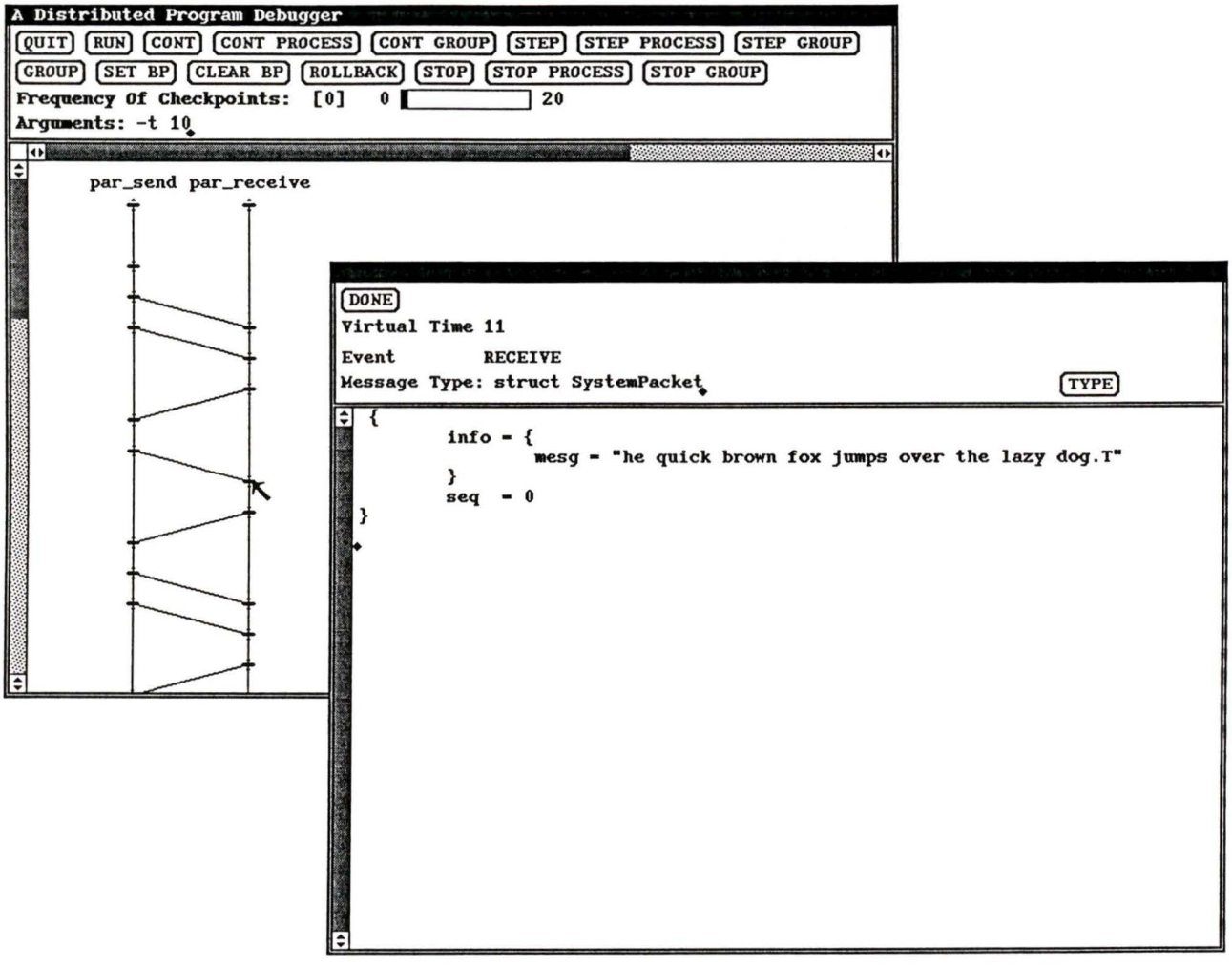


Figure 6.5: PAR's sequence number embedded in the message (Screen 1)

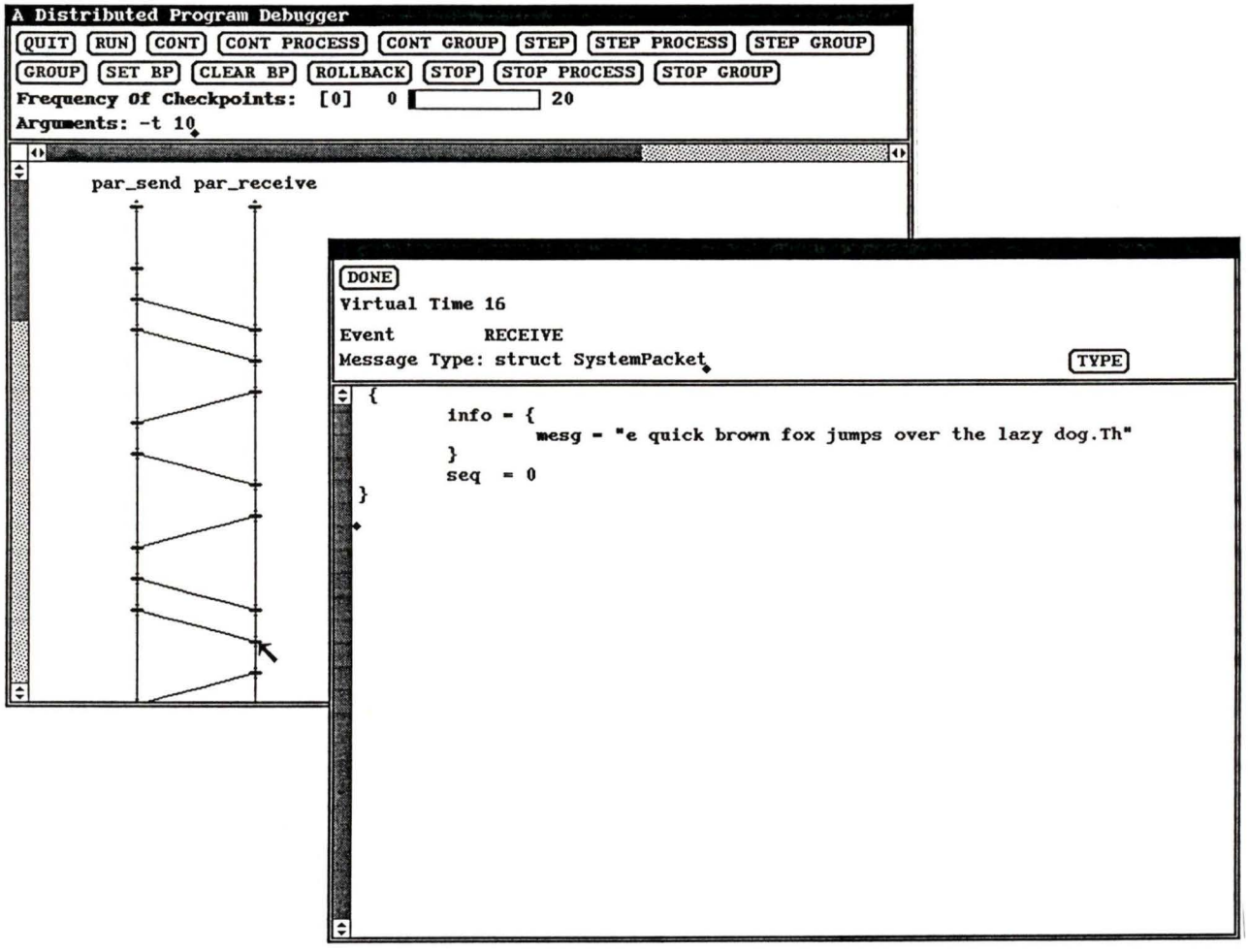


Figure 6.6: PAR's sequence number embedded in the message (Screen 2)

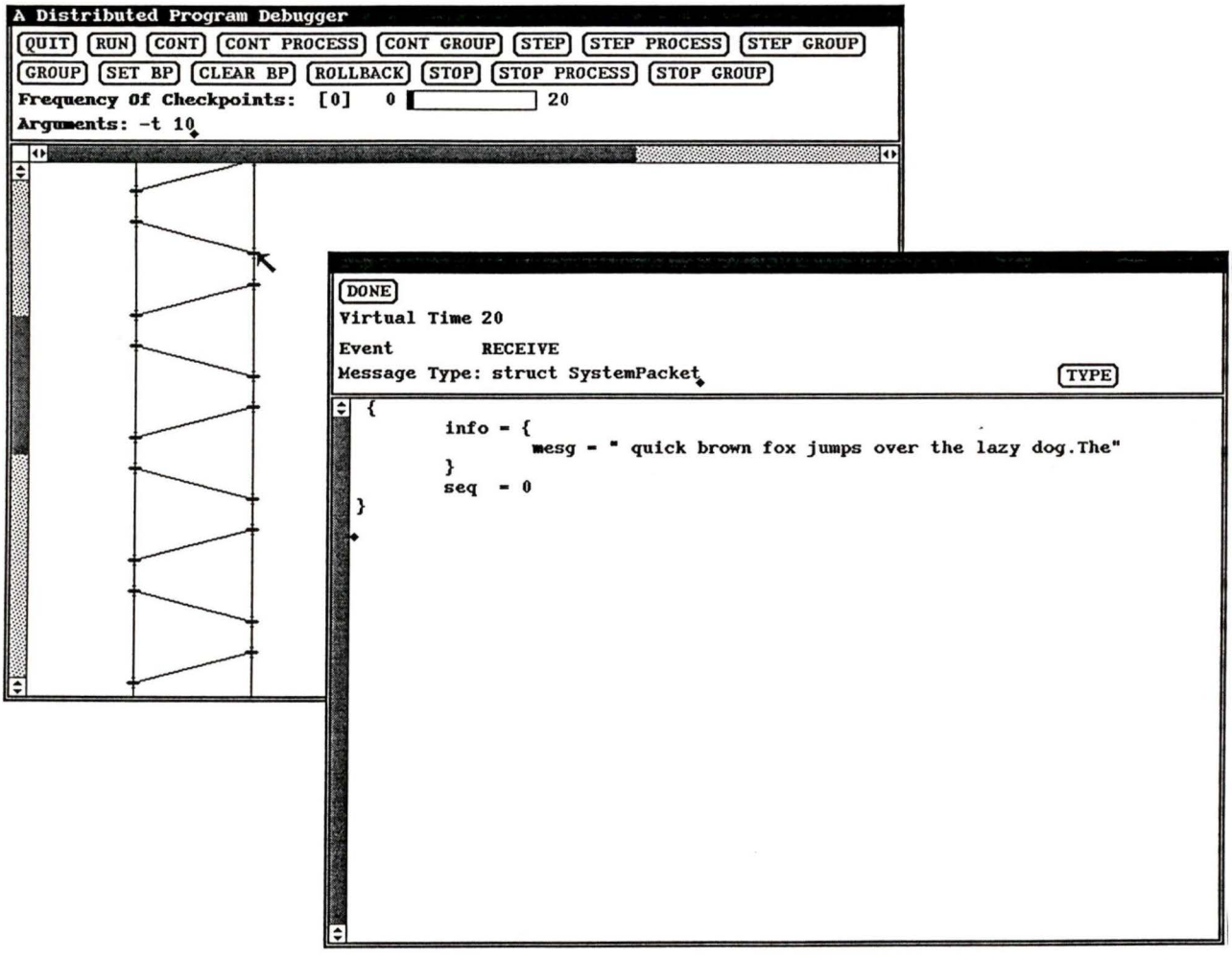


Figure 6.7: PAR's sequence number embedded in the message (Screen 3)

number and k is either the initial (i) or retransmitted (r) message. Since this protocol uses a 1-bit sequence number, j can only be 1 or 0. We use S to denote the sending process and R to denote the receiver.

The protocol fails in the following situation. S starts by sending message M_{0_i}. This message arrives at R and an acknowledgement is sent for it. However, before S receives the acknowledgement it times out and sends M_{0_r}. The acknowledgement for M_{0_i} finally arrives at S but because of the limitations of the protocol, this acknowledgement is mistaken to be the acknowledgement for M_{0_r}. The arrival of this acknowledgement causes S to send M_{1_i}. When M_{0_r} arrives at R, an acknowledgement is also sent for it. Suppose that when M_{1_i} arrives at R it is damaged. This means that there is no acknowledgement sent for it. Meanwhile at S, the acknowledgement meant for M_{0_r} arrives, but because the last message sent by S was M_{1_i}, S thinks this acknowledgement was meant for it. S now sends a new M_{0_i} message. The new M_{0_i} arrives at R, but it is out of sequence, because R did not correctly receive M_{1_i}. This message is not transferred to the host but an acknowledgement is still sent for it. Since S receives this acknowledgement, it sends a new M_{1_i} message. R accepts this new M_{1_i} message since it contains the expected sequence number and transfers this message to its host. From this time forward the algorithm works as expected. The problem is that the second and third message are not transferred to the host. Figure 6.8 shows the expected sequence of messages that are supposed to be transferred to the receiver's host and Figure 6.9 shows the actual sequence of messages transferred.

We now show how this situation can be detected by DPD. To guarantee that the required message always arrives damaged, a modification to the code was made so that the first M_1 message arrives damaged. It should be stressed that we are trying to show process control. There are no induced bugs in the protocol and that

Message 1: The quick brown fox jumps over the lazy dog.
Message 2: he quick brown fox jumps over the lazy dog.T
Message 3: e quick brown fox jumps over the lazy dog.Th
Message 4: quick brown fox jumps over the lazy dog.The
Message 5: quick brown fox jumps over the lazy dog.The
Message 6: uick brown fox jumps over the lazy dog.The q
Message 7: ick brown fox jumps over the lazy dog.The qu
⋮

Figure 6.8: The Sequence of messages expected to be passed on to the receiver's host

by stepping the program slightly differently one can produce the expected results as shown in Figure 6.8. The following sequence of commands causes the receiver to transfer to its host the messages sequence shown in Figure 6.9.

- Set a send-message breakpoint in S and a receive-message breakpoint in R.
- Run the program. When S stops, it has reached the point where it is sending M_{0_i}.
- Step S to its next event. This causes M_{0_i} to be sent and S's timer to be set. R stops when M_{0_i} arrives. Since R is stopped it does not send the acknowledgement within S's time-out interval. S times out and stops when it attempts to send M_{0_r}.
- Resume R. This causes R to send an acknowledgement for M_{0_i} and blocks for the next message to arrive. We denote this acknowledgement by ACK_a.

Message 1: The quick brown fox jumps over the lazy dog.
Message 2: quick brown fox jumps over the lazy dog.The
Message 3: quick brown fox jumps over the lazy dog.The
Message 4: uick brown fox jumps over the lazy dog.The q
Message 5: ick brown fox jumps over the lazy dog.The qu
⋮

Figure 6.9: The incorrect sequence of messages passed on to the receiver's host

- Resume S. This causes S to send M_{0_r} and receive ACK_a. Since the acknowledgement is received, S stops when it attempts to send M_{1_i}. Meanwhile, R stops when M_{0_r} arrives.

- Resume R. This causes R to receive M_{0_r}, send an acknowledgement denoted by ACK_b, and block waiting for the next message.

- Resume S. S sends M_{1_i}, receives ACK_b, and stops when it attempts to send M_{0_i}. R stops when M_{1_i} arrives. Since this message arrives damaged, R does not send an acknowledgement for it when continued next.

- Remove breakpoint from R and resume R.

- Remove breakpoint from S and resume S. S sends M_{0_i} and R receives it. Since R is expecting an M_1 message, this message is rejected but an acknowledgement is sent for it. S receives this acknowledgement and sends M_{1_i}. R receives this message, accepts it as the wanted M_1 message, and sends the acknowledgement for it. The algorithm now runs as expected.

This example showed how DPD can be used to control the program's behavior

to produce different results. This is an important feature of distributed debuggers because changing the behavior of a program may be the only way to find elusive bugs.

6.1.4 Replay

We now show how the roll back and replay feature can be used to analyze an event sequence. We use the same example as in Section 6.1.3 that caused the protocol to fail. We roll back the program to a state prior to where the unexpected behavior occurred and then re-step the program causing the protocol to work correctly.

Figure 6.10 shows the event sequence that causes the PAR protocol to fail. We roll back the program to the point where R receives M_{0_i}. How this is accomplished by DPD is shown in Figure 6.11. We can now step the program differently to produce the expected event sequence.

- Resume R. This causes R to send acknowledgement ACK_a for M_{0_i} and to block waiting for the next message to arrive.
- Resume S. This causes S to receive ACK_a. Since ACK_a is received, S stops when it attempts to send M_{1_i}.
- Resume S. S sends M_{1_i}, times out, and blocks attempting to send M_{1_r}.
- Remove breakpoint from R and resume it. R receives M_{1_i}, but because this message is damaged no acknowledgement is sent for it.
- Remove breakpoint for S and resume it. S sends M_{1_r} and R acknowledges it. The algorithm now continues to run as expected.

By replaying and stepping the program in the above manner, the protocol works as expected. If the program is rolled back once again but one starts at the fourth step

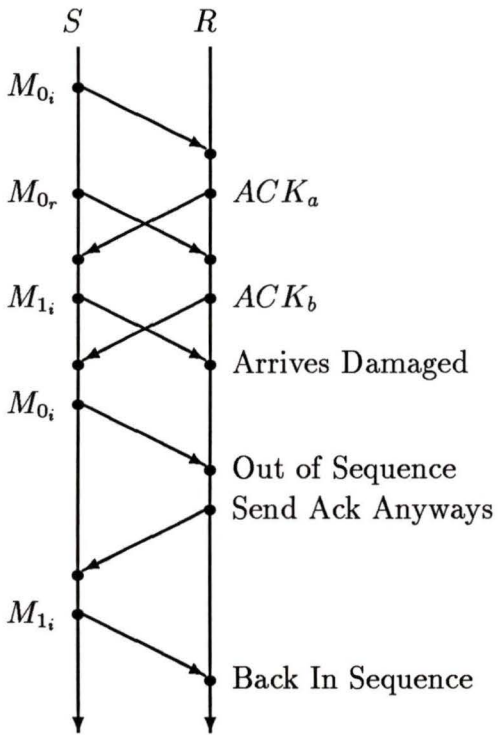


Figure 6.10: The event sequence that causes PAR to fail

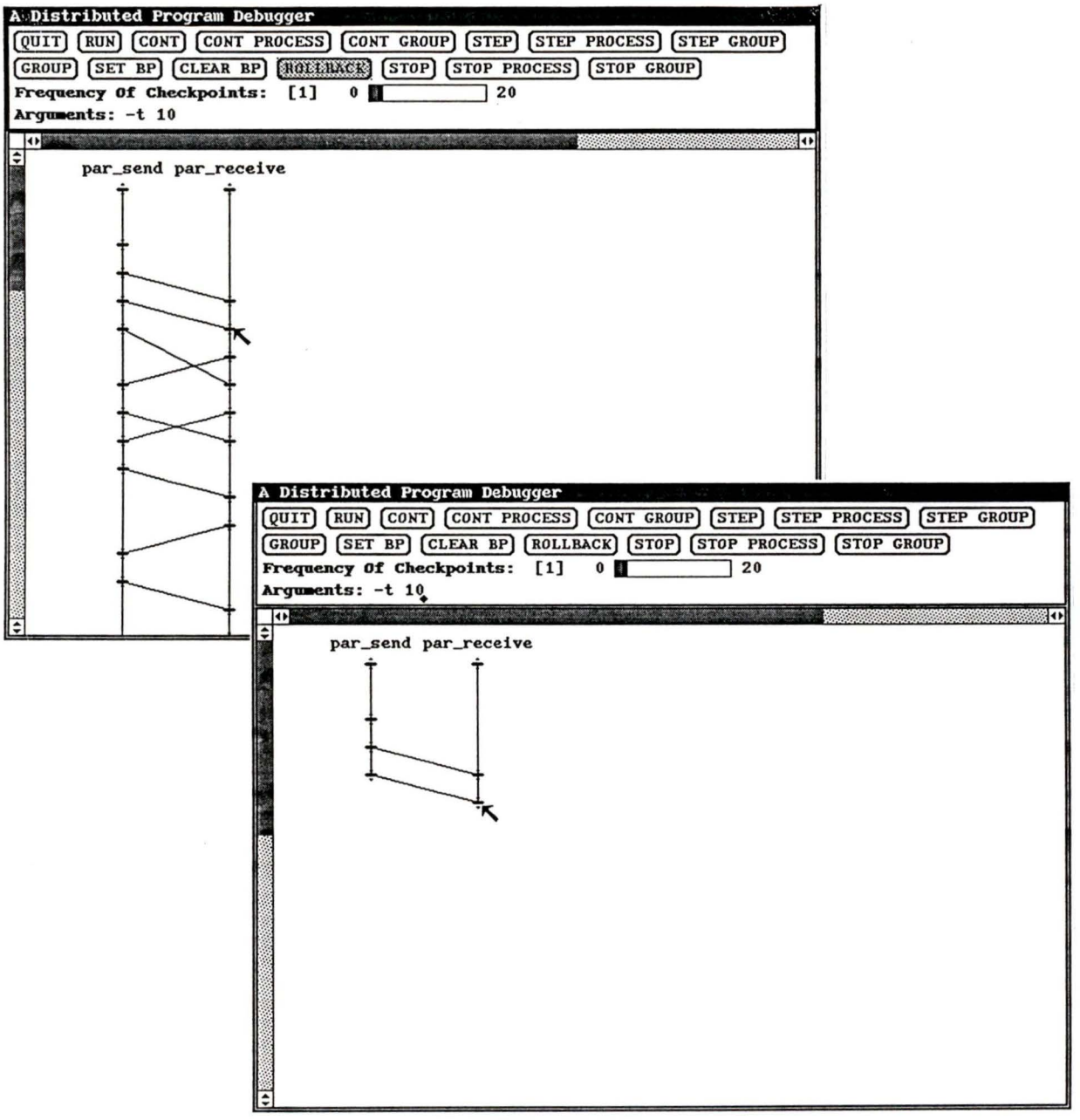


Figure 6.11: DPD rolling back PAR (Screens 1 and 2)

in Section 6.1.3, the protocol fails once more. This example shows how, by rolling back a program, different event sequences can be analyzed.

6.2 Performance Measurements

When debugging programs, the performance of the debugger becomes secondary to the its usability. As stated previously, DPD does affect the performance of a distributed program. This section shows where there are effects and describe what they are. There are two areas where DPD affects distributed programs the most. The first is at child creation time and the second is when checkpoints occur. For these two areas, we describe why they are time consuming, provide timing results, and describe their effects on a running distributed program. Two other important performance parameters for distributed debuggers are the time it takes to stop a distributed system and the time it takes to save trace data.

All the results for this chapter come from a network of eleven diskless Sun workstation connected to two Sun servers. Each workstation has a total of four megabytes of memory and all workstations in which a measurement was taken from were running the SunView windowing environment. The overall network was lightly loaded with on the average three other workstations being occupied. In general, the performance level of this network of workstations is considered poor. All results presented are the product of a sequence of twelve attempts. For each result, the highest and lowest scores of the twelve attempts were discarded and the remaining ten scores averaged.

6.2.1 Child Creation

As stated above, the most time-consuming operation performed by DPD is child creation for a number of reasons. The first is the necessity of the Remote Debugging Processes to create the child process. The second is the necessity of creating a DBX process and loading the child application process into it. The third is the necessity of initializing the child process before it can start execution.

The following steps show how normally a child process is created when it is not being debugged:

1. A message is sent from the parent application process to its local REM daemon.
2. The local REM daemon calculates which remote machine has the lowest load and sends a message to the REM daemon on it.
3. The remote REM daemon with the lowest load receives this message, creates the child process, and waits for the child to connect to it.
4. Once the child process is connected, its REM daemon sends a create acknowledgement back to the parent process via the parent's local REM daemon.
5. Once the parent process has received this acknowledgement, the child process has been successfully created.

When a child process is created under the auspices of DPD, the following steps are undertaken:

1. A message is sent from the parent application process to its local REM daemon.
2. The local REM daemon calculates which remote machine has the lowest load and sends a message to the REM daemon on it.

3. The remote REM daemon with the lowest load receives this message and sends a message to the Remote Debugging Process that resides on its machine.

4. The Remote Debugging Process receives this message and then creates the child process in the following manner:

   (a) a DBX process is created.

   (b) the child application process is loaded into memory by DBX.

   (c) breakpoints are set at the entry point of the child process and at the exit point of the roll back mechanism.

   (d) execution of the child process begins, but the child process immediately stops when it reaches the entry breakpoint.

   (e) UNIX's process identifier for the child process is extracted by making the child process write it to a file. The Remote Debugging Process opens this file and reads the child's process identifier.

   (f) the checkpoint algorithm is initialized.

   (g) the Event Collection Process for the child is created.

   (h) finally, the child process is allowed to resume.

5. Once the child process has connected with its local REM daemon, a create acknowledgement is sent back to the parent process.

6. Once the parent process has received this acknowledgement the child process has been successfully created.

Since creating a child process under DPD requires the creation of a DBX process, it obviously takes much longer than normal child creation.

Child Creation Times *(In Seconds)*		
Without DPD	Load, initialize, and start execution under DBX	Total time to start under DPD
0.60	12.54	14.26

Table 6.1: Times to create the PAR child process

Table 6.1 compares the times required to create a process without debugger control; the time to load, initialize, and start execution of the process under DBX control; and the total time to start the process under DPD control. As shown, the greatest overhead involves loading and initializing the process under DBX. The difference between the total time to create a process under DPD and the time it is loaded under DBX is approximately 1.7 seconds. This is the overhead caused by the extra communication and processing time required by DPD. One could probably reduce the total overhead time if one were to embed the functionality of the sequential debugger within DPD.

Obviously, changing the timing at child creation time can impact a distributed program. For example, it may be important that all the child processes start as simultaneously as possible so that no one process gets too far ahead of all the others. This may be desired for parallel computations or for fault tolerance. A simultaneous start with REM is almost possible since process creation time is small. If this situation is needed when debugging, this can be accomplished by setting a global *connect* breakpoint before the program is run and then starting the program. All newly created child processes stop after they have connected with REM. Once all the child processes are created and stopped, the *continue* command can then be issued and all processes start almost simultaneously.

Checkpoint Times		
Checkpoint Size (In Bytes)	Checkpoint Time (In Seconds)	File Copy Time (In Seconds)
141000	4.71	2.59
1320155	44.6	31.4

Table 6.2: Time to produce a checkpoint for a process

6.2.2 Checkpoints

Checkpointing a process is another area that causes delays in the execution of a distributed program. To checkpoint, one must save the internal state of the process. This is similar to producing a *core* dump under UNIX. To produce a checkpoint, the process's registers, stack, and dynamic memory must be saved. The amount of time it takes to checkpoint a process is directly related to the size of the process's stack and its dynamic memory. Because of the volume of the information being saved, checkpointing is influenced by the performance of the media the checkpoint is being saved to.

The times to produce checkpoints for two processes are tabulated in Table 6.2. One process has a relatively small combined stack and dynamic memory area and the other process has a relatively large data area. To be fair to ourselves, we compared the time to produce a checkpoint to the time it takes the operating system to read and write a similar size file. As shown, it takes what seems to be an excessive amount of time to produce a checkpoint. But compared with the performance of simply copying a similar size file on the system, the performance is not excessive. The reason why file accessing is so slow is because all the workstations are diskless. This means that all file accesses must occur through a remote server on the network.

The effects of checkpointing is a slower execution time for the application program. Again, any change in the timing of a faulty distributed program can cause considerable behavioral changes. Checkpointing has the greatest impact on programs that make use of the asynchronous receive feature. For example, suppose that we have a distributed program where the parent process is in asynchronous mode and its child process is in synchronous mode. The parent process sends n messages, one right after another, to the child process. For every message the child process receives it sends an acknowledgement back to the parent. If the program is run without being debugged, there is a good probability, if n is large, that the first acknowledgement from the child process arrives before the parent has sent its last message. If the program is run with the debugger and the child process checkpoints on the first message it receives, by time the parent process receives the first acknowledgement it has sent all of its messages. To reproduce the required behavior, one may have to roll back and step the application program manually. To lessen the impact of checkpoints on distributed programs one should reduce the frequency of checkpoints as described in Section 5.4.7. Another solution is to incorporate the dirty page scheme in [FB89].

6.2.3 Stopping

Stopping a process in a timely manner is important for distributed debuggers because the sooner you can stop the process the more likely it is stopped in an interesting state. As discussed previously, stopping all the application processes instantaneously is impossible. The best one can do is stop the processes as soon as possible. There are two delays involved in stopping a process. The first is the delay in sending a stop message to the required processor and the second is the processing time to physically stop the processes.

Under REM, the transmission time to send, transmit, and receive a message is approximately 48 milliseconds. Once the *stop process message* arrives at an RDP, issuing the DBX "stop" command and waiting for the required acknowledgement takes an additional 690 milliseconds. It should be noted that the target application process is stopped prior to the acknowledgement being received by the RDP. The reason for the long delay in waiting for the stop acknowledgement is that once the target process is stopped, DBX must send the stop acknowledgement to the RDP and the RDP must then read the acknowledgement from the pipe. It is not until the stop acknowledgement has been read from the pipe that the process is considered to be stopped even though it has been physically stopped much earlier.

When stopping an entire distributed program, the parent process needs to be stopped along with every remote child process. There has to be one stop message sent for every remote process because there is no broadcast mechanism supplied by REM at present. Since the time to prepare and send a message is small, all the "stop" messages are sent to the RDPs first. This means that the physical activity of stopping the remote processes can occur in parallel with the stopping of the parent process. On the average, the time (in milliseconds) to stop a distributed program can be calculated from the following equation:

$$(n-1) \times 15 + 48 + 690 \tag{6.1}$$

Where n is the number of remote processes, 15 milliseconds is the time to prepare and send a message by the UIP, and 48 milliseconds is the time for the UIP to send a message and for an RDP to receive it. Equation 6.1 is derived from the following: the time to stop the last remote process is 48 + 690 milliseconds and the $(n-1) \times 15$ is the time in milliseconds to prepare and send $n-1$ messages to the remaining remote processes. The parent process should be stopped prior to the last child process being stopped and, thus, does not figure into the calculation as shown in Figure 6.12.

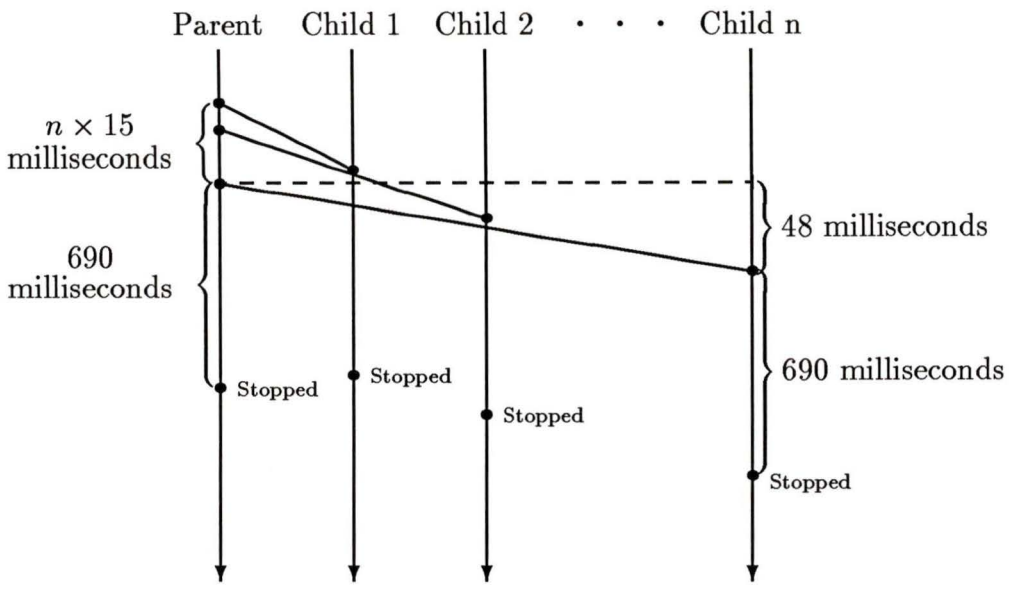


Figure 6.12: Time it takes to stop a distributed program

If the parent process is the first process to be stopped, the calculation is

$$690 + (n - 1) \times 15 + 48 + 690 \tag{6.2}$$

because the "stop" messages to the remote process are not sent until the parent process has successfully stopped. When REM incorporates its broadcast mechanism, the equation changes to:

$$48 + 690 \tag{6.3}$$

assuming that the time to prepare, send, and transmit a broadcast message is equivalent to sending a single point-to-point message.

6.2.4 Traces

Saving trace data requires event information to be written to disk. Every event written to disk contains at least a 24 byte header. If a send or receive event occurs, the header is followed by the contents of the message. If a create event occurs, the header is followed by the name of the child. Since there is usually only a small amount of data to be written, the delay is not excessive.

6.3 Summary

We have shown that DPD is usable as a distributed debugger. The program used as our test example was the PAR protocol as described in [Tan81]. DPD was used on this program to inspect its system state, to control execution, and to analyze its behavior. We presented in detail the steps used to debug three different aspects of the program. First, we have shown that by inspecting the program's system state through the graphical display and by inspecting messages, an error can be found and isolated in a process. Second, by setting breakpoints and stepping processes we have shown that DPD can be used to control a program's behavior. Finally, we have shown how replay can be used to analyze a program's behavior.

This chapter also presented the performance measurements that are important for DPD. We have shown that DPD does affect the execution speed of distributed programs, we have also described what these effects are and how they can be minimized. We have found that DPD affects distributed programs the most during child creation and checkpointing. We discussed why these two areas are so time consuming. The consensus for child creation was that the underlying sequential debugger was the bottleneck because it takes so long for it to load and initialize a process.

For checkpointing, the major bottleneck was saving the information to disk. Other performance areas measured include the time it takes to stop a process and the time to save trace data. We have shown how long it takes to stop an individual process and we presented a formula that can be used to calculate how long it takes to stop an entire distributed program.

Chapter 7

Conclusions

The main goals of this thesis were to explore distributed debugging and to implement a distributed debugging facility for REM. Our approach to the design and development of DPD was to:

- debug distributed programs in a high-level and abstract manner,
- limit ourselves to debugging only application programs not system programs,
- realize that DPD may have some effect on the target application program,
- not modify the operating system kernel,
- use as many existing tools as possible, and
- implement a dynamic roll back and replay facility.

In this chapter we state our findings and describe possible future work.

7.1 Contributions and Achievements

We have succeeded in our main goals of exploring distributed debugging and implementing a debugger for REM. We have also succeeded in implementing a true distributed debugger, not a glorified sequential debugger. Furthermore, DPD debugs distributed programs in a high-level and abstract manner. We believe that DPD is extensible, portable, and easy to use. To show usability, we used the well-known PAR protocol[Tan81] to find programming and algorithmic errors in a distributed application.

DPD did not require changes to the operating system kernel, thus making it more portable and easier to administer. We needed to monitor certain system calls to successfully implement our checkpoint facility; however, this monitoring is transparent to the user and does not affect the semantics of the monitored system calls.

Many existing tools were used to ease the development of DPD. The most prominent tools used were DBX[Muc88] and a checkpointing facility developed by Curry[Cur88]. DBX was used as the underlying sequential process controller. This relieved us from the burden of developing a distributed program debugger as well as a sequential process controller. The checkpointing facility provided us with the basis in which processes can be checkpointed under UNIX. Development of such a facility from scratch would have been difficult and time consuming.

One of our main development goals was to implement a dynamic roll back and replay facility. This facility was successfully implemented by using the recovery algorithm developed by Johnson and Zwaenepoel[JZ88]. This algorithm finds consistent and recoverable system states in which distributed programs can be rolled back to. Furthermore, our roll back mechanism does not need synchronous clocks or inter-process communication to control checkpoints such as in Curtis[CW82] and

Chandy[CL85]. Even though checkpointing in DPD is completely asynchronous, consistent system states in which to roll back to and checkpoints to restart the processes with are found quickly and easily.

We also made substantial enhancements to REM so that it is now capable of handling larger and more complex distributed computing problems than previously possible. We have enhanced REM from a prototype implementation to a reliable and useful tool to develop distributed applications.

7.2 Future Work

DPD is currently a useful tool for debugging distributed programs; however, additional tools are needed by DPD so that it can become a more complete debugging environment.

DPD is currently implemented with only sequential style breakpoints. Distributed breakpoints as described in Section 5.4.6 are desirable.

We could provide access to the source code of individual processes since there are times when this may be useful. This feature is relatively easy to implement given that a sophisticated underlying sequential debugger is already being used.

Finally, DPD should be enhanced to use the multicast facility that is currently being developed for REM. This feature will reduce DPD's communication overhead and give better response to existing tools such as distributed stopping.

Bibliography

[AM86] E. Adams and S. Muchnick. Dbxtool: A window-based symbolic debugger for Sun workstations. *Software – Practice & Experience*, July 1986.

[BFV86] Fabrizio Baiardi, Nicoletta De Francesco, and Gigliola Vaglini. Development of a debugger for a concurrent language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.

[BMV83] F. Baiardi, E. Matteoli, and G. Vaglini. Development of a debugger for a concurrent language. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 98–106, March 1983.

[Bor88] Borland International. *Turbo Debugger: User's Guide*, 1988.

[BW82] Peter C. Bates and Jack C. Wileden. Event definition language: An aid to monitoring and debugging of complex software systems. In *Fifteenth Hawaii International Conference on System Sciences*, January 1982.

[BW83a] Peter Bates and Jack Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *The Journal of Systems and Software*, 3:255–264, 1983.

[BW83b] Peter Bates and Jack C. Wileden. An approach to high-level debugging of distributed systems (preliminary draft). *Communications of the ACM*, pages 107–111, 1983.

[Car83] Thomas A. Cargill. The Blit debugger. *The Journal of Systems and Software*, 3:277–284, 1983.

[Car85] Thomas A. Cargill. Implementation of the Blit debugger. *Software – Practice & Experience*, 15(2):153–168, February 1985.

[CC89] E. Jane Cameron and David M. Cohen. The IC* system for debugging parallel programs via interactive monitoring and control. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):261–270, January 1989.

[Che89] Wing Hong Cheung. *Process and Event Abstraction for Debugging Distributed Programs.* PhD thesis, Department of Computer Science, University of Waterloo, 1989.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[Coo87] Robert Cooper. Pilgrim: A debugger for distributed systems. In *IEEE Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 458–465, September 1987.

[Cor90] Brian Corrie. A workbench for realistic image synthesis. Master's thesis, University of Victoria, 1990.

[CP83] P. Corsini and C. A. Prete. Multibug: Interactive debugging in distributed systems. *IEEE Micro*, pages 26–33, June 1983.

[Cur88] David A. Curry, 1988. Personal correspondence.

[CW82] Ronald Curtis and Larry Wittie. Bugnet: A debugging system for parallel programming environments. In *IEEE Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 394–399, 1982.

[Els89] I. J. P. Elshoff. A distributed debugger for Amoeba. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):1–10, January 1989.

[Ens78] Philip H. Enslow. What is a "distributed" data processing system? *Computer*, 11:13–21, January 1978.

[FB89] Stuart I. Feldman and Channing B. Brown. IGOR: A system for program debugging via reversible execution. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):112–123, January 1989.

[FZ89] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints (extended abstract). To Be Published, 1989.

[GMGK84] H. Garcia-Molina, F. Germano, and W. Kohler. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, SE-10(2):210–219, March 1984.

[HHK85] P. Harter, D. Heimbigner, and R. King. IDD: An interactive distributed debugger. In *IEEE Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 498–506, May 1985.

[HW88] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *IEEE Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 166–175, August 1988.

[JBW87] S. Jones, R. Barkan, and L. Wittie. BUGNET: A real time distributed debugging system. In *IEEE Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 56–64, March 1987.

[JZ88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 171–181, May 1988.

[Ken86] Michael S. Kenniston. *Debugging the Communication Behavior of Distributed Programs in a Message-Based System.* PhD thesis, Department of Computer Science, Stanford University, 1986.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):552–565, July 1978.

[Liu90] Francis Liu. Reliable multicast communications for parallel computing in distributed environments. Master's thesis, University of Victoria, 1990.

[LM89] Thomas J. LeBlanc and Barton P. Miller. Workshop summary. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):ix–xxi, January 1989.

[LMC87] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[LR85] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed programs. In *IEEE Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 515–522, 1985.

[MC88] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *IEEE Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, August 1988.

[MC89] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):141–150, January 1989.

[Mic87] Microsoft Corporation. *CodeView and Utilities*, 1987. Document No. 410840010-500-R02-0887.

[MMS86] Barton P. Miller, Cathryn Macrander, and Stuart Sechrest. A distributed programs monitor for Berkeley UNIX. *Software – Practice & Experience*, 16(2):183–200, February 1986.

[Muc88] Steven S. Muchnick. Dbx and Dbxtool interfaces. Technical report, Sun Microsystems Inc., January 1988.

[PL89] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):124–129, January 1989.

[San90] The Santa Cruz Operation. *Socket Programmer's Guide*, 1990. Document No. XN-8-01-89-1.1.0A.

[SBN89] David Socha, Mary L. Bailey, and David Notkin. Voyeur: Graphical views of parallel programs. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):206–215, January 1989.

[SCT88] G. C. Shoja, G. Clarke, and T. Taylor. REM: A distributed facility for utilizing idle processing power of workstations. *Distributed Processing*, pages 205–218, 1988.

[SCTT88] G. C. Shoja, G. Clarke, T. Taylor, and W. Taylor. A software facility for load sharing and parallel processing in workstation environments. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, volume II, pages 220–230, January 1988.

[Sho90] G. C. Shoja. A distributed facility for load sharing and parallel processing among workstations. To appear in Journal of Systems and Software, 1990.

[Sid90] Robert S. Side. Rem interface routines. Technical report, University of Victoria, 1990.

[Sin90] Craig Sinclair. Automatic parallelism of C programs using shared data objects. Master's thesis, University of Victoria, 1990.

[SK86] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *IEEE Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[Smi84] Edward T. Smith. Debugging tools for message-based, communicating processes. In *IEEE Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 303–310, May 1984.

[Smi85] Edward T. Smith. A debugger for message-based processes. *Software – Practice & Experience*, 15(11):1073–1086, November 1985.

[Sto89] Janice M. Stone. A graphical representation of concurrent processes. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):226–235, January 1989.

[Sun88] Sun Microsystems. *Sun Operating System 4.0*, 1988.

[Tan81] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 07632, 1981.

[TR85] Andrew S. Tanenbaum and Robbert Van Reneese. Distributed operating systems. *Computer Surveys*, 17(4):419–470, December 1985.

[WC85] Larry Wittie and R. Curtis. Time management for debugging distributed systems. In *IEEE Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 549–550, May 1985.

[Wit89] L. Wittie. Debugging distributed C programs by real time replay. *ACM SIGPLAN and SIGOPS: Workshop on Parallel and Distributed Debugging*, 24(1):57–67, January 1989.

Appendix A

DPD Brief User's Guide

This appendix describes how to compile and link application processes and how to use DPD. We assume that the reader already knows how to initialize and use REM.

A.1 Compile and Link

To compile an REM application program so that DPD can be used to debug it, the following flags must be passed to the C compiler:

- **-g:** This is the standard flag that causes the C compiler to produce the additional symbol table information for DBX. See cc(1) in [Sun88] for conflicts and constraints when using this flag.
- **-Bstatic:** This flag tells the compiler not to use shared libraries. By default, SUN OS 4.0 executables are created using shared libraries. The checkpoint facility cannot be used with shared libraries and thus this flag is necessary.

With the introduction of DPD, all REM application must be linked with the following libraries: `librem.a`, `libdbg.a`, `libutils.a`, and `libchkpt.a`. This means in addition to the flag needed to link with REM's library, `-lrem`, the following flags are also needed: `-ldbg`, `-lutils`, and `-lchkpt`. All of DPD's libraries are needed to be linked with all REM application programs even if they are not going to be debugged.

A.2 Getting Started

Once the application program is compiled and linked with the proper compiler flags, the application can be debugged with DPD. To invoke DPD, type `dpd <application>` as a UNIX command where `<application>` is the name of your distributed program. DPD must be run in the SunView windowing environment. There is currently no command line user interface.

When DPD starts, it requires certain environment variables to be set. These environment variables are:

- **RDPPATH:** This variable tells DPD where to look for the executable image of the RDP. If this variable is not set, the RDP executable image is expected to be in the current working directory.
- **ECPPATH:** This variable tells DPD where to look for the executable image of the ECP. If this variable is not set, the ECP executable image is expected to be in the current working directory.
- **SUITENAME:** This variable tells DPD which suite to run in. If this variable is not set, the default suite "`suite1`" is used.

These environment variables can be set in the `.login`, `.cshrc`, or `.profile` files or in a `.DPDrc` file. DPD first looks for a `.DPDrc` file in the current working directory. If one is not found, DPD looks in the $HOME directory for one. If neither `.DPDrc` files are found, the environment is searched. If your environment does not contain these variables, the defaults are used.

The syntax of the `.DPDrc` file is as follows:

```
<variable> = <value>
```

where `<variable>` is one of RDPPATH, ECPPATH, or SUITENAME and `<value>` is any string. The current format does not allow comment lines to be included in the file. An example of a .DPDrc file is as follows:

```
RDPPATH = /usr/local/bin/rdp
ECPPATH = /usr/local/bin/ecp
SUITENAME = debugsuite
```

A.3 Commands

This section presents a brief summary of the commands available to the user. See Section 5.3.2 for a complete discussion of all commands.

- **Argument:** Command line arguments for the application's parent process can be entered at this prompt. Any string can be entered after this prompt; however, UNIX's shell wild cards are not expanded.
- **Checkpoint Interval:** Sets the number of *receive message* events that must occur before a checkpoint is produced. If the checkpoint interval is 0, no checkpoints are produced.

- **Run:** Starts the application program.
- **Stop:** Stops all active application processes.
- **Stop Process:** Stops an individual application process. To select a process, a browser of processes is displayed.
- **Cont:** Resumes all stopped application processes.
- **Cont Process:** Resumes an individual application process that is stopped. To select a process, a browser of processes is displayed.
- **Step:** Steps the application program one event.
- **Step Process:** Steps an individual application process one event. To select a process, a browser of processes is displayed.
- **Group:** Creates abstract process groups. To select the processes that are to be placed in a group, a browser of processes is displayed from which multiple processes can be selected.
- **Stop Group:** Stops all processes in a group. To select a group, a browser of groups is displayed.
- **Cont Group:** Resumes all stopped process in a group. To select a group, a browser of groups is displayed.
- **Step Group:** Steps a group one event. To select a group, a browser of groups is displayed.
- **Set Breakpoint:** Sets a breakpoint for an individual process or globally for all processes in the application program. First, a menu is displayed containing *Global Breakpoint* and and all of the processes. If *Global Breakpoint* is selected,

a menu of breakpoints is displayed. All processes are set with the selected breakpoint. If an individual process is selected, the menu of breakpoints is be displayed. The selected process is set with the selected breakpoint.

- **Clear Breakpoint:** Clears a breakpoint for an individual process or globally for all processes in the application program. First, a menu is displayed containing *Clear Global Breakpoint* and all of the processes. If *Clear Global Breakpoint* is selected, a menu of breakpoints is displayed. The selected global breakpoint is cleared from all processes. If an individual process is selected, the menu of breakpoints is displayed. The selected process has the selected breakpoint cleared.

- **Rollback:** Rolls back the program to an event. To roll back, the user must select an event from the Process Display Window. Once an event is selected, the application program is rolled back to that event.

A.4 New REM Interface Routines

This section contains the definitions for the interface routines added to REM's library during the development of DPD. These routines were added to facilitate DPD; however, all of these routines can be used by other application programs. See the *REM Interface Routines*[Sid90] document for a description of all REM's interface routines.

Syntax :

void us_s_asynchandler (void (*handler)(Vpid src_pid, int msg_len, int msg_typ))

Description :

This routine is used to set the asynchronous and synchronous mode for REM. If `handler` is not NULL, whenever a new message arrives at the process, the function pointed to by `handler` is called with the virtual process identifier of the sender of the message (`src_pid`), the message len (`msg_len`), and the message type (`msg_typ`). From within the asynchronous handler, one is required to call `us_recv_dist_typ`. If `handler` is NULL and we are in asynchronous mode, the mode changes to synchronous mode.

If there are queued messages waiting to be received prior to this routine being called with a non NULL function pointer, these messages are sent to the function before `us_s_asynchandler` returns.

See Also :

`us_recv_dist_typ` and `us_s_waitasync`.

Syntax :

void us_s_waitasync (void)

Description :

This routine causes the process to block until a call to `us_s_asynchandler` is made with a NULL handler. Obviously, one can still receive asynchronous messages while blocked in this routine.

Syntax :

void us_s_flush (void)

Description :

This routine flushes all internal queues of messages, flushes the socket to REM, and causes REM to flush its internal queues.

VITA

Surname: **Side** Given Names: **Robert Samuel**
Place of Birth: **Dawson Creek, B.C., Canada** Date of Birth: **January 22, 1963**

Educational Institutions Attended:

University of Victoria	1981 to 1986
University of Victoria	1987 to 1990

Degrees Awarded:

B.Sc.	1986	University of Victoria

Partial Copyright License

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

DPD: A Distributed Program Debugger for the REM Environment

Author: [redacted]

Robert S. Side
August 17, 1990