Technical Report

UCAM-CL-TR-128
ISSN 1476-2986

Number 128

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Debugging concurrent and distributed programs

Robert Charles Beaumont Cooper

February 1988

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 1987 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Debugging Concurrent and Distributed Programs
Robert C. B. Cooper

Abstract

This thesis aims to make one aspect of distributed programming easier: debugging. The principles for designing and implementing an interactive debugger for concurrent and distributed programs are presented. These programs are written in a high-level language with type-checked remote procedure call. They execute on the nodes of a local computer network and interact with the other programs and services which exist on such a network.

The emphasis is on debugging programs in the environment in which they will eventually operate, rather than some simulated environment oriented specifically to the needs of debugging. Thus the debugging facilities impose a low overhead on the program and may be activated at any time.

Ideally the actions of the debugger should be transparent to the execution of the program being debugged. The difficult problem of avoiding any alteration to the relative ordering of inter-process events is examined in detail. A method of breakpointing a distributed computation is presented which achieves a high degree of transparency, in the face of arbitrary process interactions through shared memory.

The problems of debugging programs that interact with network services, which are shared concurrently with other users of the distributed environment, are examined. A range of debugging techniques, some of which are directly supported by the debugger, are discussed.

A set of facilities for debugging remote procedure calls is presented, and the functions required of the operating system kernel and runtime system to support debugging are also discussed. A distributed debugger is itself an example of a distributed program and so issues such as functional distribution and authentication are addressed.

These ideas have been implemented in *Pilgrim*, a debugger for Concurrent CLU programs running under the Mayflower supervisor within the Cambridge Distributed Computing System.

Preface

I thank Roger Needham, my supervisor, for giving guidance and advice when I needed it. I also thank Andrew Herbert, the prime mover behind the Mayflower Group, for encouraging me to study this topic. The Computer Laboratory, and the Mayflower Group in particular, provide a stimulating intellectual environment in which to do research. The people inside and outside Mayflower to whom I owe thanks include: Jean Bacon, Mike Burrows, Dan Craft, Graham Hamilton, Paul Karger, Ian Leslie, Derek McAuley, Andy Seaborne, and Wei Mian. Special thanks are due to those who carefully read and commented on drafts of my thesis: Roger Needham, Jean Bacon, Mike Burrows, and Miriam Leeser.

While studying at Cambridge I received funding for tuition, living expenses, research equipment, and conference travel from: the Cambridge Commonwealth Trust, the New Zealand University Grants Committee, Churchill College, and the Science and Engineering Research Council.

I thank my parents for all they have done.

Finally I thank my wife, Miriam Leeser. I couldn't have done it without you.

This dissertation is the result of my own work and, unless explicitly stated in the text, contains nothing which is the outcome of work done in collaboration. No part of this dissertation has already been or is currently being submitted for any degree, diploma or other qualification at any other university.

Contents

List of Figures

Chapter 1

Introduction

Producing concurrent software to execute on distributed hardware is a difficult business. In this thesis I am trying to make one aspect of distributed programming easier: debugging.

1.1 Motivation

There are two themes motivating this research. The first is the employment of a high-level language. While the hardware technology of distributed computing has existed for a decade, the software technology has been much slower to develop. The rise of language level remote procedure call (RPC) was the first important step. The next ought to be source level debugging of distributed programs. RPC allows distributed programs to be written without intimate knowledge of the implementation, such as which network protocol is used. But little has been achieved if this knowledge is still required when debugging distributed programs. Besides the obvious convenience of debugging in source language terms, many of the techniques for concurrent and distributed debugging detailed in this thesis would be impossible without information about the names and types of variables and objects in the program.

The second theme is that of debugging distributed programs in their target environment under real conditions. Despite the best efforts during program development, many bugs will remain even after significant, systematic testing in a debugging environment. Hence the need for debugging tools which operate on programs under conditions of actual use, and perhaps after those programs have gone into service. This is especially so with distributed programming where the surrounding environment may be large, complex and (regrettably) not fully specified and thus impossible to simulate in all its detail.

Developing tools and techniques for debugging distributed programs is a three-way trade-off between providing rich and detailed information to the programmer, while avoiding where possible any alteration to the computation under examination, and without interfering with the computations of other users of the distributed environment.

Contribution

My specific contributions to the debugging of concurrent and distributed systems are:

1. A set of debugging facilities for remote procedure calls. These facilities can be integrated easily with those for debugging local procedure calls. I present a scheme for implementing these facilities which is suited to most RPC systems.

2. A method for *transparently* halting the processes in a distributed program. This should be used by an interactive debugger when a breakpoint is triggered, or when an execution error is encountered. My definition of transparency considers not only process communication through remote message passing, but also process interactions through shared memory.

3. Techniques for debugging programs that use public network servers. These techniques enable one client of a server to be debugged interactively while avoiding interference with the other concurrent clients of the server.

4. System support for debugging concurrent processes. I discuss the facilities required to access and control process states and how this debugging support may be implemented in the operating system and the language runtime library.

5. Target environment debugging. I emphasize debugging methods and implementation techniques which are efficient enough to used at all times.

To evaluate these ideas in a practical setting I have implemented the *Pilgrim* debugger. I have implemented all the new debugging techniques described in this thesis, except where explicitly stated otherwise.

1.2 Scope

I am concerned with the debugging of distributed programs consisting of multiple concurrent processes which execute on a number of *logical nodes*. Processes within a logical node share memory and may interact via monitors, semaphores, or critical regions. Processes in different logical nodes do not share memory and communicate using a language-level type-checked facility such as remote procedure call, even when they reside in the same physical node. In general, I assume physical nodes are connected by a local area network although reference to other kinds of media will sometimes be made. The important characteristics of local networks include high bandwidth, low latency, and very low error rates in the absence of congestion at the receiving machine.

The scope of this work encompasses a large class of useful distributed computer systems and applications, such as an office automation environment, or a database server.

Some justification is required of the system types I have chosen to cover, or chosen to ignore. Systems which use untyped messages for communication have been excluded. Many bugs in such systems result from incorrectly formatting data in messages. These bugs are much more easily prevented, by the use of type-checked remote procedure call or message passing, than debugged, no matter how powerful the debugger.

I include systems in which some processes may share memory. It will become clear in later chapters that admitting these systems complicates the task of the debugger, and of the programmer doing the debugging. Yet this kind of process organization has been popular in a number of successful environments such as Cedar[1] and Amoeba.[2] Why is this so? In monitor-based shared-memory systems it is natural to associate a process with each "activity" in the application. Each process sequentially performs the steps associated with that activity, synchronizing with other processes when accessing shared resources. The resulting program clearly represents the inherent concurrency in the application. By comparison, when there is only one process per address space it is necessary to multiplex

[1] W. Teitelman. "A Tour Through Cedar." *Proc. of the 7th International Conference on Software Engineering*, Orlando, Florida, March 1984, IEEE Computer Society Press, p. 181.

[2] S. J. Mullender and A. S. Tanenbaum. *The Design of a Capability-Based Distributed Operating System.* Technical Report CS-R8418, Dept. of Computer Science, Centrum voor Wiskunde en Informatica, Amsterdam, October 1985.

a number of concurrent activities in that process. To provide timely service and avoid deadlock it will be necessary for a process to suspend one activity and save its data in a queue before selecting another activity, dequeueing its data and performing some steps on its behalf. Such a program structure is more difficult to understand. Furthermore, monitor-based shared-memory systems usually have better performance than the equivalent system in which processes have disjoint address spaces. The case for shared-memory processes is put strongly by Liskov and Herlihy,[3] and Lampson and Redell.[4]

1.3 Context

The practical work for this thesis centres on the implementation of the Pilgrim debugger for Concurrent CLU programs running under the Mayflower supervisor.

Concurrent CLU

CLU[5,6] is a sequential language which provides very good support for user-defined abstract data types. The principal program structuring device is the *cluster* which specifies the visible interface of an abstract data type and the private implementation. Those familiar with Ada or Modula-2 might think of a cluster as similar to a package or module.

CLU has been extended at Cambridge with features for concurrent and distributed programming to produce Concurrent CLU. Each node of a Concurrent CLU program possesses multiple light-weight processes which share memory. Process interactions are mediated by *gates*, a user-defined critical region construct, and semaphores. The combination of clusters, gates, and semaphores encourages a programming style based on monitors.[7]

---

[3]B. Liskov and M. Herlihy. "Issues in Process and Communication Structure for Distributed Programs." *Proc. of the 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems,* October 1983, p. 123.

[4]B. W. Lampson and D. D. Redell. "Experience with Processes and Monitors in Mesa." *Comm. of the ACM,* **23**(2), February 1980, p. 105.

[5]B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Manual. Lecture Notes in Computer Science,* **114**, Springer Verlag, Berlin, 1981.

[6]B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, Mass., 1986.

[7]R. C. B. Cooper and K. G. Hamilton. "Preserving Abstraction in Concurrent Programming." *IEEE Trans. on Software Engineering* (*to appear* February 1988).

(Concurrent CLU should not be confused with the Argus language, developed at MIT and based on CLU, which supports user-defined distributed objects with atomic and recoverable semantics.)

Communication between nodes uses the RPC system developed by Hamilton.[8] His RPC mechanism is fully type-checked and permits arbitrarily complex objects, including those of user-defined type, to be transmitted between nodes. An unusual feature is the provision of two sets of call semantics implemented by two RPC protocols. The *exactly-once* protocol provides reliable communication of arguments and results in the absence of node failures. The *maybe* protocol allows the programmer to handle both transient errors (such as congestion at the receiving node) and failures (such as a node crashing) with retry strategies appropriate to the application at hand. Binding of RPCs is performed dynamically at run-time.

The Mayflower environment

Each node of a Concurrent CLU program runs on a single processor Motorola MC68000 system connected to a Cambridge Ring network, and executes under the Mayflower supervisor. This is a small operating system providing process switching, file access, and network I/O. Mayflower can support one or more address spaces called *domains*, each of which can execute one node of a Concurrent CLU program. Normally however only one domain is used on each machine.

Mayflower is part of the Cambridge Distributed Computing System[9] and makes use of many of its servers. To run a Concurrent CLU program a user contacts one of these servers called the Resource Manager and requests a number of machines from the Processor Bank. The Resource Manager uses other servers to load a copy of the Mayflower supervisor into each of these machines. The user instructs each of these supervisors to begin running a node of the program. Once running, the program may itself use the Resource Manager, file servers, and so on.

---

[8] K. G. Hamilton. *A Remote Procedure Call System.* Ph.D. Dissertation, Technical Report 70, Computer Laboratory, Cambridge University, December 1984.

[9] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System.* Addison-Wesley, London, 1982.

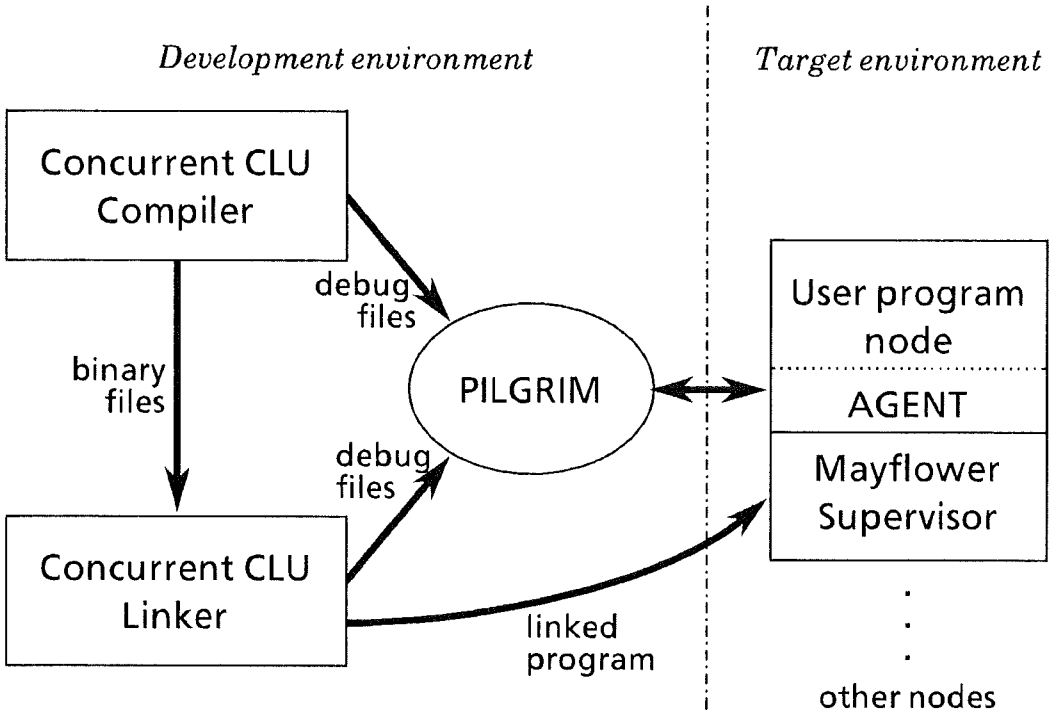

Figure 1.1: The structure of Pilgrim debugger

Concurrent CLU and Mayflower are exemplary of the kind of distributed systems which this thesis is about.

The Pilgrim debugger

The analysis of debugging in the following chapters will be much easier to understand if a short description of Pilgrim is provided now.

Figure 1.1 shows how Pilgrim interacts with the other elements of the Mayflower environment. The Concurrent CLU compiler produces a binary file and a file containing debugging information for each source file. The Concurrent CLU linker reads the compiler binaries and produces a linked image and a debugging file for each node of a distributed program. It also includes a piece of debugging support code, the *agent*, with each node. As described above, the program is started by loading the appropriate linked images onto MC68000s running the Mayflower supervisor.

At any time while the program is executing, a debugging session may be begun by starting

up Pilgrim and issuing the debug command. Pilgrim accesses and controls each program node by communicating with its agent. The agent performs debugging actions, such as setting and clearing breakpoints and accessing memory, on Pilgrim's behalf.

Various aspects of the Pilgrim implementation are described at relevant points in the following chapters. The distributed nature of Pilgrim is discussed fully in Chapter 8. That chapter includes a specification of the complete agent-debugger interface, providing a unified picture of the various implementation details presented in the intervening chapters. Some information about Pilgrim appearing in this thesis has been published earlier in a different form.[10]

1.4 Terminology and notation

Throughout this dissertation the following terms have special meanings:

user, programmer: These terms are used interchangeably to refer to the the user of the debugger who is often the person who wrote the program being debugged.

debuggee: A program being debugged under control of a debugger.

sequential debugger: A source-level debugger for sequential non-distributed programs.

client, server, service: A server is a node which offers some facility, such as file storage or authentication, in a distributed computing system. Often a facility is implemented by a collection of servers. The more general term, service, is used when the number of actual servers and their structure is unimportant. Clients are programs which use a service.

caller, callee: The caller is the node which invokes an RPC. The callee is the destination of the call, the node on which the body of the remote procedure executes. In the literature the caller is often termed the client and the callee the server, but this is confusing when a server invokes an RPC on one of its clients.

---

[10] R. C. B. Cooper. "Pilgrim: A Debugger for Distributed Systems." *Proc. of the 7th International Conference on Distributed Computing Systems*, Berlin, September 1987, IEEE Computer Society Press, Washington D.C., p. 458.

The program fragments are presented in a simple Algol-like pseudo-code. Assignment is represented by "←". When describing the parts of the implementation of Pilgrim the names of some procedures and data structures have been changed from those actually used in the source to make the meaning clearer.

1.5 Structure of this dissertation

Chapter 2 presents a model of the debugging process and lists the requirements a debugger for concurrent and distributed programs ought to meet.

Chapter 3 discusses the facilities that should be provided for debugging RPC and details how these can be implemented.

The next three chapters cover different aspects of debugging concurrent processes. Chapter 4 examines the problem of transparent debugging and in particular what it means for a breakpointing mechanism to be transparent. This motivates the breakpointing method developed in Chapter 5. There are other aspects to the debugging of concurrent processes besides breakpointing; these are discussed in Chapter 6.

Chapter 7 looks at the problem of debugging clients of a shared network service without interfering with the service's other clients.

Chapter 8 looks at a wide range of issues related to overall debugger design and structure. This chapter is the most specific to Pilgrim and Concurrent CLU.

Conclusions appear in Chapter 9.

For the sake of continuity I discuss related work by other researchers at the end of the chapter to which it is most relevant.

Chapter 2

Principles and Requirements

Debugging is "...the process of isolating and correcting mistakes in computer programs".[1] In this chapter I define what I mean by debugging, and set out the requirements I expect a debugger for concurrent and distributed programs to fulfill.

2.1 The debugging model

Debugging usually begins after the existence of a bug is known, perhaps as a result of program testing or complaints from end-users. It proceeds by examining the internal state of the program at critical points, by making further test runs, and by reasoning about the operation of the program until, we hope, the cause of the bug is discovered. After the cause is known the programmer must come up with a modification to the program which corrects the bug. Sometimes the correction is apparent as soon as the cause of the bug is found. At other times a substantial redesign of the program is necessary.

Exactly how the debugging activity proceeds can vary greatly, depending upon the programmer, the structure of the program, and the nature of the execution environment in which the program is run. However in all debugging methods some means is required for observing the internal states of the program in order to understand how it operates, and how its operation differs from what the programmer intended. This is the purpose of most tools which have been designed for debugging. These tools range from write statements inserted by the programmer, and post mortem dumps, to interactive source-level debuggers.

[1] M. S. Johnson. "A Software Debugging Glossary." *SIGPLAN Notices*, **17**(2), February 1982, p. 53.

An interactive debugger is the most useful of these debugging tools. Dumps generate great quantities of information which the programmer must sift through to find the parts of interest for a particular debugging run. As memories and programs have grown larger, the computer resources required for a dump can be excessive. An interactive debugger is more selective; it only displays information in response to a user query. Inserting write statements in the program usually requires recompilation and re-execution. Compilation can be time-consuming, and re-executing the program with identical inputs is difficult—especially for distributed programs. A good interactive debugger permits any component of a program's state to be examined at any point, without requiring unreasonable foresight on the user's part.

2.2 Requirements for a debugger

What features must an interactive debugger have to support the source-level debugging of concurrent and distributed programs in the target-environment?

Debugging at the level of the source language

The debugger must display the program state to the user in the terms of the high-level language in which the program is written. Little benefit is gained from high-level programming languages if the user must descend to the machine-language level for debugging. Machine language debugging requires the user to understand the instruction set and hardware architecture of the processor, and how the language is implemented on the architecture. Any optimizations carried out by the compiler will make this task formidable, even for a highly motivated and experienced systems programmer.

The situation is even more difficult when the language supports multiple processes and remote communication. If these features are not supported by the debugger, the user must be able to find the stacks of multiple processes stored in memory, and know how the language's view of a process relates to the operating system primitives provided for multiprocessing. She must understand how the language's remote communication facilities are implemented upon some transport protocol, how data objects in the language are represented in network packets, and some details of the interface to the network device.

Debugging user-defined abstractions

In a language which encourages data abstraction, such as Concurrent CLU, we can go a step further and provide debugging at the *abstraction level.*

For instance, the table cluster implements an unordered collection of uniquely identified elements. That it is implemented as a hash table is irrelevant to most programmers. Imagine a programmer who uses this cluster and wishes to display a table while debugging. She is likely to be interested in the values stored in the table, but not in the particular way that they are organized into a hash table. Fortunately there is a convention in Concurrent CLU that each cluster provide a print procedure which displays the abstract state of the object, and all the important built-in types provide one. The debugger should use these print operations to display objects to the user. Even in languages lacking such print procedures, the debugger should provide a simple way to invoke user-written procedures which display program data.

The debugger should allow the abstraction level to be chosen by the user. If, in the course of debugging the previous example program, the implementation of table is suspected as the source of a bug, there must be a way to display the concrete state of a table—the hash chains and the empty hash slots for instance. Also the debugger should gracefully degrade to lower levels of abstraction when the programmer of a cluster has not provided print operations. (As a minor point it is also useful for system implementors if the debugger provides some escape hatch to access the machine levels of the implementation.)

Debugging in the target environment

It is common for high-quality debugging support to be provided during program development but removed when the program is shipped to the target environment.

There are many reasons for this. Not the least is an overdeveloped and misguided sense of efficiency in many programmers. Any computer resources devoted to debugging support or checking are perceived as not performing useful work as soon as the program has been "debugged". This is wrong because the program almost certainly contains bugs at the end of the development process. Often crucial aspects of the real environment will have been

neglected during the the problem specification or design phases. These problems will be detected only when the program reaches the target environment. The consequences of bugs once the program is executing on real data can be disastrous. In most cases performance is not as important a criterion as correctness. If better performance is required then it is more sensible to do performance analysis to identify and optimize bottle-necks than to remove debugging support.

Nevertheless, debugging can incur a significant performance cost. Many compilers perform optimization of the object code, to obtain acceptable performance on the target instruction set. Frequently this code optimization must be turned off to enable the debugging support. Equally, the way in which debugging support is implemented can seriously degrade program performance, even when the program is not being debugged. Debugging support and efficient object programs should not be so incompatible. The debugger should be able to cope with many of the simpler code optimizations, with the help of compiler-produced tables describing the source-object mapping.[2] More wide ranging optimizations remain a problem, but often these optimizations can be modified to assist debugging with a small loss of efficiency.[3,4]

The debugging support provided should affect the speed and size of the object program as little as possible. The overhead should be small enough when not actively debugging that the user is not encouraged to disable debugging support.

Debugging multiple processes transparently

It is important for the program to behave the same way when it is being debugged as when it is executing normally. In other words, the presence of the debugger should be *transparent* to the execution of the program—except of course for the effects of debugger commands which are intended to alter the program state, e.g. variable assignment.

---

[2]J. Hennessy. "Symbolic Debugging of Optimized Code." *ACM Trans. on Programming Languages and Systems*, 4(3), July 1982, p. 323.

[3]P. T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs.* Ph.D. Dissertation, Technical Report CSL-84-5, Xerox PARC, Palo Alto, Calif., May 1984.

[4]P. T. Zellweger. "An Interactive High-Level Debugger for Control-Flow Optimized Programs." *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, August 1983, p. 159 (published as *Software Engineering Notes*, 4(4) and SIGPLAN Notices, 8(8)).

At the least, the debugger should not cause the program to execute computations that are not specified by the program text. However transparency is more difficult when programs contain multiple processes. Most concurrent and distributed programs specify a range of computations which are permitted to occur for the same inputs; i.e. they are nondeterministic. A different one of those permitted computations may occur when the program executes under control of the debugger than would have occurred in its absence. This violates transparency and is undesirable.

Different computations of a sequential program can occur if the memory layout is different when executing under the debugger, or if the debugger initializes all variables to a default value but the normally executing program did not. These differences can be avoided.

With concurrent and distributed programs, transparency is much more difficult to preserve. One of the main ways of using a debugger is to slow down or interrupt the execution of the program so that its state can be examined, for instance using breakpoints. If as a result the program perceives time passing more quickly in the rest of the environment, or if different processes execute at different rates, a faithful execution will not occur. As a result, the symptoms of the bug under study may disappear or another bug may reveal itself and confuse matters. The correctness of a concurrent program should rarely depend on its rate of execution, but faulty programs may contain timing errors of just this kind.

The debugger should preserve, as far as possible, the relative ordering of events in a concurrent program that would have occurred had the debugger not been present.

Debugging in the presence of shared services

A distributed program does not exist in isolation, but typically uses public servers shared with other users and programs in the distributed system. A conflict arises between fulfilling the requirements of transparent debugging, and avoiding any interference with other concurrent users of the system. To preserve the debuggee's view of time, the processes with which it interacts on a public server could be slowed down or interrupted at appropriate points. But other clients or users of that server would be co-opted into the debugging process, and have their work delayed or interrupted.

I will argue later that good solutions to this problem involve the co-operation of the server and the debugger. The debugger should provide information about the debuggee program to any server that program is using. With this information the server can maintain the transparency of the debuggee, while continuing to provide service to other clients.

Integrated debugging of concurrent, distributed, and sequential code

All the above requirements should be satisfied by a single integrated debugger. If several separate tools provide different aspects of debugging support, the user must perform many context switches when finding the cause of a bug. For instance, the first indication of a bug may come from the results returned from an RPC. Those results might have been computed by a piece of sequential code. However the ultimate cause of the bug could be a serialization error with another process in updating a shared object. Since all these pieces of the program were written in the same source language (and perhaps in the same source file) it is convenient to debug them from the same debugger.

Conclusion

I have discussed the requirements of a debugger for concurrent and distributed programs. The remaining chapters present debugging facilities and implementation techniques to meet these requirements. The next chapter describes RPC debugging facilities. Later chapters are concerned with debugging concurrent processes, and debugging clients of shared network services. Finally, the overall structure of the Pilgrim debugger, which embodies the ideas in this chapter, is presented.

Chapter 3

Debugging Remote Procedure Calls

This chapter focuses on the facilities needed for debugging RPC. The debugging of concurrent processes is discussed in later chapters. The following section presents the design of a set of debugging facilities for RPC, motivated by the requirements of Chapter 2. Some of these facilities can be implemented easily using techniques from sequential debuggers. The rest of the chapter is about how to implement the more difficult aspects. First, a general implementation scheme is developed which is suited to many different RPC systems. Next, the actual Pilgrim implementation is described in some detail, while a hypothetical implementation for another RPC system is considered. Finally other work on debugging RPC is discussed.

3.1 The design of RPC debugging facilities

The RPC debugging facilities must be based on the semantics of RPC in the programming language concerned. The debugger should be able to display the meaningful program state associated with RPCs and allow observation of state changes. RPCs are intended to be very similar to local procedure calls. The debugger should preserve this impression by offering the same debugging facilities for the aspects common to both local and remote calls.

The call history and variable access

Debuggers for sequential programs can display the call history, and provide access to and modification of local variables within each currently active procedure. The call history

(or call stack) is a sequence of procedure activations (or stack frames) beginning with the currently executing procedure and followed by the procedure which called it and so on. Debugger syntax for variable access often uses the programming language's name visibility rules applying at the current execution point. This may include access to variables of textually enclosing procedures for languages with Algol scoping rules. Additional syntax may be provided to access out-of-scope variables on the call stack.

These same facilities should be provided for debugging RPC. Call histories should be displayed across node boundaries, and variable access should be possible regardless of the node of the distributed program on which the variable resides. The only exception to this is when the user of the debugger does not have the authority to debug some node, for example when her program invokes a remote procedure on a public server. This and other issues concerning the wider distributed system are addressed more fully in Chapters 7 and 8.

The implementation of the call history is largely independent of the user interface or the exact semantics of these features. The most difficult aspect is associating the caller and the callee of an RPC. This is described later in the chapter.

The failure semantics of RPC

The failure semantics of RPC differ from those of local procedure call. A local procedure either returns successfully or fails. If it fails the entire program fails too. In contrast, when an RPC fails or the node which is handling an RPC fails the calling node may be informed of the failure, allowed to take some recovery action, and then continue executing. Actually this is a simplification. In a language with an *exception* facility there is less of a distinction between local and remote exceptions and failures. Events which would otherwise result in abnormal program termination, e.g. an array bounds error, may instead raise an exception which the calling procedure can handle.

Most sequential debuggers catch program failures and report them to the user. If the language has an exception facility, the debugger usually allows breakpoints to be placed on the occurrence of particular exceptions so that the debugger gains control before the

program handles the exception. These facilities should be provided for debugging RPCs so that the user can find out if a call has failed.

The kind of exception or failure returned may not be enough to determine its actual cause. The most common symptom of failure is simply lack of response. This could be due to the call packets being lost, the reply being lost, the callee node crashing, or the remote procedure simply taking longer to execute than the caller expected. Determining which case has actually occurred can be important for debugging. For instance, knowing whether the remote procedure has been executed partially, completely, or not at all is crucial to understanding the future behaviour of the program.

Some of these cases can be differentiated using information available from the communication protocol and the RPC implementation. But RPC systems that use specialized lightweight protocols, such as Mayflower RPC, cannot provide much information. So it may be worthwhile to retain extra information about recently completed or failed calls in order to improve debugging support. Of course, if a serious communication failure has occurred the debugger may not be able to contact the callee node to obtain this information. Support for debugging failed RPCs is described later.

The argument passing semantics of RPC

Different argument passing semantics are used for RPC, usually based on copying. Argument objects are copied into the network packets, a process termed *marshalling*. At the callee the objects are *unmarshalled* into primary memory. The same process occurs for result objects and possibly for modified argument objects when the remote procedure returns. Misunderstandings over these semantics are the source of many errors. The most straightforward way to debug these problems is to examine the objects before and after transmission to study the effects of marshalling.

Some RPC systems permit the programmer to specify how objects of particular types are transmitted. For instance, in Concurrent CLU the programmer writes marshal and unmarshal procedures for user-defined types which she wishes to be transmissible in RPCs. Since these are simply local procedures they may be debugged in the normal way by

setting breakpoints and examining data objects. Thus no new implementation techniques are needed for debugging the passing of arguments in RPCs.

Invoking RPCs from the debugger

Many debuggers permit local procedures in the debuggee to be invoked directly from the debugger. A similar feature should be provided to invoke remote procedures; however the user has to specify both the caller and callee nodes. The caller node must be given to provide the execution context in which the RPC arguments are evaluated and marshalled, while the callee node specifies where the body of the remote procedure will be executed. This distinction does not arise for invoking local procedures since clearly the same node is both caller and callee.

The implementation of remote procedure invocation is very similar to the invocation of local procedures in sequential debuggers.

3.2 A scheme for implementing RPC debugging support

This section outlines how the call history and support for RPC failures, the most difficult aspects of the debugging support just described, may be implemented. The implementations for specific RPC systems are described in more detail in the remaining sections of this chapter.

Implementing the call history

The debugger must be able to display the call stack, given the name of a process. The outline of an algorithm for doing this is shown Figure 3.1. The implementation of get_top_frame requires assistance from the operating system, including access to the current register set of the process, and is discussed in Chapter 6 on page 60. The procedure is_local_procedure is very dependent on how the RPC and process support is structured. It might be implemented by testing whether the procedure frame, F, was invoked from

```
P ← the name of the process
F ← get_top_frame(P)
loop
    display(F)
    if is_local_procedure(F) then
        if is_bottom_of_local_stack(F) then
            exitloop
        else
            F ← get_local_caller(F)
        endif
    else
        F ← get_remote_caller(F)
    endif
endloop
```

Figure 3.1: Displaying the call history across multiple nodes

the RPC runtime support, or from the code which implements local process creation. Alternatively, in a language where local and remote procedures have different types such as Concurrent CLU, type information produced by the compiler could be used to differentiate local and remote procedures.

The procedure get_local_caller can be implemented in the same way as in a sequential debugger. Typically the return program counter and the return stack frame pointer are stored in F. The program counter can be looked up in compiler-generated tables to find the calling procedure's name and type. The situation is more complicated when some local procedures have been optimized in-line.[1] These three procedures, get_top_frame, is_local_procedure and get_local_caller, can be implemented with information on the same node as F, or with information generated by the compiler.

The procedure get_remote_caller requires information stored on both the caller and callee nodes. The caller node must be located, and then the call stack which initiated this RPC found. The RPC runtime support on the callee will know a network address at the caller node, otherwise it could not return the results of the RPC. This reply address may be sufficient for the caller node to uniquely identify the caller process which is suspended waiting for the results (i.e. a unique reply address is allocated for each RPC). More likely, many RPC replies will be multiplexed on a single address. In this case the RPC protocol will include a *call-ID* which is generated by the caller and supplied with the call message.

[1]See p. 137 of P. T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs.* Ph.D. Dissertation, Technical Report CSL-84-5, Xerox PARC, Palo Alto, Calif., May 1984.

This call-ID is remembered by the callee and returned in the reply message. If there were no such call-ID there would be no way for the RPC runtime support on the caller to de-multiplex the replies and find the correct process to return the results to. (In many RPC protocols the call-ID is also used to detect duplicate packets.)

More precisely, get_remote_caller can be implemented using the following two procedures:

```
reply_address, call_ID ← locate_RPC_source(callee_frame)        executed on the callee
caller_frame           ← get_RPC_caller(reply_address, call_ID)  executed on the caller
```

The procedure locate_RPC_source determines the reply address and the call-ID for the RPC belonging to the stack frame of the callee. These two values are supplied to the procedure get_RPC_caller which is executed on the caller node, and returns the stack frame which initiated the RPC. This procedure can be implemented in several ways. A message containing the call-ID could be sent to the reply address and a message identifying the caller_frame would be returned. Essentially the RPC protocol would be extended by these two messages. Alternatively get_RPC_caller might be implemented directly as a remote procedure itself.

The inverse of these two procedures should be implemented:

```
call_address, call_ID ← locate_RPC_destination(caller_frame)     executed on the caller
callee_frame          ← get_RPC_callee(call_address, call_ID)     executed on the callee
```

This could be used as follows. Suppose the user displays the stack of a process and discovers that it is in the midst of an RPC to another node. These two procedures could be used to locate the node handling the RPC, and display the current state of the callee process. The information required to implement locate_RPC_destination will be maintained by the caller node so that the call message can be retransmitted in the case of lost packets. (For RPC protocols which do not attempt to recover from lost packets, such as Mayflower's *maybe* protocol, this information might not be maintained.)

If the RPC runtime support is implemented correctly it must maintain the information necessary to associate caller and callee stacks. However the information may not be in a form which is convenient or efficient to search. As will be seen in Section 3.4, extra tables for accessing this information can be worthwhile, and do not significantly affect the performance of the RPC system.

Implementing debugging support for RPC failures

The debugger should be able to catch RPC failures or set breakpoints upon them. This can be implemented in the same way that failures are caught by sequential debuggers. For instance, breakpoints can be set at the places in the RPC runtime support where RPC failures are detected, and a trap made to the debugger.

For ambiguous failures, such as communication failures, the debugger could endeavour to find out more information, in particular whether the remote procedure at the callee was actually invoked, or whether it is still executing. The latter case is termed an *orphan*. It arises when the caller ceases to wait for a reply, believing that the callee has crashed, or when the caller itself crashes. In either case the callee process continues executing the RPC, becoming an orphan. Orphans are a source of bugs because they are largely invisible but continue to modify program state. Again the debugger should be able to provide information about orphans. The procedures defined in the previous section could be used but these would not always be able to help. They rely on information which must be present while an RPC is in progress. Once the RPC has failed the RPC runtime support is free to delete the information.

A cheap solution is to maintain a small buffer containing the outcome of the last few calls. The information retained can be limited to the call or reply address, the call-ID, and an indication of whether the call failed or succeeded.

3.3 The Mayflower RPC system

In order to understand the description of RPC debugging support in Pilgrim, it is necessary to know something of the Mayflower RPC implementation. In what follows some details have been omitted and simplifications made.

Syntax and semantics

The Concurrent CLU programmer is provided with the choice of two sets of semantics for RPC. The *exactly-once* semantics makes every effort to ensure the remote procedure will

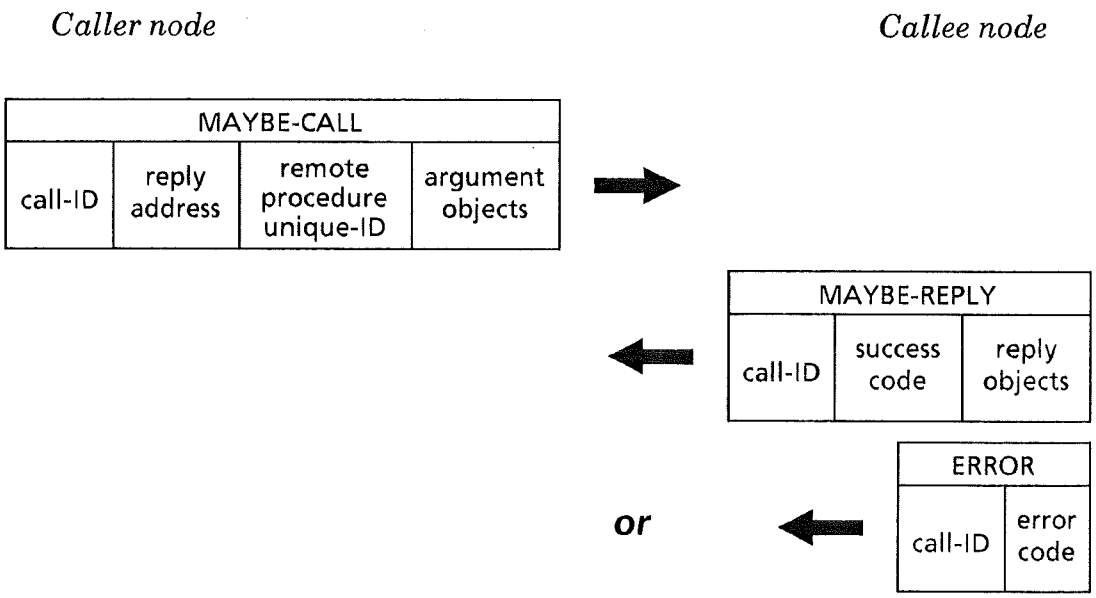

Figure 3.2: The *maybe* protocol

be executed once, despite communication failures and congestion at the callee. But if the callee node crashed during the call the procedure could be executed completely, partially, or not at all. If the callee crashed and restarted during the call the procedure could be executed more than once. If necessary this is avoided by choosing a different port number when restarting, to ensure that stale calls cannot be received and handled twice.

The *maybe* semantics makes no attempt itself to recover from communication failures or crashes, leaving such actions to the programmer. If a *maybe* RPC returns successfully to the caller the remote procedure will have been executed once. In all other circumstances the procedure could have been executed completely, partially, or not at all. A timeout value may be specified on a *maybe* call causing an exception to be raised if the RPC does not return in time.

The *exactly-once* semantics are used by most application programmers who are not concerned with node crashes. The *maybe* semantics are used by more experienced systems programmers to build higher-level protocols with the desired reliability properties such as atomicity or recoverability. The *maybe* semantics are also suitable in cases where timeliness is more important than guarantees of reliability (e.g. when obtaining the time from a network time server).

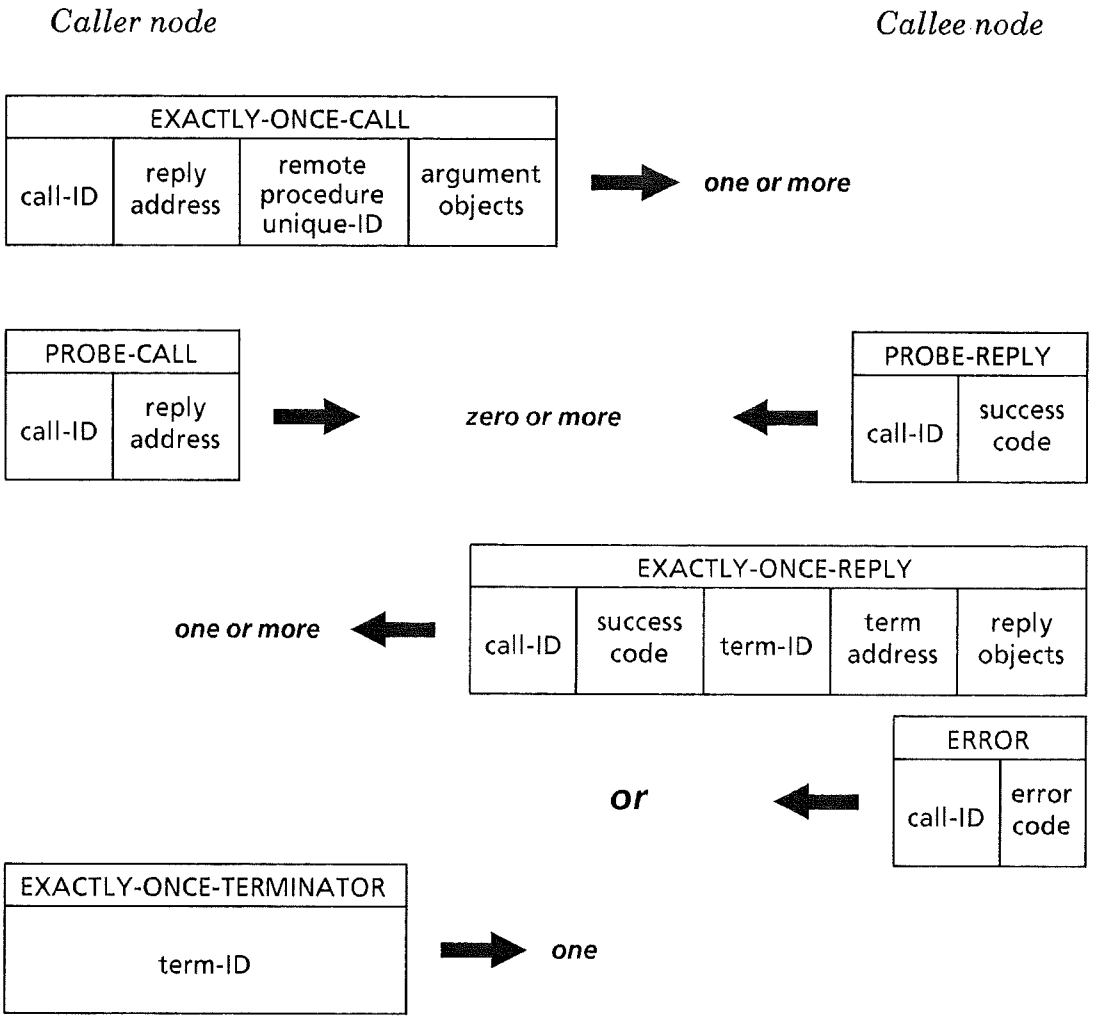

Figure 3.3: The *exactly-once* protocol

RPC binding in Concurrent CLU is performed at run-time. Binding is the process of determining for each remote call the callee node which will handle it. A node of a Concurrent CLU program can bind to all the remote procedures offered by a specified address. Alternatively the callee node's address can be explicitly supplied when the remote procedure is invoked.

The RPC protocol

The two sets of semantics are implemented by two message protocols. The *maybe* protocol is shown in Figure 3.2. A successful call consists of a single call packet sent by the caller

node and a single reply packet sent by the callee node. The remote procedure unique-ID shown in the call packet is a 62 bit number generated by the compiler which uniquely identifies the name and type of the remote procedure.

The *exactly-once* protocol is shown in Figure 3.3. In the ideal case a successful call consists of a call packet sent by the caller node, a reply packet sent by the callee node, and a termination packet sent by the caller to acknowledge the reply. The call packet is retransmitted until it has been received by the callee. Probe packets are sent at intervals until the callee sends the reply packet. The reply is retransmitted until a terminator is received from the caller.

This description does not include RPCs for which the arguments or results are too large to fit in a single network packet. In such cases a multi-packet protocol is used, which incorporates flow control and uses a different network address in the receiving node.

The caller RPC runtime support

Each outgoing call in progress is described in the *reply block* list. A process wishing to do an RPC constructs a reply block before assembling the call packet and transmitting it to the callee. The caller process then waits on a semaphore until a reply or error packet is received. A pool of *receiver* processes listens for packets. When one is received the reply block with the corresponding call-ID is found and the waiting process notified.

For *maybe* calls the caller process waits for a single reply or error packet, or for a timeout to expire. For *exactly-once* calls the caller process loops, retransmitting the call packet and doing probes, until a successful reply is received or a non-recoverable error occurs. After a successful reply a single terminator packet is sent to acknowledge receipt of the results.

The callee RPC runtime support

Each incoming *exactly-once* call is described in the *in-progress* list. In addition there is an entry in the *terminator block* list for every *exactly-once* call which is waiting for a terminator packet. There is also a pool of receiver processes, as in the caller.

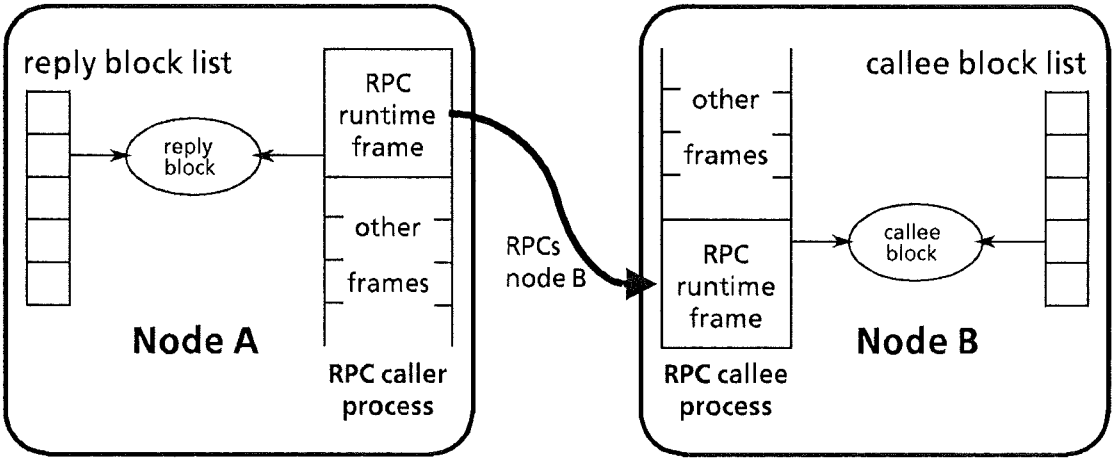

Figure 3.4: Associating the caller and callee processes in Pilgrim

When a receiver receives a *maybe* call packet it simply invokes the procedure (thus taking the role of a callee process) and returns the results. When an *exactly-once* call packet is received an entry is added to the in-progress list. If an entry for this call-ID already exists the call packet is a duplicate and is discarded. Otherwise the procedure is invoked, then a terminator block is constructed, and the results returned in a reply packet. The receiver process then loops, alternately waiting on a semaphore and retransmitting the results, until a terminator packet has been received. Finally the call is removed from the in-progress list. Some other receiver process, when it receives a terminator packet, will find the corresponding terminator block and notify the receiver process that is transmitting the results.

3.4 The Pilgrim implementation

In this section I describe how I applied the general implementation scheme developed above to provide RPC debugging facilities for Concurrent CLU in Pilgrim. Figure 3.4 illustrates some of the debugging support described next.

Associating the caller and callee processes

Section 3.2 describes two procedures to be implemented on the caller node:

```
caller_frame              ← get_RPC_caller(reply_address, call_ID)
call_address, call_ID ← locate_RPC_destination(caller_frame)
```

The procedure get_RPC_caller is implemented using the reply block list. Each reply block contains the call-ID of the RPC, and the semaphore on which the caller process is waiting. I added an extra field to the reply block, pointing to the caller process, from which it is straightforward to obtain its top stack frame. This is strictly unnecessary since the caller process could be found by examining which process was waiting on the semaphore, using the debugging support for concurrent processes described in Chapter 6. However, doing so would make display of RPC call histories in the debugger extremely expensive.

A variable in the top stack frame of the caller process points to the reply block for that call. The locate_RPC_destination procedure is implemented by accessing this variable to obtain the call-ID and the network address of the callee. This required adding another field to the reply block containing this network address.

Two further fields were added to the reply block to assist debugging. One describes the current state in the RPC protocol. This defines when the other fields in the reply block have meaningful values and may be safely accessed by the debugger. The remaining field contains the remote procedure unique-ID.

Similar techniques are used to implement the two debugging procedures on the callee:

```
reply_address, call_ID ← locate_RPC_source(callee_frame)
callee_frame              ← get_RPC_callee(call_address, call_ID)
```

Each receiver process owns a *callee block* which already contained most of the necessary information such as the reply address and call-ID for the call being handled. I added a pointer to the receiver process, and collected all the callee blocks in a list. (This list is updated infrequently when a new process is added to the receivers pool.) The procedure get_RPC_callee searches this list. The procedure locate_RPC_source accesses the callee block via a variable in the stack frame.

Recently completed calls

I implemented a history buffer as described at the end of Section 3.2. This records the outcome of the ten previous successful calls and the ten previous failed calls, along with their associated call-IDs and node addresses. If either of the procedures get_RPC_callee or get_RPC_caller is unable to find a process associated with the RPC, the history buffer is searched to determine if the call has been recently processed.

Performance

Acceptable performance of the RPC debugging support is crucial to the achievement of target environment debugging. The normal execution of RPCs must not be slowed down too much, otherwise users will disable the debugging support in the target environment. Of less importance is the cost of the debugging facilities to the debugger. Paradoxically it is the cost of debugging support when not debugging that is important.

An informal analysis of the Pilgrim implementation reveals that the critical path of an RPC has been increased a constant amount by the extra assignments to the reply block and the callee block, and by the updating of the history buffer. Performance measurements demonstrate that the total time for an RPC has been increased by about 200μs. The time for the quickest possible RPC (a *maybe* call of a remote procedure having no arguments, no results, and a empty body) increased from 15600μs to 15800μs, a slow-down of 1.3%. On more typical RPCs the slow-down is proportionally much less. (Further changes to the Concurrent CLU runtime system, described in Section 5.2, add a further 1500μs. This is discussed fully in that section.)

The performance figures were obtained using 10,000 repetitions of an RPC between two MC68000s connected to the same Cambridge Ring. Some other traffic was present on the Ring, but this was not significant since none of the thousands of *maybe* call and reply packets was lost. The 20ms processor clock on the caller node was used to measure the elapsed time for the entire sequence of RPC invocations. The 20ms clock is accurate in absolute terms to only 10%, however since the clock has negligible drift, relative timings performed in close succession are very accurate.

(The base figure of 15600μs is approximately twice the figure quoted in Hamilton's thesis.[2] Since its publication a number of changes have been made to the RPC system to enhance its capabilities including the addition of user-defined marshalling, and use of a larger heterogeneous network address in the RPC protocol. In addition, two of Hamilton's optimizations have been removed. To ease software maintenance, the direct support for RPCs in Mayflower's network device driver was removed, introducing an extra process switch on the critical path. And Hamilton's improved control program for the Ring interface processor, the *Mace*, was not adopted in our research environment.)

Discussion

The small performance cost of these debugging facilities means they can be permanently enabled. If the facilities were expensive and were enabled only after connection to a debugger, information about in-progress RPCs which were begun before the debugging session would not be available. Further, RPCs during a debugging session would be much slower than normal, significantly changing the relative timing of processes in the program when under control of the debugger.

3.5 The generality of the implementation techniques

How applicable are these RPC debugging techniques to other RPC implementations?

Nelson's Emissary RPC design

The most detailed, publicly available description of an RPC system is *Emissary* which appears in Nelson's thesis.[3] His proposed implementation maintains all the information required for debugging in a form which is readily accessible.

---

[2]K. G. Hamilton. *A Remote Procedure Call System.* Ph.D. Dissertation, Technical Report 70, Computer Laboratory, Cambridge University, December 1984.

[3]See p. 114 of B. J. Nelson. *Remote Procedure Call.* Ph.D. Dissertation, Technical Report CMU-CS-81-119, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn., May 1981.

The call-IDs transmitted by Emissary in every RPC packet comprise the caller's node address, the address of the caller process object, and the value of a per-process counter. The process object for a caller process contains the network address of the callee node, and the current state of the RPC protocol. On the callee node, a list of RPC callee processes is maintained describing the state of the RPC protocol and the call-ID for each RPC being handled. Nelson presents several algorithms for detecting and eliminating orphans. These algorithms could be used by the debugger to report orphaned calls to the user.

Other RPC systems

A similar implementation scheme will work for most RPC implementations, and indeed should be easier to realize than for Mayflower RPC. In general the more failure and orphan detection provided by an RPC system, the more information useful for debugging will be maintained by the debugger. The main difficulty for Pilgrim was to find information about *maybe* RPCs for which the RPC mechanism maintains little state information.

3.6 Related work on debugging RPC

The only other source level debugger I am aware of which provides support for RPC is the Argus debugger.[4] An Argus program is composed of logical nodes, termed *guardians*, which are the units of failure and recovery. In general a single instance of the debugger is used to debug only one guardian, but a command is provided to follow the call history of a process through multiple guardians, and is thus similar to the call history facilities provided in Pilgrim.

An RPC in Argus, termed a *handler call*, is an atomic transaction. A handler call either executes exactly once and commits, or aborts in which case the effects of any partial execution of the handler are completely invisible. So there is no need for the Argus debugger to provide special facilities for debugging failures arising from lost packets, incomplete calls, or orphans. No details of the Argus debugger have been published.

[4] P. R. Johnson. *Personal communication*, July 25, 1986.

Chapter 4

Transparent Debugging and Breakpoints

This chapter examines the problem of transparent debugging when breakpointing a distributed program. This will motivate the development in the next chapter of a transparent breakpointing mechanism. With such a mechanism, the program state which the user can observe when a breakpoint is triggered is the same state that would have occurred in the absence of the breakpoint. When the program is resumed, it continues as if the breakpoint had never happened. Achieving this is difficult when the program consists of concurrent processes, some of which share access to memory, while others are distributed on many different nodes of a computer network.

The distributed program is assumed to execute in isolation. The problems arising from other concurrent users in a distributed computing environment are discussed in Chapter 7.

4.1 Transparently examining program states

If there is one defining feature of interactive debuggers it is the ability to observe the internal state of the program at practically any point during its execution. A common facility for this is the breakpoint, in which the program is interrupted when a specified event occurs. At this point the user is able to examine the states of variables and then resume the program which continues execution as if it had never been interrupted. Alternatively the user might change the values of variables and alter the flow of control to experiment with some changes to the program.

Different kinds of events can trigger a breakpoint. A code breakpoint is triggered by the execution of a particular statement in the program, a data breakpoint by the access

or update of a data object, and an exception breakpoint by the raising of a particular exception. Most debuggers also gain control of the program when some fatal execution error occurs.

Production rule breakpoints

Some researchers have generalized breakpoints to recognize combinations of events, e.g. to detect when the variable v is set to the value 3 after procedure p has been invoked but before it returns. In Bruegge's[1,2] work, events are specified using *generalized path expressions* which are essentially production rules over an alphabet consisting of all possible program events, augmented with boolean predicates on the program state. Baiardi, De Francesco, Latella and Vaglini[3,4] use their *behavioural specification* language to specify the expected ordering of certain program events in their debugger. When the program fails to obey this ordering, a breakpoint is triggered. Their specifications appear to have the same power as predicate path expressions. Production rule breakpoints are very useful for concurrent and distributed debugging. In a program with multiple asynchronous threads of control, points of interest to the user are unlikely to correspond to the triggering of a single code or data breakpoint.

Instantaneously halting multiple processes

To achieve transparency, it must appear as though all of the processes in the program have been instantaneously suspended when a breakpoint is triggered. Problems will occur if some processes continue executing while others are suspended at a breakpoint. Firstly, the still-running processes may alter the state which is being observed. Secondly, the running

---

[1]B. Bruegge and P. Hibbard. "Generalized Path Expressions: A High Level Debugging Mechanism." *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, August 1983, p. 34 (published as *Software Engineering Notes*, **4**(4) and SIGPLAN Notices, **8**(8)).

[2]B. Bruegge. *Adaptability and Portability in Symbolic Debuggers*. Ph.D. Dissertation, Technical Report CMU-CS-85-174, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn., September 1985.

[3]F. Baiardi, N. De Francesco, and G. Vaglini. "Development of a Debugger for a Concurrent Language." *IEEE Trans. on Software Engineering*, **SE-12**(4), April 1986, p. 547.

[4]N. De Francesco, D. Latella, G. Vaglini. "An Interactive Debugger for a Concurrent Language." *Proc. of the 8th International Conference on Software Engineering*, London, August 1985, IEEE Computer Society Press, Washington D.C., p. 320.

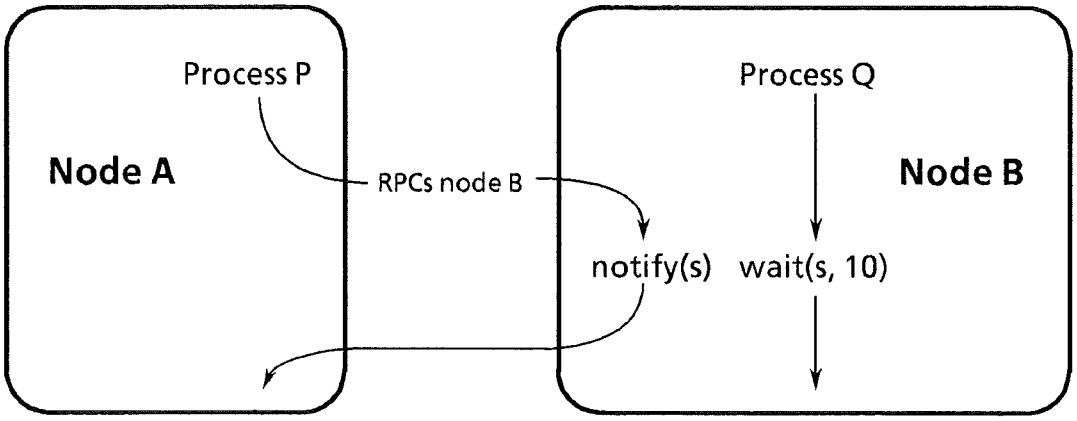

Figure 4.1: Triggering a breakpoint in a distributed program

processes, and the suspended ones after they have been resumed, may carry out a different computation to the one that would have occurred had the breakpoint not been triggered. Thirdly, if the programmer wishes to alter the state of the program or cause a different computation to occur, she may be unable to do so if some processes are still running.

An example of this which almost any person who has written a distributed program will have experienced is how timeouts expire while part of a program is suspended. Consider the specific example in Figure 4.1. Process Q on node B is waiting on the semaphore s with a timeout of 10 seconds. Also on node B is the body of a remote procedure which notifies s. Process P on node A calls this remote procedure. If a breakpoint is triggered and the processes on node A are halted before those on node B then there is a chance that Q will "see" that P has halted. That is, its semaphore wait may timeout whereas if the breakpoint had not been triggered, process Q might have been notified by P instead.

In reality all processes need not be halted simultaneously at the instant the breakpoint is triggered. Rather, any process may continue executing as long as it neither notices that any processes have halted nor alters any state accessible to them. In addition we must ensure that the values to be input to the program after it resumes are preserved unchanged.

To justify this claim, a model of concurrent and distributed computations is developed in Section 4.2, and used in Section 4.3 to determine when processes must be halted to achieve transparency.

Alternatives to breakpoints

A important alternative to breakpoint debugging is to log program events while running the program without user intervention. This log can be examined by the user at leisure. If the log is sufficiently detailed the original execution of the program can be simulated or replayed using the log. Logging and replay are difficult or impossible to do both transparently and efficiently (for target environment debugging). The arguments for this are set out in Section 5.4.

4.2 Relative ordering of events in a distributed program

Lamport[5] has noted that the order in which events occur in a distributed system is a partial order (rather than a total order). The implementor of a distributed computation is thus free to choose any total ordering which is consistent with the partial ordering of events in that computation. This is of use when halting a distributed program. Without this observation we might imagine that when a breakpoint is triggered by an event in one process, all the processes in the program must be halted before they begin their next event. In fact we need not halt a process until its next interaction (including indirect interactions) with a halted process.

Lamport's analysis concerns systems of concurrent processes which communicate only by sending and receiving messages. The remainder of this section presents a summary of his work, and then applies it to the kind of distributed programs this thesis is concerned with—particularly processes which communicate via shared memory. This will motivate some more precise criteria for achieving transparent halting.

Lamport's model of distributed systems

In Lamport's model, a distributed system consists of a number of processes which communicate only by sending and receiving messages. Each process consists of a sequence of

[5]L. Lamport. "Time, Clocks and the Ordering of Events in a Distributed System." *Comm. of the ACM*, **21**(7), July 1978, p. 546.

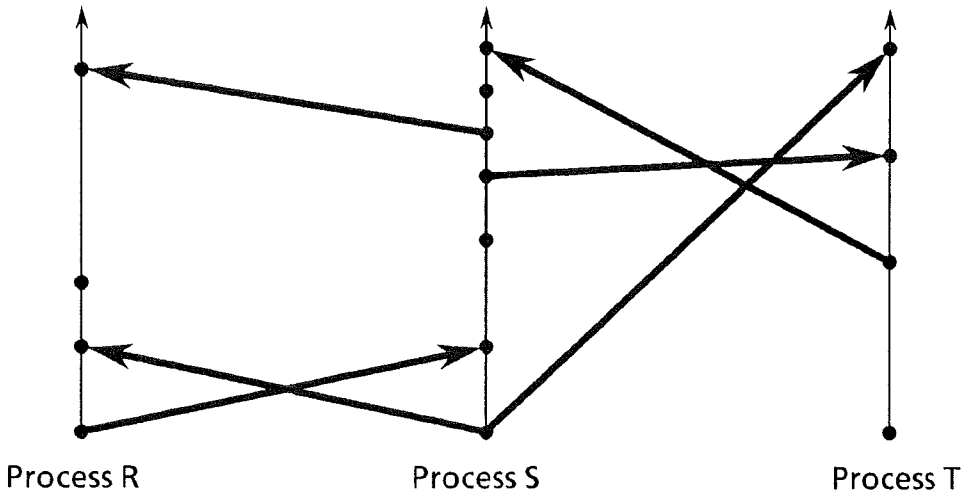

Figure 4.2: A space-time diagram depicting an event ordering

events. Sending a message and receiving a message are events, and other events include the execution of machine instructions or language primitives such as variable assignment.

Lamport defines a "happened before" relation on the events in such a system (written as "$\rightarrow$") which satisfies the following three conditions:

1. If a and b are events in the same process and a comes before b, then $a \rightarrow b$.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$. Thus $\rightarrow$ defines a partial ordering on all the events in a distributed system, and the events within one process are totally ordered.

Lamport introduces a *space-time diagram* to illustrate his model (see Figure 4.2). Each vertical line represents the execution of a process, and the dots represent events, with time advancing up the diagram. The thick arrows represent messages. $a \rightarrow b$ if and only if there is a path from event a to event b going up the process lines and along the message arrows. The existence of such a path means there is a flow of information from a to b, or that a can causally affect b.

The usefulness of Lamport's analysis is that it isolates all that it is meaningful to talk about in relation to the ordering of events in a distributed system. An event happens before another event only if there is a flow of information or a chain of potential causality between the two. Although an observer outside the system might be able to discover which of two *concurrent* events happened first, such a fact has no effect on any of the computations performed by the system. It is not really meaningful to talk of two events being simultaneous.

The notion of transparent debugging can now be defined more precisely and yet in a way which gives more freedom in the design of a transparent halting mechanism. Transparent debugging occurs when the same partial ordering of events occurs in the execution of a distributed program under control of the debugger as "would have occurred" had the debugger not been present. This definition is unusual since the execution which actually occurred (under control of the debugger) is compared with a hypothetical execution. We cannot test whether a debugger meets this definition, for instance by executing the program without the debugger and comparing the event orderings, since in general a distributed program is nondeterministic and its executions cannot be reproduced reliably given the same inputs. Instead we must scrutinize the implementation of the debugging support to ensure it does not affect the relative ordering of events in processes.

How shared-memory processes interact

As described above, Lamport's model of distributed computation considers message passing as the only means of communication between processes. However, in the kinds of systems this thesis is concerned with, processes may interact in many other ways.

Processes on the same node can communicate through shared memory. This may take place via data objects designed for inter-process sharing such as semaphores, monitor locks, or critical regions. These objects may in turn protect access to sequentially accessed data objects such as arrays and records. Interaction may occur through undisciplined or unsafe concurrent access to data. It is important to consider this possibility since the programs which the debugger must cope with may contain bugs of this kind. In addition some systems use intra-node message passing between processes which share

memory. Communication may also occur when a process *forks* a new process which begins executing on the same node, or when a process *join* occurs. Processes on different nodes may communicate by sending and receiving messages, or by performing RPCs. Processes may interact by not sending a message or not updating memory before some time limit has expired. Another kind of interaction which is often overlooked is communication via third parties such as network servers or long term shared memory such as files. This last kind of interaction is examined in Chapter 7.

Lamport's model can be applied to programs with shared memory and RPC by expressing all these kinds of process interactions using messages and in some cases extra processes:

semaphores: Notifying and waiting on a semaphore corresponds to the sending and receiving respectively of an empty message.

monitors: Each instance of a monitor lock can be modelled as a process. Invoking a monitor procedure corresponds to sending a message containing the arguments of the procedure to the process implementing the monitor. Returning from the procedure corresponds to a reply message back to the invoking process. The monitor process continually waits for any message which corresponds to a monitor procedure, executes the body of that procedure and returns the results. The data protected by the monitor lock is never accessed concurrently and does not constitute process communication via shared memory. However, any access by the monitor to data objects that are also accessible to processes outside the monitor must be treated as accesses to arbitrary shared memory (see below). This model is reminiscent of the duality transformation described by Lauer and Needham.[6]

critical regions: A critical region can be modelled as a monitor in which the data objects protected by the critical region are treated as the monitor data and the critical region lock as the monitor lock. The same comments apply to data objects which are accessible both inside and outside the critical region.

---

[6]H. E. Lauer and R. M. Needham. "On the Duality of Operating System Structures." *Proc. of the 2nd International Symposium on Operating Systems, Theory and Practice.* IRIA, Rocquencourt, France, October 1978. (Reprinted in *Operating Systems Review,* **13**(2), April 1979, p. 3.)

Ada rendezvous: An entry call[7] is modelled as a message containing the in parameters from the invoking process to the process executing the accept statement. The return from the entry call is modelled as a reply message containing the out parameters. This ignores aspects of rendezvous such as conditional and timed entry calls, and use of the delay and terminate statements within an accept statement.

shared memory cells: All other access to shared memory can be considered as reading and writing a cell of memory. The size of the cell is determined by the granularity of any hardware interlock on memory. A memory cell can be modelled by a process which accepts *write* messages to update the value of the cell, and accepts *read* messages to which it replies with a message containing the cell's value.

test and set: The memory cell which is the object of the test and set can be modelled by a process as in the previous case. In addition to *read* and *write* messages this process will accept a *test_and_set* message which will perform the test and set operation and reply with a message specifying whether the test was satisfied.

process fork and join: Process fork is modelled as a message sent by the forker process to the forkee process (in fact the first message it ever receives). The message contains the arguments to the forked procedure. Process join is modelled as a message from the dying process to the process performing the join.

asynchronous message passing: This is directly handled by Lamport's model.

synchronous message passing: Various forms of synchronous message passing cause some information flow back to the process which sent the message in the form of an acknowledgement or reply message. This is modelled by a pair of messages between the two processes.

RPC: This is modelled similarly as a pair of messages. Often in RPC protocols the receipt of the result message by the caller process is also acknowledged, but this acknowledgement is not seen by the application program.

timeouts: The expiration of a timeout (on a semaphore wait or a message receive) may be modelled as a message received from a clock process. This is the only place where real time delays can appear. In practice the values of timeouts are rather imprecise, and as a consequence most programmers choose timeouts generously.

---

[7] *Ada Programming Language.* ANSI/MIL-STD-1815A, U.S. Dept. of Defence, January 1983.

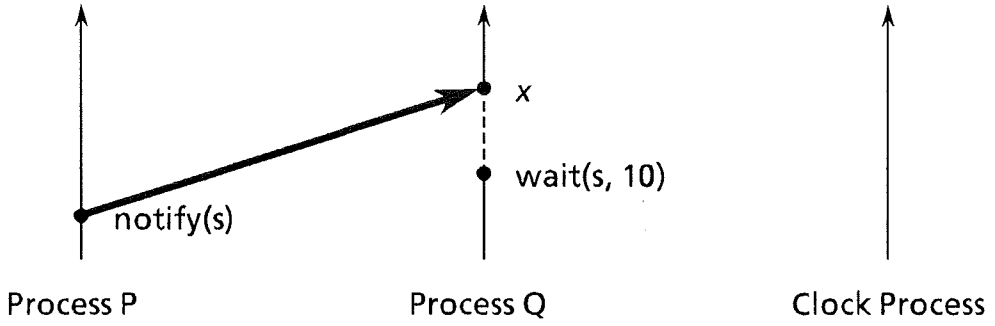

Figure 4.3: A normal execution of processes of the example in Figure 4.1

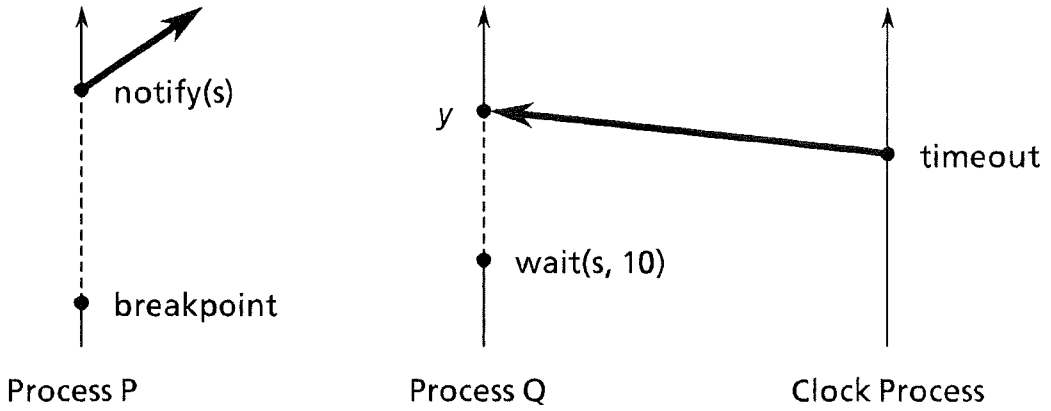

Figure 4.4: An execution of the example in Figure 4.1 with a breakpoint

To illustrate this model consider the example shown earlier in Figure 4.1. Figure 4.3 depicts the ordering of events in the normal execution of this example while Figure 4.4 shows what might have happened when the same example was run under the debugger: event x has been replaced by y.

4.3 When processes must be halted

Using this model of distributed programs we can now determine when processes must be halted by a transparent breakpointing mechanism. The process which triggered the breakpoint must be halted immediately. Other processes must be halted before they could have received a message from a now halted process.

Pessimistically, this implies that all local processes must be halted instantaneously (or rather before they next gain use of the processor) since the next process event of either

a halted process or a still-running process could be a read or write to arbitrarily shared memory. It is not practical to halt the notional processes which implement each memory cell (at least not without hardware support). But it suffices to halt all the real processes which can perform reads and writes on memory.

Remote processes may be halted somewhat later. The case which concerns us here is if the next event of a halted process were the sending of a message to a remote process. The remote process may continue executing until the point when it could have received such a message. When is this? It is the later of two times: (1) when a message from a now halted process would have arrived, and (2) when the remote process next attempts to receive a message from a now halted node. In most RPC systems at least one process is continually waiting for an incoming call message and so must be halted at time (1) above. All the processes on this remote node must be halted since they can all communicate via shared memory.

As soon as one process on a node is halted, we must halt all the other processes on that node too. This reflects the potential high frequency and low latency of interactions through shared memory, as well as the difficulty of detecting when processes communicate in this way. As soon as one node halts, all the nodes to which it can directly send remote messages must be halted within the time taken to deliver such a message.

Assuming that each node can communicate directly or indirectly with every other, i.e. the program is strongly connected, all the nodes would eventually halt by the repeated application of these two rules. In RPC systems with dynamic binding and connectionless protocols (such as Mayflower RPC) every node is a potential RPC destination of every other. Thus all nodes must be halted within one remote message delay after a breakpoint.

Preservation of input values

Besides preserving the ordering of inter-process events we must ensure that the program receives the same input values regardless of the program halting. It is up to the user of the debugger to ensure that input files, for example, remain available for the duration of a debugging run. However one important case which the user cannot handle is when the program reads the value of a real time clock. If the real time values were read just before

and after a breakpoint the program would notice sudden jumps in the progress of time. If any program control decisions were made on the basis of these time values transparent halting would not occur.

Discussion

Stringent requirements are imposed on transparent breakpointing by the potential high frequency of shared memory interactions. Of course the efficiency of these operations is one reason why shared memory is so popular. However, most concurrent programming styles avoid unsynchronized access to shared memory in favour of access via monitors or critical regions. If there was a way to determine that all the data objects in a program were modified under mutual exclusion we would have to halt processes only when they claimed a monitor lock, or waited on a condition variable. This is possible for Concurrent Pascal[8] in which the variable scoping rules enforce mutual exclusion. However these rules are too restrictive and require mutual exclusion in many cases where processes do not conflict. This causes unnecessary serialization of processes and can have a significant impact on performance. Thus the designers of Mesa[9], Ada, and Concurrent CLU decided to be more permissive about access to shared memory in return for some loss of safety in concurrent programming.

It would seem that developments in language design, program analysis, or protected architectures could provide guarantees of mutually exclusive access, without overly restricting concurrency in the program.

One idea worth exploring is to generalize monitor locks to N-reader-1-writer locks. A read lock could be used in circumstances where a write lock would cause undesirable serialization. If this discipline proved sufficiently flexible it could be enforced by the compiler as in Concurrent Pascal. There are problems, however, guaranteeing that unsafe concurrent memory access does not occur when the language contains features such as pointers and procedure variables.

---

[8]P. Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice Hall, Englewood Cliffs, N. J., 1977.

[9]B. W. Lampson and D. D. Redell. "Experience with Processes and Monitors in Mesa." *Comm. of the ACM*, **23**(2), February 1980, p. 105.

Another possibility is a protected architecture in which access to regions of shared memory would be controlled by locks which were owned by only one process at a time. This dynamic checking would be more flexible than static compile time checks. Such a scheme is only practical as part of a thorough protected architecture design such as CAP.[10]

Determining whether these ideas are feasible to implement and whether they provide the intended flexibility is beyond the scope of this thesis. However, if successful, this approach would not only facilitate transparent debugging, but also prevent an important class of concurrency bugs.

Conclusion

I have argued in this chapter that a breakpointing mechanism for a distributed program should endeavour to halt every process to preserve the transparency of the debugging session. A breakpointing method based on this approach is presented in the next chapter.

[10]M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System.* North Holland, New York, 1979.

Chapter 5

Breakpointing a Distributed Program

This chapter presents a method for breakpointing a concurrent and distributed program, using the ideas developed in the previous chapter. Section 5.1 describes a general scheme for implementing breakpoints in a target environment debugger, and Section 5.2 details how process halting is implemented in Pilgrim. The large body of research related to this problem is surveyed in Sections 5.3 and 5.4.

5.1 A scheme for breakpointing distributed programs

This section develops a technique for halting the processes of a distributed program. This technique possesses a high level of transparency and has a negligible effect on the performance of the program. The implementation requires changes to the software which manages processes, which will often include the supervisor (operating system).

A logical time clock is maintained at each node of the program. While the program is executing, the clock runs at the same rate as the real time clock. When a breakpoint occurs, the processes and clocks on each node are halted. The processes executing on the node which encountered the breakpoint are halted immediately. Those executing on other nodes are halted as soon as possible subject to communication delays. A significant feature of the technique is that no messages are sent and little processor time is consumed except when the program is to be halted. Thus programs under control of the debugger do not execute more slowly because of this breakpointing mechanism.

The clock delta

The logical clock is implemented by computing the difference, or *delta*, from the real time clock value maintained at each node. When the program is executing normally the delta remains constant, and is subtracted from all date and time values read by the user program. The instant the node is halted the logical time is recorded:

$$halt_{logical} = halt_{real} - delta$$

While the program remains halted this value is used for the logical time. When the program resumes the delta is recalculated:

$$delta' = resume_{real} - halt_{logical}$$

The real time clocks at each node could be synchronized by an algorithm such as the Berkeley Time Protocol.[1] If the clocks are not exactly synchronized the deltas will still preserve the (inaccurate) time values which would have been read had the breakpoint not happened.

At the end of a debugging session, when the debugger is disconnected from the program, the logical clock is reset to real time. The effects of this may be unpredictable so it may be unwise to let the program continue executing after the debugger has been disconnected.

Halting processes on a single processor

It is relatively easy to transparently halt processes which are time-sliced on a single processor. A special *halt queue* on which to put halted processes is added to the runtime process support. When a runnable process is to be halted it is removed from the ready queues and placed on the halt queue, and will not be allocated to the processor by the process scheduler. A process which is waiting on a semaphore, etc. with a timeout has that timeout frozen. If a waiting, halted process is woken up (e.g. by a semaphore notify) it is added to the halt queue.

---

[1] R. Gusella and S. Zatti. "TEMPO: A Network Time Controller for a Distributed Berkeley UNIX System." *Proc. of the USENIX Summer Conference,* June 1984.

To identify which processes to halt when a breakpoint or an execution error occurs each process should belong to a *halt group*. Whenever a process encounters a breakpoint or an execution error all the processes in its halt group are put on this wait queue. If possible this should happen before the processor is yielded (perhaps by disabling interrupts).

Some processes must not be halted when debugging. When more than one logical node is supported on the same computer, nodes which do not belong to the program being debugged must not be halted. Processes in the operating system itself and processes in the agent necessary for debugging support must remain runnable. These should be in a different halt group from any user process. Also, user processes which are executing in critical sections shared with these system processes must be allowed to complete those sections before halting. For instance user processes must be allowed to return from supervisor calls (SVCs). So there must be a way to move a process from one halt group to another. This must be quite fast to be used upon every supervisor call.

The mechanism by which processes are transferred from the ready queues to the halt queue must be fast too. Potentially large numbers of processes are involved, and if interrupts are to be disabled throughout this period, the correct operation of device drivers could be compromised. This problem is reduced if the ready queue is structured as a collection of smaller partial ready queues. This organization is commonly used in operating systems to give each priority level its own ready queue. If it is arranged that all the processes on a partial ready queue are in the same halt group, the operation of transferring them to the halt queue will be much faster and will not depend on the number of processes to be transferred.

Halting remote processes

To halt remote processes, messages are sent by the agent on the node which initiated the breakpoint to the agents on other nodes in the program. On receiving this message each agent will halt the processes on its node using the local halting mechanism. To achieve transparency it would be necessary to guarantee to send messages to each affected node within the time taken for an RPC, or some other application level message. Something approaching this can be achieved on a single broadcast network (such as the Ethernet)

that provides client nodes with a multicast facility, although there is no guarantee that the message has been received by all the addressed nodes. However a number of networks, including the Cambridge Ring, do not provide multicast.

A fast reliable multicast protocol is required. A number of protocols are available which achieve the necessary reliability.[2] In the simplest protocol, acknowledgements are sent by all nodes who receive the message. This typically overloads the node which sent the multicast and acknowledgements are lost, causing unnecessary retransmissions. As the number of nodes in the multicast set increases, this problem becomes more serious. A *negative acknowledgement* scheme is much more desirable since only one successful reply is sufficient to decide if a retransmission is needed. The loss of later replies cannot affect the decision. This protocol must be supported at the lowest level in the network interface hardware since a negative acknowledgement must be generated whenever the network interface is congested by other recent network interactions. The Ethernet, for instance, does not have this ability, but the Cambridge Ring does. A *saturation protocol* is also quite effective. The broadcast is performed a sufficient number of times to guarantee to an extremely high probability that every node received the message. The number of repetitions is based on the reliability of various network components. The reliability of the network medium itself is usually known quite accurately, but it is often impossible to determine the likelihood of packet loss due to congestion at the receiver. Therefore a conservative figure should be chosen for the number of repetitions.

On networks lacking multicast, a tree structured protocol can improve the performance, by exploiting parallelism. With such a protocol the nodes of the program are logically arranged in a tree with the node originating the multicast at the root. When a node receives a packet from its parent it forwards the packet to each of its children in turn. Great care must be taken when implementing such a tree multicast to avoid overloading the bandwidth of the network, since the resulting contention could well reduce the performance of the multicast.

[2]P. V. Mockapetris. "Analysis of Reliable Multicast Algorithms for Local Networks." *Proc. Eighth Data Communications Symposium*, ACM, October 1983, p. 150.

The level of transparency achieved

The success of this scheme for halting local and remote processes depends on many details of the hardware and software environment.

If it is feasible to modify the process scheduler as described, or to consider debugging early in its design, the halting of processes within one node will be very successful. The state of the processes and memory on that node will be precisely the state which occurred when the breakpoint was triggered. However, transparency will often be lost when the processes are resumed after the breakpoint. This is so because the precise order in which processes are scheduled following the breakpoint may depend on factors such as the timing of hardware interrupts and any non-deterministic activity of processes in the supervisor. These effects ought to be similar to the non-determinism experienced by the program when not under control of the debugger.

The degree of transparency achieved when halting processes on remote nodes depends heavily on the characteristics of the network. On the Ethernet, a saturation protocol would succeed in reaching most nodes with the first multicast packet. It can be argued that nodes which were congested and only received the multicast on the second or third transmission would also have failed to receive an application level message on the first attempt.

The situation is more difficult on networks which do not have a multicast facility, since a number of individually addressed packets must be transmitted. When the number of nodes to be halted reaches a certain point, the delay in receiving a packet will exceed the delay for an application level message. For large numbers of nodes transparency may be lost. Well-behaved programs, which use timeouts appropriate to the reliability and delay of the network, will be handled correctly by the debugger. But highly timing-sensitive faults in programs consisting of large numbers of nodes may be difficult to debug.

Improvements to remote halting

The problems with remote halting are in part created by the assumption of dynamic binding in the RPC system. With static binding, the RPC bindings are fixed when the

program is compiled, linked, or loaded. Each node need only multicast to its immediate neighbours, since they are the only nodes which could have received a message directly from the now halted node. Suppose that a node may multicast to m nodes in the time taken for a single application level message. In other words, transparency will be lost if a node must multicast to more than m nodes. With dynamic binding this will occur for programs of more than $m+1$ nodes. However, with static binding this will happen if some node of the program has degree greater than m, considering the program as a graph. We would expect the maximum degree to be much lower than the number of nodes, especially for large programs. Thus static binding makes it more likely that transparency will be preserved when halting remote processes.

Unfortunately, static binding is quite restrictive for the construction of distributed programs and services. However, we can apply this approach to dynamic binding by dynamically establishing *connections* between nodes wishing to communicate. When a node halts, it multicasts the halt message to those nodes to which it currently has connections. When a new connection is made, if either party knows that the program has halted, it will inform the other. Finally we insert a delay into the binding procedure to ensure that creating a new connection takes longer than the longest path which a halt message could take. This satisfies the constraint of delivering a halt message in less time than an application level message, both for messages sent over existing connections, and those sent over newly created connections.

The delay could be chosen statically, placing limits on program sizes and connectivity, or it could be updated dynamically given some assumptions on how quickly this value could be propagated and how quickly it might change. If a program was initialized with sufficient connections to make it strongly connected, we could enforce the delay by requiring binding messages to be forwarded along existing connections, i.e. along the same paths which halt messages must use.

Connectionless RPC systems, such as Mayflower RPC, could use this technique if connections were created on demand by the RPC implementation without requiring extra steps by the programmer, and deleted automatically after some period of disuse. However this modification to Mayflower RPC was not tried for Pilgrim.

5.2 Distributed breakpointing in Pilgrim

Pilgrim provides code and exception breakpoints. Details of how these events are detected are described in Chapter 6. Once an event has been detected the program is halted by the agent with assistance from the Mayflower supervisor. The agent also maintains the clock delta, recomputing it whenever the node is halted or resumed.

The supervisor: The supervisor implements a halt group mechanism similar to that described in the previous section. Recall that each logical node of a Concurrent CLU program occupies one domain (address space). The supervisor is in a special domain of its own. For each domain the supervisor maintains a halt group. To allow agents to implement node halting, two SVCs are provided: one to add or remove a process from the halt group, the other to halt or resume the halt group for the domain. When the domain is halted, no processes in the halt group are run, and timeouts on semaphore waits by processes in the halt group are frozen.

Upon certain kinds of hardware exceptions (such as division by zero and breakpoint traps) the supervisor passes control to a process in the agent. Before doing so the supervisor halts the node. In other cases the agent itself detects faults or breakpoints, and calls an SVC to halt the node. In all cases a node is resumed by the agent calling an SVC.

Managing the halt group: The agent is part of the debuggee node it controls and they share many functions in the Concurrent CLU runtime library. For instance the agent uses the same RPC system and allocates storage off the same heap as user code. So, to prevent deadlock when the node is halted, user processes should be removed from the halt group when executing in critical sections which are also used by the agent.

It proved too expensive for user processes to do an SVC before and after such agent critical sections to temporarily remove themselves from the halt group. Instead I implemented a faster mechanism which avoids calling the SVCs until the node actually halts. Each user process records its own nesting level within critical sections but does not call the SVC. Immediately after the node halts, the agent finds all processes within agent critical

sections and removes them from the halt group until they exit the section or the node restarts.

This mechanism would be unnecessary if the agent ran in a separate domain, with a different copy of the runtime library and a different heap from the debuggee node that it controls. The agent ought to have full access to the debuggee's memory, while the debuggee should not have access to the agent's. This structure was not chosen for a number of practical reasons. Sharing of resources and communication between domains on the same machine is not well supported in Mayflower, and would be difficult to add. Also multiple domains waste memory, and many of the machines in the processor bank do not have large memories.

Halting remote processes: When the node is halted, the agent initiates a multicast protocol to instruct all other agents to halt their nodes. At the beginning of a debugging session, each agent is given the RPC address of every other agent for this purpose. The multicast protocol used is a variation of a negative acknowledgement protocol. A sequence of messages is sent, one to each agent. On the Cambridge Ring the transmitting hardware is informed if the packet just sent was not received by the destination network interface. This information is examined by the agent and failed transmissions are repeated. This guarantees all the destinations receive the message into their network buffers. It is assumed that either the agent software in those nodes is functioning correctly and will process the message, or the entire node has crashed.

Performance

Managing the halt group: Adding and removing a process from the halt group takes a total of approximately 500μs. Of this about 200μs is the overhead involved in performing two SVCs. However entering and then exiting an agent critical section without calling the SVCs takes only 50μs. The SVCs need only be called after the node has actually halted, so they do not slow down the program when executing normally and do not alter the node's logical time.

There are many agent critical sections and they occur frequently in the execution of a

user process, when allocating storage off the heap, when performing stream I/O, or when using RPC. Since these will be executed at all times, even when the program is not under control of the debugger, their impact on performance could be crucial. For instance, performance measurements demonstrate that stream output operations take about 100μs longer, because of agent critical sections. This represents a 5% slow down when outputting a single character to a remote terminal stream. Similarly, a null *maybe* RPC takes 1500μs longer or about 10%. This is a reasonable price to pay for the benefits of transparent, target environment debugging. By running the agent in a separate domain, these figures could be reduced.

Halting local processes: Halting and restarting a node has a variable cost because halt groups are implemented by an adaptive algorithm. Halt group membership is specified by a bit in the supervisor state of a process. When the node is halted, a special debugging scheduler is used which ignores processes with this bit set. Meanwhile the idle process searches the ready queues for such processes and removes them to a special wait queue. Upon node restart these processes are moved back to the ready queues. The effect of this is that halting a node and resuming it a short time after (e.g. before the next process switch) costs very little—in fact about 400μs. This figure is independent of the number of processes in the halt group. Halting and restarting a node over a longer period costs about 250μs for each process in the halt group.

Halting remote processes: The distributed halting protocol is able to transmit a halt message to one node every 3.5ms if no packets are lost. Unfortunately, the minimum latency time for an RPC under Mayflower is about 8ms because of its efficiency and light-weight semantics. Thus we can be confident of contacting only two nodes in the time available for transparently halting remote processes.

The performance of this protocol could be improved by arranging the nodes to be contacted into a tree. As each node received a halt message it could contact further nodes in parallel. However in Pilgrim's case, only one more node could be contacted within the time available. The performance of remote halting could also be improved using the technique described in Section 5.1, in which the size of the multicast set is reduced by adding connections to the RPC system.

5.3 Related work: transparent halting

The problem of transparently examining concurrent program states has been recognized by many other researchers, some of whom have developed techniques to handle it.

Halting processes using priorities

The HARD debugger[3] for non-distributed Ada programs halts all the processes in a program when a breakpoint is hit, and freezes the timeouts on delay statements until the program is resumed. This facility is very similar to Pilgrim's mechanism for halting local processes, but it is implemented quite differently.

Breakpoints are implemented by performing an entry call to a debugger process which executes at higher priority than user processes and prevents them from running. This halting technique is only appropriate to a single processor, and relies on absolute priority scheduling in which a higher priority process is never pre-empted by one of lower priority.

HARD is not a target environment debugger; it operates upon an inefficient Ada interpreter which uses a relational database to store the program and its dynamic execution state.

Breakpointing at a consistent state

Miller and Choi[4] have developed a method of halting distributed programs which is very similar to my method of halting processes on remote nodes, although they do not consider timeouts, or processes which share memory. In their scheme, when one node halts it sends halting messages to all other nodes to which it can send messages.

---

[3]A. Di Maio, S. Ceri, and S. Crespi Reghizzi. "Execution Monitoring and Debugging Tool for Ada Using Relational Algebra." *Ada in Use, Proc. of the Ada International Conference*, Paris, (*Ada Letters*, 5(2)), September 1985, p. 109.

[4]B. P. Miller and J. Choi. *Breakpoints and Halting in Distributed Programs.* Technical Report 648, Computer Sciences Dept., University of Wisconsin-Madison, Wisc., July 1986.

Although their algorithm is similar, their analysis of the problem is different. They base their method on Chandy and Lamport's[5] algorithm for obtaining a snapshot of the state of a distributed program while it is running. Miller and Choi's goal is to halt the distributed program in such a way that the state at each node and the undelivered messages in message queues form a *consistent global state* of the computation. A consistent global state is any state reachable from the state of the program at the point the halting algorithm is initiated.

Maintaining a consistent state when a program is halted is a weaker requirement than transparent halting. Under transparent halting, we wish a particular ordering of events to occur despite the program being halted. A consistent state is a state resulting from any valid ordering of events specified by the program. Indeed it would appear that, in the absence of timeouts, *any* halting method in which all processes eventually halt will result in a consistent state.

Miller and Choi do consider the problems which result from the debugger altering the absolute timing of concurrent processes. They conclude that highly timing sensitive programs (or bugs) probably cannot be debugged transparently without using a hardware monitor.

The most significant contribution of their research, however, concerns production rule breakpoints in which the predicates on some rules are distributed in different processes. They analyze what such distributed predicates mean, given the lack of a unique total ordering of events in a distributed program, and then discuss how these distributed predicates can be detected to trigger a breakpoint.

Cautious execution

In my halting scheme, each node executes without restriction until it receives a halting message. If that message is delayed a node may execute for too long and destroy the transparency of the debugging session.

---

[5]K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Trans. on Computer Systems*, 3(1), February 1985, p. 63.

An alternative would be to ensure that no other nodes have halted before allowing a node to receive a message. Schiffenbauer[6] developed a method for monitoring and debugging message passing among the nodes of a distributed system which does just that. In his technique, all inter-node messages were displayed to the user and could be modified, deleted, or allowed to continue to their destinations. However his scheme could equally be used for a breakpointing mechanism, in which all messages flowed without user intervention until a breakpoint was triggered.

Under this scheme, all messages are redirected via the debugger node. The value of the logical clock at the sending node is appended to every message and this is used to order messages, and to maintain logical time at the receiving node. When a node wishes to receive a message it sends a request message to the debugger. The debugger replies with the next available message for that node. In the meantime all processes and the logical clock on the receiving node are halted. Upon receiving the message the logical clock at that node is set to the time value in the message plus the time taken to transmit the message, and the processes are resumed.

When a breakpoint is triggered, a message is sent to the debugger, which will cease to relay messages to other nodes. Nodes are allowed to continue execution as long as they either perform local computations or send messages. It is only when they might receive information from remote processes that they are halted.

This scheme is impractical. Two extra messages must be sent upon every message event, and the debugger will become a bottle-neck. Even worse, the entire system may deadlock if at least one process on each node tries to receive a message when there are no currently outstanding message transmissions. All processes on all nodes will be halted, including any potential message senders. In a program using RPC it is likely that every node will have at least one process waiting for incoming call messages. Schiffenbauer's solution is to allow each node in turn to execute for one clock tick until a message is sent, resulting in an exceedingly slow serial execution of the system.

---

[6]R. D. Schiffenbauer. *Interactive Debugging in a Distributed Computational Environment.* M.S. Dissertation, Technical Report MIT/LCS/TR-264, Laboratory for Computer Science, MIT, Cambridge, Mass., September 1981.

Inserting artificial delays

The designers of the ECSP debugger[7] were concerned that evaluating behavioural specifications (in effect production rules) in their system would significantly alter the relative speeds of processes being debugged. To help prevent this a pseudo-event, delay, is provided which halts named processes while the behavioural specification involving some other process is being matched. The user must determine where these delays should be placed and which processes to delay.

In Pilgrim, a node or the entire program could be halted during any debugging activity which could affect the transparency of the debugging session. However the support for debugging RPCs and processes currently provided in Pilgrim is efficient enough to be permanently enabled and thus there is no difference in execution speed of a program whether it is being debugged or not.

5.4 Related work: replayable and reversible execution

Halting and resuming a concurrent and distributed program transparently is clearly difficult. Many researchers have chosen instead to avoid interrupting the program at all when debugging. Rather the program is run to completion while various program events are logged for examination later. Logging is not strictly an interactive debugging technique, but when it is used to implement reversible or replayable execution, the result is a set of debugging facilities which to the user appear similar to interactive breakpoint debugging.

Complete logs

In a few systems all program state changes are recorded in a log, including variable assignments as well as inter-process events. Any piece of information about the run may

[7] F. Baiardi, N. De Francesco, and G. Vaglini. "Development of a Debugger for a Concurrent Language." *IEEE Trans. on Software Engineering*, **SE-12**(4), April 1986, p. 547.

then be obtained by examining the log. Some systems[8] view the log as a database, and the user can formulate sophisticated queries about the program run.

Recording all events considerably slows down the execution of the program, and huge amounts of storage are required for the log.

Logs of inter-process events

Even in a concurrent program, long sequences of operations in each process will be deterministic. By recording only inter-process communication and other sources of non-determinism, a smaller log will result. To obtain the state of a process at some point, that process can be re-run against the log ensuring that the same inter-process communication and non-deterministic choices take place. This is known as replayable execution.[9]

Replay schemes have been applied almost exclusively to systems in which processes communicate only by message passing. In shared memory environments, the higher frequency of process interactions and the difficulty of detecting them makes most logging schemes completely impractical.

Logging is sometimes combined with periodic checkpoints.[10,11,12] This means the program or one of its processes can be re-run from the most recent checkpoint prior to some point of interest, rather than from the beginning. This reduces the delay in recreating some previous program state and can thus be termed reversible execution.

---

[8]C. H. LeDoux and D. S. Parker Jr. "Saving Traces for Ada Debugging (YODA)." *Ada in Use, Proc. of the Ada International Conference*, Paris, (*Ada Letters*, 5(2)), September 1985, p. 97.

[9]E. T. Smith. "A debugger for message-based processes." *Software—Practice and Experience*, 15(11), November 1985, p. 1073.

[10]R. Curtis and L. Wittie. "Bugnet: A debugging system for parallel programming environments." *Proc. of the 3rd International Conference on Distributed Computing systems*, Miami, Florida, October 1982, IEEE Computer Society Press, Silver Spring, Maryland, p. 394.

[11]L. Wittie and R. Curtis. "Time Management for Debugging Distributed Systems." *Proc. of the 5th International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985, IEEE Computer Society Press, Silver Spring, Maryland, p. 549.

[12]D. R. Jefferson. "Virtual Time." *ACM Trans. on Programming Languages and Systems*, 7(3), July 1985, p. 404.

Replaying atomic transactions

A variation on this theme is Chiu's proposals for debugging atomic transactions in Argus.[13] (This debugging scheme should not be confused with the Argus debugger actually implemented.) In Argus, an atomic transaction is deterministic with respect to the versions of the (atomic) objects that it reads. In Chiu's technique, a log would record the serialization order of transactions, along with pointers to the particular object versions that each transaction read. By retaining old object versions indefinitely, Chiu would be able to replay any atomic transaction.

Chiu's technique is more economical because much of the cost normally associated with logging and replay is already borne by the Argus system itself. Thus, in order to provide atomic and recoverable semantics, the Argus system preserves old versions of the objects modified by a transaction, at least until that transaction commits. Chiu would merely retain these objects on archival storage for use in future debugging sessions.

Instant replay

Further economies can be made when logging inter-process events by only recording the order in which events occur, without recording the data associated with the events (such as message contents or object versions). As long as the partial ordering of process events recorded in the log is precisely recreated, and the program receives the same external inputs, the processes themselves will correctly recompute the data associated with each event.

This technique, termed *instant replay*, has been developed by LeBlanc and Mellor-Crummey,[14] based partly on work by Carver and Tai.[15] Instant replay produces a much shorter log than the previous techniques and the logging activity is much cheaper.

---

[13] S. Y. Chiu. *Debugging Distributed Computations in a Nested Atomic Action System.* Ph.D. Dissertation, Technical Report MIT/LCS/TR-327, Laboratory for Computer Science, MIT, Cambridge, Mass., December 1984.

[14] T. J. LeBlanc and M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay." *IEEE Trans. on Computers*, **C-36**(4), April 1987, p. 471.

[15] R. H. Carver and K. Tai. "Reproduceable Testing of Concurrent Programs Based on Shared Variables." *Proc. of the 6th International Conference on Distributed Computing Systems*, Cambridge, Mass., May 1986, IEEE Computer Society Press, Washington D.C., p. 428.

However all inputs to the program must still be logged completely, and all the processes in the program must be replayed together. This makes the technique difficult to apply in a distributed environment where a program may run for long periods and receive large amounts of input from other programs in the system. To date, the technique has only been applied to parallel but non-distributed programs which had total execution times of less than five minutes, and appeared to perform no input. It would be interesting to combine instant replay with a non-intrusive checkpointing algorithm such as Chandy and Lamport's[16] for use on distributed programs.

The main problem with the technique is that, if any events affecting event orderings are not logged (such as unsafe concurrent access to shared memory), a different ordering may occur when the program is replayed and an invalid computation will result, i.e. a computation not specified by the program text. Thus, while instant replay has been used on shared memory programs, those programs communicated using well-defined locking primitives to ensure safe concurrent access.

Problems with logging

Replayable and reversible execution have one definite advantage over breakpoint debugging in that they require less foresight on the part of the user. Often by the time a program is halted (at a breakpoint or because of an execution error) the state or event which caused the bug will have occurred much earlier and the evidence been destroyed. Simply re-executing a concurrent program with a breakpoint at an earlier point, in an attempt to recreate the desired state, may fail since we may not be able to reproduce the computation exactly.

However replay techniques must be rejected for debugging distributed programs in the target environment for the following reasons. Writing the logs slows the execution of the program down considerably, and the logs themselves are very large for non-trivial programs. Of all the logging techniques, instant replay has the best performance. However it still requires all external inputs to the program to be logged. In a distributed computing environment this could be very expensive because programs may run for long periods and

[16]Chandy and Lamport, 1985.

interact with many services on the network. (Consider all the messages received from a file server during program execution). Thus, no such scheme is seriously used for distributed, target environment debugging.

Replayable and reversible execution solve the reproduceability problem, but only at considerable cost in execution speed and storage. So the user will have to remove debugging support in the target environment. This brings us to the dilemma of special "debug mode" techniques: if a bug should appear in the target environment there will be no tools to debug it; if the program is returned to the debugging environment which permits program replay the bug may be impossible to recreate.

Incomplete logs

Finally we should consider partial logging which does not permit program replay. This is a valuable debugging and monitoring technique and has often been used where there is a complete lack of debugging tools.[17] Program tracing and tracepoints, which are based on breakpoints, can also be considered here. Rather than interrupt the program indefinitely, these facilities display a specified portion of the program state and resume the program straight away.

The significant drawback of partial logs is that only a small portion of the program state is visible. To obtain further information about the suspected cause of a bug the program must be re-run to log extra information. However partial logs remain an important programming analysis technique, in conjunction with interactive breakpoint debugging.

Conclusion

In this chapter I have presented a scheme for breakpointing concurrent and distributed programs. This technique is suitable for use in the target environment, and preserves a high degree of transparency. I have also reviewed the related work and explained the shortcomings of several significant alternatives to my approach.

[17] H. Garcia Molina, F. Germano Jr., and W. H. Kohler. "Debugging a Distributed Computing System." *IEEE Trans. on Software Engineering*, **SE-10**(2), March 1984, p. 210.

Chapter 6

Debugging Concurrent Processes

The major issue in debugging concurrent processes is maintaining transparency, in particular ensuring the actions of the debugger do not alter the relative orderings of inter-process events in the program. This has been addressed at length in Chapters 4 and 5.

However besides providing transparency, we need facilities to actually observe and control concurrent processes. The first three sections of this chapter discuss the facilities needed and sketch in general terms how they can be implemented. Sections 6.4 and 6.5 describe how these facilities are actually implemented in Pilgrim. The user interface of the debugger becomes particularly important when debugging concurrent programs. The chapter ends with some ideas for user interface design.

6.1 Viewing process states

The first requirement of a debugger for concurrent programs is to be able to display the states of all its concurrent processes at practically any instant. The state of a process includes the schedule state (e.g. runnable or waiting), the current source location, the call history, and the values of all the variables and data objects accessible to the process.

The main difficulty here is in discovering the schedule state and locating the call stack of every process. The implementation of this debugging support is usually highly dependent on how processes are implemented in the programming language.

The debugger must have access to an up-to-date list of all the processes in the program. If the runtime support for the concurrent language maintains a list of all the processes, that

list can be used. Alternatively the debugger may insert a hook into the process creation and deletion code allowing it to maintain its own list.

Locating call stacks

Given a particular process we wish to display its state. To do this the debugger requires two pieces of information from the process support: the schedule state, and the register set of the process. Two registers are important: the program counter and the register which points to the topmost frame on the call stack. Other registers may contain the current values of program variables if the compiler has performed certain kinds of code optimization.

The process support should be modified to allow this information to be obtained by the debugger. The debugger can then display the current execution point and the variables in the call stack in source language terms using tables produced by the compiler and linker. This will be much the same as in a sequential debugger.

Interpreting the very top of the call stack can be more difficult than for a sequential program. A sequential program (or rather its single constituent process) can be interrupted at well defined places, such as between statements, when the stack and the registers are in a state which is easy to interpret. When debugging a concurrent program, we wish to be able to display a process's stack not only when it has hit a breakpoint, but at any point where it may be suspended by the scheduler.

Two strategies are available to cope with this. The first requires the compiler to produce sufficiently detailed debugging information to ensure that the source code location and topmost stack frame can be found for any value of the program counter. The other strategy is to single-step a process until it reaches a well-defined place, such as the beginning of the next statement, before interpreting the stack. This could be quite expensive if applied to every process whenever the program is halted. So it should be done only when the user actually wishes to display a process's state.

6.2 Controlling process execution

The debugger should allow the execution of processes to be monitored and controlled by the user. The breakpointing facility described in Chapter 5 should be used. Under this facility, most processes on a node belong to the *halt group*. Upon specified program events, all the processes in the halt group are halted. After the user has examined the program state these processes are resumed.

The user may wish instead to single-step one process while all the others remain halted, or to resume a subset of the processes in the halt group. This activity can be supported by having more than one halt group in a program, and by allowing the user to change halt group membership. The actual method by which processes are halted and resumed depends on details of the runtime support. Single-stepped execution can be implemented in the same way as in sequential debuggers.

6.3 Accessing concurrent data objects

For much of the time, a process in a concurrent program behaves like a sequential program: it accesses and updates objects which are used solely by itself. Thus, much of the activity of debugging is concerned with sequential code for which the techniques provided by conventional debuggers are appropriate.

To debug the concurrent aspects of a program, we are concerned with how processes interact and the state which is shared between them. A list of the ways in which processes can interact was presented in Section 4.2. Processes principally communicate through concurrent data objects, such as semaphores, which are defined in the programming language. In many languages the programmer can construct new concurrent data objects using monitors. In Concurrent CLU, programmers can write new abstract data types for inter-process communication by encapsulating semaphores and critical regions along with other data items using clusters.

The debugger should provide facilities to observe and perhaps modify these kinds of concurrent data objects. When asked to display a semaphore, the debugger should display the

value of the semaphore's counter and the state of any processes which are waiting on the semaphore. A critical region lock (or a monitor lock) is essentially a binary semaphore, and should be displayed as such. Programmers of user-defined objects should provide procedures which can be invoked from the debugger to display objects. In Concurrent CLU, most clusters provide a print procedure which the debugger can invoke automatically to display an object.

Consider a single-slot buffer used for communicating a string message between consumer and producer processes. Its state might consist of the following record:

```
record [ slot: string,
         full: bool,
         wait_empty: semaphore,
         wait_full: semaphore ]
```

The semaphores wait_empty and wait_full are used by processes which are waiting for the slot to become empty or full, respectively. The programmer should write a print procedure which shows whether the slot is full or empty, and displays any processes waiting on either of the semaphores by calling the print procedure for a semaphore.

The user may want to modify some concurrent objects, for instance to test a hypothesis about the cause of a bug. This can be done by invoking the appropriate procedures provided by the concurrent object, and can be implemented with the debugger's normal procedure invocation mechanism. Extra procedures could be provided specifically for debugging, for instance to allow the queue of processes waiting on a semaphore to be re-ordered.

6.4 Processes in Mayflower

Before describing how I provided concurrent debugging support for Pilgrim, it is necessary to describe some details of how Concurrent CLU processes are implemented in the Mayflower environment. The implementation of processes is split between the Mayflower supervisor and the Concurrent CLU library. The supervisor executes in privileged mode on the MC68000 processor, giving it access to hardware interrupts and I/O devices. The

Concurrent CLU library is a set of clusters and procedures which are included in a program by the linker. Communication between the library and the supervisor takes place using supervisor calls (SVCs).

The supervisor provides very simple processes which are little more than independently schedulable threads of control. SVCs are provided for thread creation and deletion. Preemptive, time-sliced scheduling with priorities is used. Threads can synchronize with each other on queue-IDs (23-bit unique numbers) by calling the *wait* and *kick* SVCs quoting the appropriate queue-ID. These are similar to, but more primitive than, semaphore wait and notify operations.

Most of the support for Concurrent CLU processes is found in the process, semaphore, and lock (critical region lock) clusters in the library. The process cluster implements Concurrent CLU processes out of threads, adding management of the call stack and the semantics of fork and join. A semaphore consists of a queue-ID and a counter, and uses the kick and wait SVCs to arbitrate any contention. A lock is built out of a semaphore.

6.5 Concurrent process debugging support in Pilgrim

Both the supervisor and the Concurrent CLU library were modified to provide debugging support for the Pilgrim debugger. Recall that part of the debugger, the agent, is included with each node by the linker. The agent uses the supervisor and the library to perform debugging actions on behalf of the rest of the debugger which executes on another node. The interface between the agent and the rest of the debugger is detailed in Chapter 8.

The supervisor

The get_thread_info SVC was added to the supervisor to obtain debugging information about a specified thread. The information returned is:

- the address of the saved register set
- the thread's schedule state: running, waiting on a queue, kicking a queue, or dead
- the queue-ID the thread is waiting on or kicking, if any

- the thread priority
- whether the thread belongs to the halt group for this node

At crucial points, interrupts are disabled in order to obtain a consistent state. It is possible for the agent to modify the saved register set and to cause these new values to be loaded into the registers for the thread when it resumes.

On the MC68000 processor, a number of program errors are detected by the hardware including division by zero, addressing errors, and illegal instructions. Each of these hardware exceptions generates an internal interrupt which is caught by the supervisor. For debugging purposes these exceptions must be reported to the agent for the node concerned.

When a node is initialized, the agent registers the address of a procedure with the supervisor. This procedure is called upon hardware exceptions that occur within that node, providing what is essentially a user-level interrupt handler. Before calling the handler procedure, the supervisor halts all the threads in the halt group. The handler has access to the debugging information described above for the thread which caused the exception, and can change the values of registers. For certain exceptions, the handler procedure may return to the supervisor and the excepted thread is allowed to resume.

The 68000 provides two hardware features, the *trap* instruction and *trace* mode, which are suitable for debugging. The trap instruction causes a hardware exception when it is executed. Pilgrim implements code breakpoints by overwriting an instruction word with a trap. When a special bit is set in the processor, a trace exception is generated at the end of every instruction. This is used to step over a breakpoint by replacing the overwritten instruction and executing in trace mode for one instruction while all other processes remain halted by the halt group mechanism. In the future trace mode will also be used to implement single-stepped execution in Pilgrim.

The runtime library

The process cluster already maintained a list of all the processes on that node. I replaced it by a hash table to speed up searching.

The **print** procedure for **process** displays most of the programmer-visible state of the process including the call history and schedule state. To do so it requires source-level debugging information which is only available when the program is connected to a debugger. During normal execution the **print** procedure merely displays the process identifier.

The **print** procedures for **semaphore** and **lock** currently display only the value of the counter. They should also display the processes which are suspended on the object. The latter information is rather expensive to obtain since it requires calling the **get_thread_info** SVC for every process to discover which of them are waiting on the given semaphore. An improvement would be to add a **get_queue_info** SVC which returned the process identifiers of those processes waiting on a particular queue-ID.

The compiler

The task of interpreting the top of a process's call stack is complicated by the need to observe processes at any point in their execution, and by the optimizations performed by the Concurrent CLU compiler. The two pieces of information required from the top of the stack are a pointer to a procedure stack frame for the highest well-formed frame on the stack and a pointer to the current object code location within that procedure.

Most of the time, the frame for the currently executing procedure is pointed to by a fixed register, and the program counter is within that procedure. However this is not the case during the procedure entry and exit sequences, as a new frame is being built or destroyed and control is being transferred. The compiler generates a multiplicity of code sequences at these points, based upon the number of argument and return values to the procedure. Very short procedures are implemented without a proper stack frame at all, with the frame pointer addressing the stack frame of the calling procedure. Finally there are some assembly code routines in the library which build their stack frames in an ad hoc way. Cargill reports very similar problems in implementing a debugger for C for an MC68000-based computer.[1]

[1] T. A. Cargill. "Implementation of the Blit Debugger." *Software—Practice and Experience*, **15**(2), February 1985, p. 153.

To cope with this I modified the compiler and assembler to generate tables describing, for a given program counter value, where a consistent frame pointer and program counter could be found (e.g. in a register, or at a fixed offset from the top of stack pointer). Assembly language code had to abide by some extra conventions for this scheme to function. There are a few assembly routines which do not; these must be modified.

6.6 Managing complexity: user interface issues

The user is flooded with a great deal of information when debugging a concurrent program because of the multiple execution contexts provided by multiple processes. The user interface is important in any debugger, but it becomes crucial in coping with the complexity of multi-process debugging.

Window- and menu-based interfaces

Multiple windows and pop-up menus combined with a large bit-mapped screen and a mouse pointing device can be used to good effect. However the structure of such a user interface must be carefully considered if a convenient and useful debugger is to result.

A technique commonly used in multi-process debuggers is to provide one window for each process in the program. Each window accepts commands and displays results concerning the process it is attached to. Often such an interface reflects the design of the underlying debugger which is little more than a collection of independent sequential debuggers.[2,3]

There are many problems with this approach. Once the program comprises more than a few tens of processes the number of windows will become unmanageable. Associating windows with processes promotes a process-centred view of a computation. This makes debugging data-centred or object-oriented programs difficult. For instance investigating how a monitor behaves will involve switching among many windows as control passes

[2]J. Gait. "A Debugger for Concurrent Programs." *Software—Practice and Experience*, **15**(6), June 1985, p. 539.

[3]E. Adams and S. S. Muchnick. "Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations." *Software—Practice and Experience*, **16**(7), July 1986, p. 653.

among the processes executing in the monitor. Also, evaluating expressions containing variables from more than one process may be impossible.

A better scheme is to have the flexibility to debug any process from any window, and different windows might have different defaults or naming contexts. It should also be possible to multiplex the output from different processes on as many or as few output windows as desired.

Unfortunately no suitable hardware and software providing multiple windows was available when Pilgrim was being implemented, so it is restricted to a glass teletype style interface.

Object-oriented user interface design

The object-oriented approach is a useful way to design the user interface of a debugger. In this approach the entire user-visible state of the debugger and the program being debugged is represented as a network of objects, each of which responds to various commands. For instance, in Pilgrim the object representing a local variable responds to a print command by displaying its value, while the object for a source location accepts a brk command which sets a breakpoint at that point in the code.

There are also commands for navigating around the object space. In Pilgrim, all objects can be accessed by a consistent pathname syntax. For instance the pathname:

```
carver.openfile.filedesc.name
```

refers to the name field of a record which is the contents of the variable called filedesc in a procedure openfile on the node carver. Objects referred to in this way can be assigned to debugger variables for easier reference later.

Often the user wishes to refer to collections of objects when using the debugger. This would be especially useful for applying the same command to a number of processes or breakpoints. Bruegge provides such collections, which he calls *classes*, in his debugger, Kraut.[4] A class can be supplied to a debugger command in any place where a single

[4]See p. 35 of B. Bruegge. *Adaptability and Portability in Symbolic Debuggers.* Ph.D. Dissertation, Technical Report CMU-CS-85-174, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn., September 1985.

member of the class is legal. Operations to add an object to a class or to remove an object are provided as well.

The object-oriented approach combines well with a window-based interface. Each object can display itself in (a portion of) a window on the display, and provide access to its sub-objects by displaying a pop-up menu containing their names. A frequently accessed object could be represented on the screen by an icon (a small graphical place-holder), to avoid the tedium of repeatedly selecting several nested menus.

As stated above, Pilgrim does not operate in a window-based environment. Even so the object oriented approach has helped to organize the internal structure of the Pilgrim, and make it easier to modify. Cargill's Pi debugger for C programs contains an object oriented, window-based interface of the kind I advocate.[5]

Naming processes

Another issue is deciding how processes are to be named by the user. Some languages, e.g. Concurrent Pascal, have a static process structure in which each process is declared explicitly and given a name in the program text. This provides a unique name by which to refer to each process when debugging. Ada also has a static process structure and processes are identified by a unique name, or a name and an index (for process families).

The real problem arises in languages with dynamic processes. In these languages an unspecified number of processes can be created or destroyed under program control. In Mesa, Modula-2+[6] and Concurrent CLU, for instance, any procedure can be forked to create a new process. In Argus the parallel coenter construct allows an arbitrary number of child processes to be created to execute the enclosed block of statements in parallel.

In such languages, often the only unique name for a process is some kind of process number or process identifier. Process identifiers are usually large meaningless numbers which can be different every time a program is run. Using them for process names is analogous to

[5]T. A. Cargill. "Pi: A Case Study in Object-Oriented Programming." *Proc. of OOPSLA'86*, Portland, Oregon, September 1986, p. 350 (published as *SIGPLAN Notices*, **21**(11), November 1986).

[6]P. Rovner, R. Levin, J. Wick. *On Extending Modula-2 For Building Large, Integrated Systems.* Technical Report 3, DEC Systems Research Center, Palo Alto, Calif., January 1985.

referring to a data object by its memory address. During an RPC, the callee process which handles the call usually has a different process identifier from the caller process which initiated the RPC, and the callee process identifier may change on subsequent calls. Naming processes by their identifiers in these cases conflicts with the aim, pursued in Chapter 3, of making local and remote calls appear similar in the debugger.

There are a number of more pleasant ways to name processes when debugging, although these names are not guaranteed to be unique. In some languages processes are first class objects and can be assigned to variables in the program. The name of such a variable can be used to refer to the process it contains. In practice process variables are seldom used in programs so they cannot be used as the sole means of naming processes in the debugger.

The root procedure of the process (i.e. the procedure which was forked) can be used as a name. Alternatively the source location of the fork expression or parallel statement which created the process can be used. The latter method requires that a copy of the program counter is saved when a fork expression is executed.

Commonly the process which is to be identified is engaged in some process interaction, such as a semaphore wait. The debugger should provide some syntax by which the processes waiting on a semaphore can be referred to.

If one of these methods identifies more than one process, all the processes referred to could be displayed for the user to choose which one she intends, based on each process's state. Finally, once a process has been uniquely identified, the user should be allowed to assign a new name to that process for future reference during the debugging session.

Pilgrim currently allows a process to be named by its process identifier, or by the name of a process variable which refers to the process. I have added support for naming a process by its root procedure, or by the fork statement which created it, but have not completed the components of the user interface which use this support. Once a process has been identified it can be assigned to a debugger variable, like any other object in Pilgrim, for easier reference later.

6.7 Related work on debugging concurrent processes

Some debuggers of concurrent programs make no special effort to display processes in source-level terms. In particular, concurrent debuggers that are implemented as a collection of single process debuggers[7,8] naturally provide no facilities for naming or displaying processes. The process associated with each single process debugger is usually implicit, and displaying relationships between processes is often impossible. In contrast, the Alsys debugger[9] displays Ada tasks in a source-level format, and provides an easy way to list all tasks which are synchronized with a particular task. The IDD debugger[10] displays processes and messages graphically in the same way as Lamport's space-time diagrams (see Figure 4.2). The Communication Graph Display System[11] displays an animation of a distributed program, depicting processes as circles with messages as arrows between them.

In many debuggers there is an interval between the occurrence of an event (such as a breakpoint or a program error) and the reporting of the event to the debugger. For instance in Bruegge's debugger, Kraut,[12] the process which triggers an event is halted but other processes may continue until the debugger gains control and explicitly halts them. During that interval, the program state can change and data associated with the event lost. By supporting process halting at a low enough level, such as in the scheduler, this timing window can be avoided.

---

[7]Gait, 1985.

[8]Adams and Muchnick, 1986.

[9]C. Mauger and K. Pammet. "An Event-Driven Debugger for Ada." *Ada in Use, Proc. of the Ada International Conference*, Paris, (*Ada Letters*, 5(2)), September 1985, p. 124.

[10]P. K. Harter, D. M. Heimbigner and R. King. "IDD: An Interactive Distributed Debugger." *Proc. of the 5th International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985, IEEE Computer Society Press, Silver Spring, Maryland, p. 498.

[11]J. N. W. Bei and E. G. Manning. *CGDS: A Graphical Debugger for a Distributed System.* Technical Report 85-10, Institute for Computer Research, University of Waterloo, Ont., Canada, June 1985.

[12]Bruegge, 1985.

Conclusion

In this chapter I have discussed the facilities needed for observing and modifying the states of processes when debugging a concurrent program. I have described how the process support, in the supervisor or language runtime library, can be modified to provide this. Finally I have discussed the importance of user interface design for concurrent debugging.

Chapter 7

Debugging Clients of Shared Services

Until now it has been assumed that the user program executes in isolation on a distributed system. But distributed programs typically use public servers shared concurrently with other users and programs. When a program is breakpointed and its processes are halted, the user might wish processes on any servers it is using to halt, to maintain transparency. But to do so would unreasonably interfere with other clients of those servers. Instead the debugger and the servers can co-operate to provide transparent debugging of the program, without requiring servers to halt.

Section 7.1 describes the problem and a range of approaches to solving it. The most general of these approaches has been followed in Pilgrim, and its implementation is described in Section 7.2. Section 7.3 describes how these techniques can be used by considering specific examples of services in the Cambridge Distributed Computing System. The chapter ends by considering related work.

7.1 The shared server problem

In one sense, the program which is being debugged includes not only the processes of the debuggee itself, but also those on the servers which it uses. The problem is that processes which remain running on a server, like those on the client, can affect the state being debugged as described in Section 4.2.

For instance, a server may notice lack of response in a client and cancel the client's session or abort the transaction currently in progress. If a server grants a resource to a client to be reclaimed after a predetermined time period, this period may expire while the

client is halted. If the client and server exchange time values relating to recent events, their different views of time will cause problems while the client is being debugged. The processes on the file servers, name servers, print servers and so on cannot be halted since other users would be denied service when a client was halted.

Using private copies of servers

One solution is to use private copies of servers when debugging. This is inconvenient, expensive, and often impossible. Consider duplicating the state of a file server, for instance, including all the files which might be used by the debuggee. Even then differences may exist between the private and public versions of a service—the dynamics of load provided by other users of a service will be especially hard to reproduce. Consider also the expense of duplicating special hardware, such as the printer attached to a print server. Indeed one reason for providing public servers is that it is too expensive to provide users with ones for their own private use.

Access to the software or hardware of a server may be privileged, preventing a user constructing a personal instance of a service. It is difficult to provide the necessary privileges to ordinary users without making it easier for malicious users to create counterfeit versions of a service.

Finally, using private copies of servers contradicts the requirement of target environment debugging. If a problem arises with the program after it has been released into the target environment, the program will have to be re-run using private copies of servers, with the consequent difficulties of recreating the circumstances which produced the problem.

The debugger as intermediary between client and server

In particular cases, the debugger can insulate the server from the effects of client processes halting, and insulate the halted processes from the actions the server might take. The debugger could send periodic messages to a server on the client's behalf to maintain a deadman's handle or idle handshake, or to refresh locks on resources held by the client.

This would be possible for well-known protocols, but will not work in the general case. Quite complex application specific protocols can be built using RPC and the debugger would have to understand each of these. Every time a new protocol came into use, the debugger would need to be extended to handle it.

This technique may still cause interference with other clients of the server, even though the server itself does not halt. The debuggee may halt while holding a lock on an important public resource. If the debugger automatically refreshes those locks, other clients will be denied access to the resource for long periods. The user of the debugger may be quite unaware this is going on.

Making the server aware of debugging

The best approach is for servers to handle the possibility of their clients being halted when debugging while they themselves continue running and take appropriate action. In this way they can help to preserve the same logical time frame for the client that is provided by the breakpointing mechanism of Chapter 5, while avoiding any interference with other clients.

The server is often the only member of the distributed system which can make a decision between preserving the state belonging to a client that is being debugged, and providing normal service to other clients. Servers which maintain connections to their clients already recognize whether a client is alive or dead. These debugging facilities introduce a third state: that a client is halted for debugging. If debugging is considered in the design of shared services from the outset, a number of advantages may result. If a client is halted, the server may schedule work for that client in background, or release some of the resources held by the client (as long as they can be regained later).

It may be useful for some processes on the client node to remain running. For instance it may be easier for the process in the client which handles all communication with the file server to remain running rather than complicate the code of the file servers themselves for debugging support. The term "server" is used in the following but the facilities provided and the examples given also apply to local processes which continue running.

7.2 Support for debugging clients of shared services

The debugger must give support to servers to enable them to maintain some degree of transparency for a client that is being debugged. Three procedures suffice: one implemented by the agents on all the nodes of a program being debugged, and two by the rest of the debugger which executes on another node. These have been implemented in Pilgrim.

The agent provides the procedure:

```
debugger_address, logical_time ← get_debuggee_status()
```

The first result is the network address of the debugger to which this node is connected. A special value signifies that the node is not currently under control of a debugger. The second result is the value of the node's logical clock. The logical times at each node of a program being debugged should be almost the same, and the time reported for a node which is not being debugged should equal real time—within the tolerance of the distributed clock synchronization algorithm used.

The debugger provides the procedures:

```
logical_time ← real_to_debuggee_time(real_time)
real_time ← debuggee_to_real_time(logical_time)
```

The first converts a time value for some point in the past into the equivalent client logical time; the second performs the reverse conversion. The network address provided by the agent gives the correct destination for the calls. Time values maintained by the server referring to past events can be converted into the client's time scale by calling real_to_debuggee_time, while time values supplied by the client can be converted into real time by debuggee_to_real_time.

To implement real_to_debuggee_time, the debugger maintains a log of the n events (breakpoints, etc.) for which the debuggee has been halted. Entry i of the log is of the form:

$$(halt_i, resume_i, delta_i), \quad 1 \leq i \leq n$$

Here $halt_i$ and $resume_i$ are the real times at which the debuggee was respectively halted and resumed for the ith time, and $delta_i$ is the value of the logical time delta when the debuggee was resumed. While the debuggee is halted, $resume_n$ is not yet known, and should be given a value greater than all possible time values. For real time t, the client logical time is found by:

$$\begin{array}{rll} halt_i - delta_{i-1} & \text{when} & halt_i \leq t < resume_i \\ t - delta_i & \text{when} & resume_i \leq t < halt_{i+1} \end{array}$$

Here $resume_0$ and $delta_0$ are taken as 0, and $halt_{n+1}$ is greater than all possible time values. The reverse conversion is similar, and the correct log entry can be found with a binary search in both cases.

7.3 Examples of use

To understand how the above support is useful some examples follow. The servers used in the examples are taken from the Cambridge Distributed Computing System, although none of these servers has actually been modified to provide this debugging support.

Ignoring long timeouts

The simplest way a server can use this debugging information is to determine if the client is under control of a debugger. To do so the server calls get_debuggee_status and examines the result debugger_address, to see if the client is actually being debugged. If so, the server can indefinitely extend any timeouts relating to the client. For instance, the Resource Manager allocates machines to users and programs. These resources are reclaimed by the manager after long timeouts (typically three hours) have expired. Extending the timeouts on a client's resources, at least until the end of the debugging session, will satisfy almost all situations.

Even if there is some client code which handles the expiration of these timeouts, this is unlikely to be debugged by waiting three hours for a timeout to actually expire. Rather the programmer will build a small test harness which allows particular timeout situations to be created.

```
timeout ← original timeout
client ← network address of client
client_start, debugger_address ← call get_debuggee_status() at client
        -- This is a remote procedure call

keep_waiting ← true
while keep_waiting do
    keep_waiting ← false

    wait(sem, timeout)
    except when timed_out:
        client_now, debugger_address ← call get_debuggee_status() at client

        if now() > client_now + clock_tolerance then
            -- Client logical time is slow: client may have been
            -- breakpointed during timeout.

            -- Compute how much of the timeout remains.
            time_left ← timeout − (client_now − client_start)
            if time_left > clock_tolerance then
                timeout ← time_left
                client_start ← client_now
                keep_waiting ← true
            endif
        endif
    endexcept
endwhile
```

Figure 7.1: Extending timeouts using only **get_debuggee_status**

In most circumstances the client will be unaware that the timeout does not expire at the correct point in its view of time. So this simple technique will be very effective. The server still maintains ultimate control of the resource and can reclaim it from the client if necessary.

Precisely extending time intervals

With more sophisticated use of the debugging information, a server can extend timeouts by precisely the amount necessary to match the client's logical time scale.

For instance the authentication manager, the Active Name Table, issues *temporary unique identifiers*, or TUIDs, which are capability-like objects describing rights of access or service. TUIDs must be continually refreshed before their timeouts, typically two to five minutes long, expire. Finding a bug in a client, such as accidentally omitting to refresh a TUID, would be much easier if the Active Name Table extended timeouts by the correct amount

```
timeout ← original timeout
client ← network address of client
keep_waiting ← true
while keep_waiting do
    keep_waiting ← false

    wait(sem, timeout)
    except when timed_out:
        client_now, debugger_address ← call get_debuggee_status() at client
        real_now ← now()

        if real_now > client_now + clock_tolerance then
            -- Client logical time is slow: client may have been
            -- breakpointed during timeout.

            -- Compute how much of the timeout remains.
            client_start ← call real_to_debuggee_time(real_now − timeout) at debugger

            time_left ← timeout − (client_now − client_start)
            if time_left > clock_tolerance then
                timeout ← time_left
                keep_waiting ← true
            endif
        endif
    endexcept
endwhile
```

Figure 7.2: Extending timeouts using get_debuggee_status and real_to_debuggee_time

when the client was under control of the debugger. With the mechanisms provided by Pilgrim, there are two ways for a server to correctly extend a timeout. An algorithm for the first method is shown in Figure 7.1. The server obtains the client's logical time just before the timeout begins. The server then waits on a semaphore, sem, which is notified when an event such as receiving a refresh from the client occurs. The second argument to the semaphore wait is the timeout value in seconds. If the timeout expires before the semaphore is notified, the exception timed_out is raised. This is handled by the block of code beginning "except when ...". In this case the logical time is obtained again from the client. If the client has not been breakpointed in the interim, the difference between these two times should equal the length of the timeout in real time. If a breakpoint has occurred, the period remaining in the timeout can be calculated and used as the new timeout value as another iteration of the loop is performed.

This scheme has the disadvantage that an invocation of get_debuggee_status on the client is required at the start of every timeout, even when that client is not being debugged, and

even when the timeout will not in fact expire. An improved method, shown in Figure 7.2, avoids this work unless the timeout does expire. However it then involves a call to both get_debuggee_status and real_to_debuggee_time.

The clock tolerance used in these methods for extending timeouts must be large enough to prevent the server consuming too much processor time while waiting for a client to resume. Otherwise the server will effectively busy wait if the clients is halted very near to the end of the timeout period.

Resource contention with other users

The methods just described are applicable to a server which manages some set of shared resources for which many clients contend. When a resource is allocated to a client a timeout on the allocation is usually imposed. However extending that timeout when debugging may be wrong if the resource is very scarce and other clients require it. A decision must be made between delaying these other clients or reclaiming the resource (and thus upsetting the debugging session).

The server could provide some administrative commands by which the scarcity of a resource and the importance of some debugging session could be indicated. This might be unwieldy to implement or use. A simpler approach has the server extending a timeout on some resource allocation until a client, not under control of the same debugger, requires the resource. At that point the resource is reclaimed and reallocated.

Converting time data

A client that is being debugged may notice inconsistent timing if it receives explicit time values from a server, for instance as the time of last modification of a file. A client may also supply time values to the server, although this is less common. A server can convert these time data using real_to_debuggee_time and debuggee_to_real_time. However this may not be enough to preserve complete transparency without also reproducing the content of the file at an appropriate time in the past. Where necessary the user should ensure that external data such as files are preserved throughout the debugging session.

Limitations

There are aspects of transparency which servers cannot achieve with the support described. The time of future events cannot be converted correctly and clients cannot time real world events over a breakpoint. However transparency is maintained about events within the program being debugged, even when time data about such events is communicated through other parties such as servers. Converting time intervals and time values in the past into the client's logical time seem reasonable steps for a server to take, and do not cost a great deal. It is important to recognize limits to how much network servers should do to maintain transparency, since their main purpose is still to provide reliable and efficient service to all of their clients.

The three debugging support procedures constitute a security problem. A malicious program could provide implementations of these procedures allowing it more favoured treatment from servers. Thus some kind of authentication is needed so servers can detect misuse. This issue is discussed in Chapter 8.

7.4 Related work on debugging clients of shared services

The only other work which considered the problems of debugging in a multi-user distributed system is the Tripos Remote Debugger. This is a machine language debugger for BCPL programs under the Tripos operating system. Tripos,[1] like Mayflower, is used in the Cambridge Distributed Computing System. The debugger was run on one machine and connected to the debuggee running on another. The remote debugger takes over the task of sending refresh messages to the Resource Manager, preventing it from reclaiming the machine after it crashes, and thus avoiding an abrupt termination of the debugging session.[2] This mechanism only applied to the machine in which the debuggee was running. In contrast my techniques apply to any resources the debuggee may have obtained, and any services the debuggee is using.

---

[1]M. Richards, A. R. Aylward, P. Bond, R. D. Evans, and B. J. Knight. "TRIPOS—A Portable Operating System for Mini-computers." *Software—Practice and Experience*, 9(7), July 1985, p. 513.

[2]See p. 55 D. H. Craft. *Resource Management in a Distributed Computing System.* Ph.D. Dissertation, Technical Report 77, Computer Laboratory, Cambridge University, March 1985.

7.5 A strategy for debugging clients of shared services

The techniques described in this chapter allow a program which uses shared network services to be debugged interactively. The debugger and the services concerned cooperate in maintaining the logical time of the debuggee and in avoiding interference with other users of those services. However the program being debugged is not entirely insulated from outside influence and is not prevented from potentially affecting other users of a distributed system. So the programmer must use some discretion, and balance her needs when debugging a program against those of other users in the system.

In the early stages of debugging a program, the servers it uses can be shared with other clients, using the facilities described in this chapter. But the data objects operated on by the debuggee should be private copies (e.g. private files rather than public shared ones). It is wise to do this, not only to avoid delaying other clients while debugging, but also because the possibly faulty client may carry out erroneous operations on the object or even destroy it.

In the final stages of debugging and tuning and after the program goes into service, the client will be operating on live data, shared with other users. It will still be possible to examine the program under the debugger, but primary consideration should be given to other users of the distributed system. By this stage, the program may well have its own clients who also deserve consideration. Use of the debugger in this situation should probably be restricted to examining the program state. Extensive debugging would require a return to one of the private or semi-private environments used earlier.

Chapter 8

The Debugger as a Distributed Program

The Pilgrim debugger is itself a distributed program. Many of the issues which arise in the design of any distributed program must be faced by the designer of a distributed debugger: distribution of function, concurrency and synchronization, error detection and recovery, authentication, and efficiency. This chapter examines these issues, and how they were addressed in the design of Pilgrim. It concludes by considering alternative ways of organizing a distributed debugger.

The largest component of Pilgrim is the central debugging node, referred to simply as the debugger. The debugger provides the user interface and manipulates the source-to-object information produced by the compiler and linker. The rest of Pilgrim consists of the agent: a piece of debugging support code included in each node of the program by the linker. The agent communicates with the debugger and performs low-level debugging actions on its behalf such as setting and clearing breakpoints and accessing memory at the node it runs on.

8.1 Distribution of function

What factors are important when deciding whether to implement some function in the debugger or in the agent?

A copy of the agent is included in every node of every program. It is present even when a program is not actually being debugged. In keeping with the goal of target environment debugging, the agent must impose little overhead on the normal execution of the program. The extra code and data space required for the agent must not be so large that users are

tempted to leave out debugging support when the program is released to the production environment. Thus a small, general set of primitives should be provided by the agent.

The performance of these primitives is also important. Commonly used features in the debugger, such as setting breakpoints and examining variables should take a reasonable amount of time (e.g. less than a second). Performance is affected not only by the complexity of the primitives themselves, but by the speed of the network over which the debugger and agent communicate. Throughout this chapter it is assumed that the debugger and agents communicate over a local area network exhibiting high bandwidth and low delay (less than 10ms end-to-end delay).

Sections 8.2 to 8.6 detail the functionality of the agent and present the remote procedures that it provides. Section 8.7 describes the remote procedures provided by the debugger. Together these sections summarize most of the implementation details described in preceding chapters. The remaining sections discuss communication, concurrency, and authentication issues, and describe related work on distributed debugger design.

8.2 Connection and disconnection to the debugger

A debugging session is begun by the debugger calling the agent's connect procedure:

```
linker_debug_file_name,
link_time_ID,
load_start_address,
logical_time ← connect(debugger_address,
                       session_ID,
                       override,
                       other_nodes,
                       remote_print_stream)
```

The debugger_address is the network address to which the agent sends any RPCs.

The session_ID is an identifier unique to this agent-debugger connection. It is supplied upon all agent-debugger communication in order to reject stale or erroneous calls. It does not protect against malicious calls (authentication is addressed later in the chapter).

Normally the agent will refuse connection to one debugger if it is in the midst of a session to another, for instance when one user attempts to debug a program already being debugged by some other user. However, if the override flag is set in the new connect call, the current session is abruptly terminated and the agent accepts the connection to the new debugger. This can be used in cases when the first debugger has crashed, or is unable to be contacted. The agent does not try to detect such situations but relies on the user deciding to start a new debugger session after noticing lack of response.

The other_nodes argument is a list of the network addresses of the other nodes of the program. It is used to initialize the remote halting mechanism described in Chapter 5, in which a message is multicast to all the other nodes of a program.

The remote_print_stream is an output stream which the agent and user code on the debuggee can use to output information to the debugger's terminal. It is useful because, in a distributed environment, the debugger's terminal may be some distance away from output devices directly connected to a debuggee node.

If the agent accepts the connection it then initializes its data structures and returns several results from the connect call. A unique identifier is generated by the linker and stored in the debugging file. The name of this file, and the link-ID are stored in the code file. The agent extracts these two values and returns them as the first two results of connect. The debugger reads in the linker debugging file and verifies that the link-ID in the file is the same as that supplied by the agent, thus avoiding accidents resulting from debugging files and code files which are out of step. Mayflower loads the code file for a node into a contiguous area of memory. The start address of this area is the third result of connect. The final result is the current logical time for the node, which is used to initialize the debugger's breakpoint log (see Chapter 7).

When the debugger wishes to terminate a debugging session it simply calls the disconnect procedure:

```
disconnect(session_ID)
```

The agent clears all remaining breakpoints, resumes (if possible) any halted processes, and switches the debugging support into a dormant state until the next connect call. The

disconnect procedure also resets the logical clock to real time. The effects of this are unpredictable so it is often unwise to allow the debuggee to continue executing at the end of a debugging session.

The get_debuggee_status procedure is relevant here:

```
debugger_address, logical_time ← get_debuggee_status()
```

It is used by servers who wish to know if one of their clients is being debugged, and if so the value of its logical clock. The debugger_address returned is that supplied in the connect call.

8.3 Memory access and update

The agent provides procedures to access and modify values stored in primary memory or in the registers of a process:

```
object_token ← get_memory(session_ID, memory_address)
               set_memory(session_ID, memory_address, object_token)
object_token ← get_register(session_ID, process_ID, register_number)
               set_register(session_ID, process_ID, register_number, object_token)
```

In essence an object_token is the value of a 32 bit memory word or register. However things are complicated by Concurrent CLU's use of a garbage collected heap to store much of the data of a program.

Consider this example. A memory word, M, contains the only pointer to an object, X, stored in the heap. The debugger calls get_memory on M, which for the purposes of this example simply returns the 32 bit pointer to X. The debuggee continues execution and M is overwritten by a variable assignment. A garbage collection occurs and X is collected. The debugger now holds a pointer to a non-existent object, or worse, a pointer into the middle of some other object allocated in the space vacated by X. This problem is more serious for the Concurrent CLU implementation since a relocating garbage collector is used, in which non-garbage objects are moved to a contiguous area of the heap at the end of a garbage collection. Thus any heap object may change its address unexpectedly.

The solution provided in Pilgrim is to use a *token*, consisting of a unique identifier, to refer to a heap object remotely. (Tokens are sometimes called *cookies*.) Rather than passing a heap address to another node, a token is created and the pair {*token, pointer to heap object*} stored in a table on the node. Only the token is passed remotely, while the heap pointer ensures that the object is not garbage collected. When the remote token is passed back to this node in a later procedure the heap object referred to can be retrieved.

Care must be taken to avoid retaining heap objects indefinitely with tokens. Individual tokens can be deleted from the table by calling the forget procedure:

```
forget_object(session_ID, object_token)
```

All the tokens are deleted when the debugging session ends.

Only pointers to objects within the heap are represented by tokens. Pointers to non-heap objects, and scalar values such as integers and booleans are simply represented by their 32 bit values.

The remote memory reference problem arises in other automatic storage management schemes such as reference counting, and can be tackled in the same way. In languages which provide explicit unchecked storage management primitives (such as Pascal's new and dispose, and Ada's unchecked_deallocation) the deletion of objects to which remote references exist cannot be prevented. The best that can be done is to detect when such objects are deleted by monitoring all calls to dispose.

The agent's remote memory procedures are only used on words of the call stacks of processes, and to access own variables. Access and modification of structured heap objects is performed by invoking local procedures on the debuggee (described next). For instance, to evaluate the expression a[4], which obtains the fourth element of an array, the fetch procedure provided by the array cluster is invoked. This is safer and simpler than having the debugger perform the address calculations itself and access the appropriate word of the heap directly.

8.4 Procedure invocation and object printing

The debugger calls local procedures on the debuggee using the invoke procedure:

```
result_objects ← invoke(session_ID, routine_descriptor, argument_objects)
```

Concurrent CLU has a number of control abstractions, in addition to the local procedures found in conventional languages. An *iterator* computes a sequence of values and is invoked by a looping construct which assigns each value to the loop control variable on successive iterations of the loop. A *gate* protects a block of code, performing some action before and after the block regardless of how it exited. Gates are often used to provide critical regions. And of course Concurrent CLU provides remote procedures. All these control abstractions are collectively referred to as *routines* and all may be invoked from the debugger. The particular routine to be invoked is identified by the routine_descriptor.

The argument_objects are object tokens describing the arguments to the routine. Each of these objects must already exist on the node. They could be the values of variables in the debuggee, or they could have been constructed by earlier invoke calls. A similar list of tokens for result_objects is returned. If the routine raised an exception, the exception name and its associated objects are returned instead.

The invoke procedure is used for all expression evaluation in Pilgrim, except for the operations of the built-in types (such as integers, booleans, and strings) whose implementation is fixed. This ensures that, if the programmer has written a new cluster to implement a type, or even if there are different implementations of the same type in different nodes of the program, the correct effects of routine invocation will occur.

Object printing

To display objects in the debuggee program, Pilgrim invokes print procedures which reside in the debuggee program itself. By convention, the cluster for a type T provides a print procedure which displays the abstract state of T objects:

```
print(T, print_stream)
```

A print_stream is an output stream for printing structured text. The built-in types provide their own print procedures too. Pilgrim simply calls these procedures using invoke, and passes a special remote_print_stream which redirects output strings to the debugger for display on the user's terminal. This stream was one of the arguments to the connect procedure described in Section 8.2.

Contrast this with the more conventional approach in which low-level representations of objects would be transferred to the debugger, and displayed using a small fixed number of output routines for the built-in types. Pilgrim's method of object printing promotes debugging at the abstraction level, by using the print procedures provided by the programmer for her own abstract data types. If the programmer provides different print procedures in different nodes of the program, that will be reflected correctly when objects are displayed in the debugger.

Printing processes and procedure variables

A difficulty arises when displaying the objects of the process type. To display them adequately requires source-level information not present in the run-time representation of a process (see Chapter 6). For instance, the process's current execution point should be displayed in source-language terms, but only the machine-level value of the program counter is available to the print procedure on the debuggee. Pilgrim avoids storing any source-level debugging information on the debuggee node to reduce its size, in keeping with the goal of target environment debugging.

So the print procedure first checks whether the debuggee node is currently connected to a debugger. If so the print procedure calls back to the debugger requesting it to display the process in source-language terms. Otherwise it simply displays the process identifier.

This might appear a rather cumbersome solution. Why doesn't the debugger notice when it is about to print a process object and not attempt to invoke the print procedure at all? The reason is that a process may be displayed as part of the state of some higher level object such as a user-defined abstract data type. The user ought to be able to write a print procedure which calls the print procedure for process, and have that process displayed in source-language terms.

Displaying the values of variables which refer to procedures causes the same problem, because such a variable merely contains the start address of a procedure, but not its name and type. The same solution is used.

8.5 Process information and control

The debugger can obtain the current state of a process by calling the get_process procedure:

process_info ← get_process(session_ID, process_ID)

The process_info contains:

- the process identifier
- the process *run state* which consists of one of the following:
  - running
  - waiting (along with the queue-ID the process is waiting on)
  - kicking (and the queue-ID)
  - dead
  - performing an SVC
  - halted because of a hardware exception
  - halted because of a software detected event
- the thread priority
- whether the process belongs to the halt group for this node
- part of the saved register set
- a sequence of 10-20 words from around the top-of-stack pointer register
- the code address of the fork expression which created this process (to assist process naming)

The first five items are derived from the get_thread_info SVC described in Chapter 6. The process run state is a refinement of the thread's schedule state. A process which is

currently waiting on a queue-ID belonging to the supervisor is reported as performing an SVC, a more meaningful state. The agent determines which processes are halted because of some hardware exception or software detected event, and reports them as such. The get_thread_info SVC would simply report these processes as running or waiting inside some piece of the agent.

The register values and the words from the stack are supplied as simple 32-bit quantities, without using the object token mechanism. These values are used to interpret the very top of stack when finding the topmost well-formed procedure call frame as described on page 65 in Chapter 6. Normally the debugger looks at only two registers and one or two words of stack to do this, but the agent does not know which particular values are required in each case so it supplies most of the registers and 10-20 stack values. Converting these to object tokens would be expensive and is not necessary for this algorithm. If subsequently the value of a heap object pointer stored in a register or on the stack is needed, get_register or get_memory must be used.

Breakpoints

Setting and clearing code breakpoints is implemented by the following procedures:

```
set_code_break(session_ID, PC_value, break_on_branch)
clear_code_break(session_ID, PC_value)
```

The break_on_branch argument is a boolean which applies to breakpoints set on conditional branch instructions. In some cases the breakpoint should only be triggered if the branch is actually taken. This is used to to handle a branch-chaining optimization performed by the Concurrent CLU compiler.

Breakpoints on exceptions and the single-step execution of processes have not been fully implemented yet. However the agent interface procedures have been defined as follows. Exception breakpoints may be set and cleared using:

```
set_signal_break(session_ID, exception_name)
clear_signal_break(session_ID, exception_name)
```

To execute a process one statement at a time the following procedures are used:

```
start_stepping(session_ID, process_ID, low_PC, high_PC)
stop_stepping(session_ID, process_ID)
```

The start_stepping procedure will execute the given process in *trace* mode until it is about to execute an instruction outside the range [low_PC, high_PC]. This range will chosen by the debugger to correspond to the start and end of the code for a single statement. A similar technique was used in the Blit debugger.[1]

Process halting

The halting algorithm, described in Chapter 5, is initiated whenever a breakpoint or program error occurs. The debugger can halt and resume the debuggee itself by calling:

```
freeze(session_ID, freeze_or_unfreeze)
```

The last argument is a boolean indicating whether the node should be frozen or unfrozen. The debugger can control an individual process using:

```
change_process(session_ID, process_ID, process_action)
```

The process_action is one of the following:

- add the process to the halt group
- remove the process from the halt group
- resume the process which has halted at a breakpoint or because of an execution error
- indicate that the process is never to run again (usually because it has encountered some execution error from which it cannot resume)

[1] T. A. Cargill. "Implementation of the Blit Debugger." *Software—Practice and Experience*, **15**(2), February 1985, p. 153.

8.6 RPC information

Procedures to trace RPCs between nodes were described fully in Chapter 3. They are repeated here for completeness (and now include the session_ID arguments omitted earlier):

```
reply_address, call_ID ← locate_RPC_source(session_ID, callee_frame)
caller_frame           ← get_RPC_caller(session_ID, reply_address, call_ID)

call_address, call_ID  ← locate_RPC_destination(session_ID, caller_frame)
callee_frame           ← get_RPC_callee(session_ID, call_address, call_ID)
```

8.7 Procedures provided by the debugger

Finally, the debugger provides some procedures which are used by the agent and by public servers. When a process halts because of single-stepping, because of an execution error or at a breakpoint, the agent gains control and reports the event to the debugger using:

```
report_to_debugger(session_ID, agent_address, program_event, process_info, event_time)
```

The agent_address is the network address of the agent. The program_event is whatever caused the process to halt, and the event_time is the logical time when it happened. The process_info is the same as that returned by the get_process procedure above.

To provide source-level display of processes and routines, as described in Section 8.4 on page 88, the following two procedures are provided:

```
print_process(session_ID, agent_address, process_ID)
print_routine(session_ID, agent_address, routine_address)
```

The routine_address is the memory address of the beginning of the routine to be printed.

The debugger implements two procedures to allow servers to convert between client logical time and real time (see Chapter 7):

```
logical_time ← real_to_debuggee_time(real_time)
real_time ← debuggee_to_real_time(logical_time)
```

command	elapsed time	number of RPCs	total RPC data size
connect	1.47s	2	352 bytes
set code breakpoint	.12s	1	84 bytes
clear code breakpoint	.13s	1	80 bytes
resume from breakpoint	.35s	2	164 bytes

Figure 8.1: Performance breakdown of some Pilgrim commands

8.8 Performance

An idea of the cost of various debugging actions is given in Figure 8.1. The times are approximate, being the result of only ten repetitions of each command, with a discretization error of $\pm$.02s. These figures are intended only to demonstrate that Pilgrim has acceptable interactive performance. The overhead on each RPC is about .05s, including argument and result processing at each processor, and network transmission time. These RPC times are slower that those quoted in Chapter 3 because the debugger was running on Vax/Unix which has a slower RPC system.

The agent increases the code size of a typical debuggee node by 56K bytes. Existing Concurrent CLU programs range in size from 100 to 400K bytes.

The remaining sections of this chapter discuss general design issues for a distributed debugger.

8.9 Communication: using RPC

Almost all agent-debugger communication uses the Concurrent CLU RPC mechanism. The only exception is the algorithm for halting distributed processes which uses its own low-level protocol in the interests of efficiency. Using RPC made it easy to experiment with different agent interfaces when developing Pilgrim. The RPC library must now be included in every program, but all distributed programs which run under Mayflower use RPC in any case.

The main drawback is that the agent, and thus the whole activity of debugging, depends on the correct operation of the RPC mechanism. For example, if a poorly-designed user

program exhausts some resource within the RPC mechanism (such as the pool of free network packet buffers), the debugger may be unable to contact the agent. Also Pilgrim cannot be used to debug the RPC mechanism itself. This is a problem for our research group because the Concurrent CLU RPC system is undergoing continuous development, but it should not be a problem outside a research environment.

Retry policy

Concurrent CLU RPC provides two sets of RPC semantics: *maybe* and *exactly-once*. The *maybe* RPCs are used for calling the connect and disconnect procedures, to allow precise control of the number of retries performed before assuming the node is unable to respond. The *exactly-once* protocol is very persistent and typically retries for 100s before giving up. This would be extremely inconvenient if, for instance, the user attempted to connect to a machine which was not running Mayflower and Concurrent CLU, or to disconnect from a dead machine.

The session-ID performs an extra duty on connect and disconnect calls by detecting duplicate calls from the same debugger. This works only because there is at most one of each of these RPCs during a debugging session.

All the other agent-debugger RPCs use *exactly-once* calls, simplifying the programming but making the caller process dependent on the correct termination of the RPC. For instance, deadlock may occur in the agent and cause a process in the debugger to wait indefinitely for the reply to an RPC. This is part of a more general problem of agent-debugger dependency. How it is solved is described in the next section.

8.10 Concurrency and safety in the debugger

The correct operation of the debugger should not depend on the behaviour of the debuggee program: the debugger must insulate itself from errors in user code. The main part of the debugger runs on a separate computer from the user program, so it is protected from most conceivable hardware and software errors on the debuggee.

The agent is more vulnerable since it executes in the same address space as the debuggee. However a type-safe language like Concurrent CLU prevents many disastrous errors entirely, such as overwriting arbitrary areas of memory. The agent is written in Concurrent CLU and shares the heap with the debuggee, so a bug whose primary symptom was heap exhaustion might prevent the agent from operating, and would be very difficult to debug. This problem is similar to the dependency on the RPC system described in the previous section.

For performance reasons, the agent must execute on the same computer as the debuggee, but it ought to reside in a separate domain as described in Section 5.2. There the motivation was to prevent deadlock between a user process and the agent when the node was halted. Here it is to make the agent independent of program failures in the debuggee.

When the debugger invokes user code

Whenever the debugger invokes a procedure on the debuggee there is the risk of that procedure crashing or deadlocking. To guard against the first risk, the user procedure is invoked within the scope of a Concurrent CLU exception handling statement which catches all software exceptions. To avoid deadlock, the debugger could associate a timeout with the invoke RPC. But how could this timeout value be chosen? The debugger has no knowledge of the expected execution time of a procedure and so cannot decide when it might have deadlocked.

Instead the command-loop process which handles Pilgrim's user interface creates a separate slave process to obey each command and then waits for the slave to return. If it does not return within a fixed timeout period (usually 10 seconds) the command-loop process resumes and will accept further commands from the user. However the slave is not destroyed at this point, but continues executing the command, detached from the command-loop process. There is an explicit join command with which the user can again wait for the completion of one of the detached slaves. Any process on the debuggee performing an RPC for a slave may be breakpointed, halted, and resumed just as any other user process can, without causing deadlock in the debugger.

The timeout applies to all RPCs the slave performs, not just calls to the invoke procedure,

and so protects against the failure of any agent procedure. The timeout period may be changed by the user too.

Sometimes the user wishes to invoke a debuggee procedure which she knows will never return (e.g. the initial procedure of the program). For this the fork command can be used, which explicitly creates a slave and does not wait for its completion.

Concurrency in the debugger

The concurrent command execution just described greatly complicates the debugger. In any case a debugger of concurrent programs must cope with the concurrent execution of the debugger and processes in the debuggee. Thus all debugger data structures are designed to be safe under concurrent access.

Many actions of the debugger require that the debuggee program be halted—using the process halting method of Chapter 5. For instance, interpreting the call history of a process is impossible if that process is still executing. Another example was mentioned in Chapter 6, when a process is resumed from a code breakpoint. While the *trap* instruction is temporarily removed and the process steps over the breakpoint, all other processes on that node must be halted, in case one fortuitously executes the breakpoint location and misses the breakpoint.

8.11 Authentication and security

The debugging interface provided by the agent presents great opportunity for security breaches. A program masquerading as a debugger could connect to the agent and access or modify any part of the debuggee program's state. Simply by monitoring communication between the agent and debugger, valuable information might be gained. An intruder could send messages to the debugger appearing to come from an agent to initiate the halting of the debuggee or cause other spurious events, perhaps to deny service to other users and programs. A modified agent could gain more favourable treatment from a server by pretending that it was currently halted, e.g. to cause timeouts to be extended.

Since the agent is intended to operate in the target environment, these problems cannot be ignored. Otherwise users will be obliged to leave out debugging support to provide adequate security. The current implementation of Pilgrim makes no attempt to solve the security problem. The session_ID supplied with each agent-debugger interaction merely protects against non-malicious errors.

To prevent these threats we must ensure:

- the agent knows it is conversing with a debugger authorized to debug the debuggee program of which the agent is a part
- the debugger knows it is conversing with a node of the debuggee program
- no agent-debugger communication is comprehensible by parties other than the debugger and the debuggee
- a server knows that the logical time information it receives came from an agent or the debugger, and is correct

Secure agent-debugger communication

We can satisfy the first three requirements using an authentication service and an authentication protocol such as those devised by Needham and Schroeder[2] and more recently by Otway and Rees.[3] In these protocols each *principal* (a named participant in the communication) possesses a secret key known only to itself and a secure authentication service. Each principal has a way of locating the authentication service which stores principals' names and secret keys. At the successful completion of the authentication protocol both principals possess a conversation key, *CK*, unique to their conversation. Each principal knows that any communication it receives that is encrypted with *CK* originated at the other, and that any communication it transmits, encrypted with *CK*, will be received only by the other.

---

[2]R. M. Needham and M. D. Schroeder. "Using Encryption for Authentication in Large Networks of Computers." *Comm. of the ACM*, **21**(12), December 1978, p. 993.

[3]D. Otway and O. Rees. "Efficient and Timely Mutual Authentication." *Operating Systems Review*, **21**(1), January 1987, p. 8.

The task when applying this protocol, is to decide exactly who are the principals to be authenticated, how they obtain their keys, and how they keep them secret. Ultimately authority must be granted by human principals, and keys protected by lock-and-key and other means outside the computer system. It is assumed, for instance, that users of the distributed system authenticate themselves to the authentication service with passwords which they must keep secret.

The users we are interested in are the programmer of the debuggee program, and the person wishing to debug the program. Consider a program for auditing a company's accounts which was written by a programmer called *Jill*, and which is to be debugged by someone called *Jack*. (This separation of responsibility is common in the development of sensitive software.)

The audit program should possess Jill's secret key and accept debugging connections only from a debugger authenticated as Jack's. Jack should supply the debugger with his key and instruct it to communicate only with a program authenticated as Jill. In practice Jill and Jack should not give away their secret keys so readily but create new identities in the authentication service called *Jack_as_debugger_user* and *Jill_as_programmer*, (or even *Jill_as_programmer_of_the_audit_program*), and use them. If the secret key of one of these more specific identities was compromised, the consequences would be less disastrous.

We might also have a principal called *Pilgrim_debugger* with its secret key hidden in the code of the debugger. With this the agent would have some confidence that Jack was not inadvertently running some pirated version of the debugger containing a Trojan horse which could transmit information to some third party without Jack's consent. The *Pilgrim_debugger* principal also prevents Jack from using his own version of the debugger, but this does not really provide any more security since the debugger already gives Jack complete access to the debuggee program. Storing a secret key in the code of the debugger is only as secure as the file system in which the debugger code file is stored. Jack should take reasonable precautions to ensure that he is running a legitimate version of the debugger.

Providing a *Pilgrim_agent* principal and encoding its secret key in the agent binaries might provide similar assurances to Jill against Trojan horses. However these binary files will be

widely accessible and it will be more difficult to ensure long term secrecy of the agent's key than the debugger's.

This authentication scheme can be extended to cope with a distributed program which is subdivided into several protection domains. Thus each programmer has responsibility for only one domain, and should not be able to modify or debug other domains. To enforce this, each domain should consist of a subset of the nodes of the program, and principals will be created who are each entitled to debug only one subset. Essentially each subset behaves as a separate distributed program which communicates with the other subsets in the same way as a program communicates with public servers on a distributed system. In particular, each subset should use the techniques described in Chapter 7 to preserve the logical time scale for the other subsets it interacts with.

Authenticating the debugger to a server

The final requirement is that logical time information provided to a server by the debugger and agent is authentic. This is more difficult to satisfy. The server could authenticate either the *Pilgrim_agent* or the *Pilgrim_debugger* principals when calling the logical time procedures, subject to the warnings above about the secrecy of their keys. However a simple sequence of commands issued to a legitimate debugger can convince a server that the debuggee has halted when most of its processes continue executing. The user first halts the debuggee, and then removes almost all processes from the halt group. Thus most of the processes continue execution normally, and presumably gain more favourable treatment from the server. This threat is not as serious as the previous ones and many servers will not require this level of authentication.

8.12 Alternative debugger organizations and related work

The agent-debugger structure of Pilgrim is a popular way of organizing distributed debuggers. It was used by Schiffenbauer[4] who termed the agent the *nub*, while in the IDD

[4]R. D. Schiffenbauer. *Interactive Debugging in a Distributed Computational Environment.* M.S. Dissertation, Technical Report MIT/LCS/TR-264, Laboratory for Computer Science, MIT, Cambridge, Mass., September 1981.

debugger[5] the agent is termed the *satellite*. The same structure is advocated by Garcia-Molina *et al.*[6]

Also relevant are *remote* debuggers, that is debuggers which run on a host computer and debug non-distributed programs running on another computer. In this class is the Blit debugger[7] which runs on a Vax/Unix host and debugs sequential programs executing on an MC68000-based graphics terminal connected to the host by a dedicated serial line. A debugging process in the terminal performs requests on behalf of the main debugger process in the host. The protocol between the host and terminal processes is similar in many ways to Pilgrim's agent-debugger interface, although it is not implemented with RPC. The terminal process provides breakpoint and single-stepping primitives essentially the same as those in Pilgrim. In other cases, the terminal process tries to ensure more information is supplied to the host for each primitive because Unix is quite slow to respond to messages received on the serial line.

The Distributed Incremental Compiling Environment[8] (DICE) can also be considered as a remote debugger. In DICE, programs are compiled using an incremental parser and code generator. Interactive debugging activities such as setting breakpoints and displaying variables are achieved by editing new statements into the source program. The object program running on another computer is updated, after which it is usually able to resume execution. Occasionally the updates are too extensive and the program must be restarted from the beginning. The interface between the compiler and the debuggee program comprises primitives to read or write a block of memory (mainly used for updating the object code), to read or write the registers, and to halt or resume the debuggee. Incremental compilation provides a good basis for an integrated programming environment, and obviates the need for a separate debugger. However such a compiler is currently more complex to

---

[5]P. K. Harter, D. M. Heimbigner and R. King. "IDD: An Interactive Distributed Debugger." *Proc. of the 5th International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985, IEEE Computer Society Press, Silver Spring, Maryland, p. 498.

[6]H. Garcia Molina, F. Germano Jr., and W. H. Kohler. "Debugging a Distributed Computing System." *IEEE Trans. on Software Engineering*, **SE-10**(2), March 1984, p. 210.

[7]Cargill, 1985.

[8]P. Fritzson. "Preliminary Experience from the DICE system, a Distributed Incremental Compiling Environment." *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, August 1983, p. 113 (published as *Software Engineering Notes*, 4(4) and SIGPLAN Notices, 8(8)).

design than a conventional compiler and tends to produce poorer quality object code. The prototype DICE system compiled very slowly and could only handle tiny source programs.

Is the agent necessary?

The MC68000-based computers in the processor bank of the Cambridge Distributed Computing System include a separate network interface processor, termed the *Mace*. Its main duty is to perform I/O to the Cambridge Ring on behalf of the 68000, but it also provides some primitives allowing another node to access and control the 68000, both for debugging and processor bank management.[9] There are functions to halt, run, interrupt, and reset the 68000, and to read or write a block of its memory.

These Mace primitives are sufficient to perform most debugging activities, perhaps obviating the need for the agent at all. However the result would be very slow because each primitive requires a message exchange over the Ring, and many messages are needed for common debugging activities. Setting a code breakpoint can be achieved by first reading the instruction word to be breakpointed, and then overwriting it with a *trap* instruction. Stepping a process over a breakpoint would use four operations: replacing the overwritten instruction, single-stepping the debuggee by modifying one of the process's saved registers, resetting that register, and replanting the *trap* instruction. More complex activities provided in Pilgrim such as invoking procedures on the debuggee, or obtaining information about RPCs would require many more messages unless some debugging support code existed on the debuggee program too. To date the only use of the Mace debugging primitives is by the Tripos Remote Debugger described in Chapter 7.

The advantage of the Mace debugger interface over the agent-debugger structure of Pilgrim is that the Mace is not susceptible to failure because of the actions of a faulty program in the 68000.

---

[9] N. H. Garnett. *Intelligent Network Interfaces.* Ph.D. Dissertation, Computer Laboratory, Cambridge University, 1983.

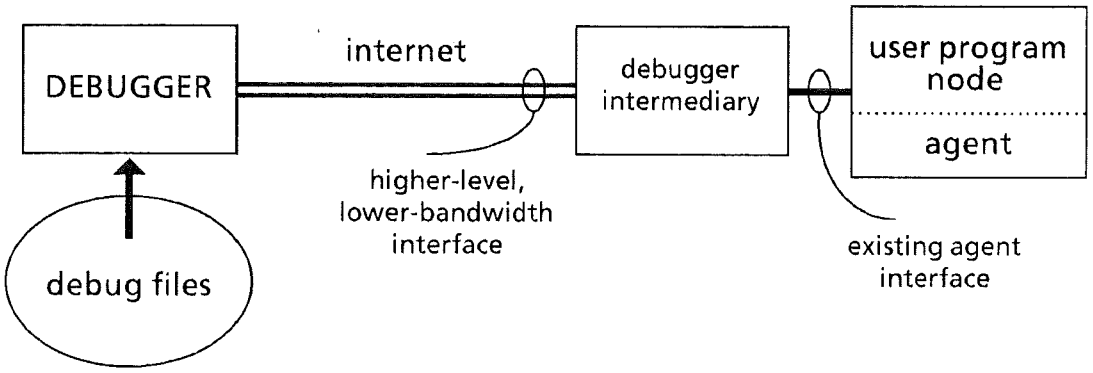

Figure 8.2: A debugger structure for a wide area network environment

Pilgrim in a wide area network environment

Pilgrim was designed to operate on a local network. If the debugger and agent were separated by a wide area network or internet, unacceptably low performance would result. Some user commands perform several RPCs. For instance displaying one procedure frame requires at least two RPCs for each variable in the procedure. On a wide area network each RPC might take over one second, resulting in poor interactive response times. However debugging over a wide area network or internet could be performed acceptably with the modified debugger structure, shown in Figure 8.2. Here part of the debugger close to the user communicates higher level requests over the internet to another part of the debugger situated on the same local network as the debuggee.

Conclusion

This chapter has discussed overall design considerations of the Pilgrim debugger. The first half of the chapter discussed the division of responsibility between the agent and debugger, and detailed the agent-debugger interface. Then Pilgrim's use of the Mayflower RPC system and how the debugger is insulated from errors and deadlock in the debuggee was discussed. Authentication issues were examined and a scheme for ensuring secure communication between the agent and debugger was presented. Finally alternative debugger organizations were considered. Much of this chapter was specific to the organization of Pilgrim. However similar decisions present themselves in the design of any distributed debugger, and similar choices will have to be made.

Chapter 9

Conclusion

In this thesis I have shown how to provide source level debugging tools for distributed programs running in the target environment. Highlights have included: the debugging facilities for RPC and how to implement them; a definition of transparent debugging and a breakpointing mechanism to achieve it; the problems of debugging clients of shared servers and some solutions; and a discussion of the system support needed for debugging concurrent processes and the implications for operating system design. Throughout I have emphasized target environment debugging in which the debugging support is inexpensive enough to be available at all times.

9.1 Evaluation

How successful are the debugging mechanisms presented here, and how practical are they to implement? Almost all of the debugging mechanisms have been implemented. In important cases I have measured their performance to demonstrate that they are indeed cheap enough to be provided in the target environment.

Yet to be demonstrated is whether these debugging facilities actually help people to produce correct and effective distributed software. This can only be determined by having real programmers use these techniques on real programs. So far Pilgrim has been used only by myself on small test programs, and to help debug Pilgrim itself. Pilgrim is to be released shortly to our research community. Concurrent CLU and the Mayflower environment are used by about ten people on a number of substantial projects. This should provide realistic feedback on Pilgrim and the ideas in the thesis.

9.2 Applying these ideas

In this thesis I have concentrated on practical debugging techniques for current languages and existing distributed systems architectures. My ideas can be applied now to ease the development of distributed software. Here I discuss how this should be done.

The distributed program development environment

A debugger such as Pilgrim should be used as part of a complete distributed program development environment. Such an environment should also include tools for program specification, program analysis, performance monitoring, and source code management. There is a place too for more expensive debugging tools based on event logging and program replay. These tools would be available only in the development environment.

To support target environment debugging, there must be a way to communicate between the target and development environments, in particular to transfer object programs and to support agent-debugger communication. It is advantageous if this communication is possible over long distances to provide debugging services to remote sites, subject to considerations of security.

Practical obstacles to building debuggers

Programming environments of the kind just described are now available, but frequently the source language debugger is the last tool to be provided, if it is provided at all. There are at least two reasons for this. Firstly, it is possible, though not pleasant, to write and debug programs without the help of a source language debugger. So essential tools, such as the compiler, are given higher priority. The second reason follows from this. Since the debugger is neglected in the early stages of building a programming environment, its requirements can be ignored. When the task of writing the debugger is finally started, it is much harder than need be because source-object mapping information is not available from the compiler, and runtime debugging information is not readily accessible in the execution environment. Modifying an existing compiler and runtime system is a daunting task, as I have experienced. These practical problems apply equally to translation systems

for sequential programs. More research into debugger implementation techniques is not required, but rather education of compiler writers and others to needs of debugging.

9.3 Further work

I wish to identify two directions in which the research in this thesis could be developed.

Concurrent language design

One unexpected observation from Chapter 4 is how the design of concurrency features in the programming language directly affects whether it is possible to provide transparent halting. If the language enforces some restrictions on unsynchronized access to shared data, the task of writing the debugger is easier. However the real benefit of further work in this area is to make the programming process itself much easier and less error prone.

Multiprocessors

My debugging techniques are specifically designed for single processors connected by a local area network. Extending my work to tightly coupled, shared memory multiprocessors is important, especially since multiprocessors are gaining in popularity. Most of the debugging techniques described here should translate easily to a multiprocessor. The main difficulty will be the transparent halting mechanism of Chapter 5. This is because true parallelism is combined with shared access to memory.

A simple approach consists of one processor interrupting the other processors before they have time to perform their next memory access. Whether this is feasible depends on the interconnection architecture and the structure of the multiprocessor scheduler. However, I expect that many multiprocessors will not be able to satisfy this strict time constraint, so a few instructions are likely to be executed on each processor before they are all interrupted. During this small latency period it is possible, although unlikely, that process communication may take place through shared memory.

A more promising alternative is to categorize memory into private and shared regions, with a way to change the categorizations at run time. Access to the shared regions would be arbitrated using a lock protocol. By instrumenting the protocol we could halt processes before they perform their next process interaction via shared memory. Access to non-shared memory would be unaffected and should comprise the bulk of memory accesses, even in a shared memory multiprocessor.

9.4 Final word

In this thesis I have described how interactive source-level debugging of concurrent and distributed programs can be provided using low-cost implementation techniques. Using the ideas in this thesis, and with Pilgrim as a model, we can now implement debuggers for distributed programming. As a result I expect distributed programming to become easier and more commonplace.

Bibliography

Ada Programming Language. ANSI/MIL-STD-1815A, U.S. Dept. of Defence, January 1983.

E. Adams and S. S. Muchnick. "Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations." *Software—Practice and Experience*, **16**(7), July 1986, p. 653.

F. Baiardi, N. De Francesco, and G. Vaglini. "Development of a Debugger for a Concurrent Language." *IEEE Trans. on Software Engineering*, **SE-12**(4), April 1986, p. 547.

J. N. W. Bei and E. G. Manning. *CGDS: A Graphical Debugger for a Distributed System.* Technical Report 85-10, Institute for Computer Research, University of Waterloo, Ont., Canada, June 1985.

P. Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice Hall, Englewood Cliffs, N. J., 1977.

B. Bruegge. *Adaptability and Portability in Symbolic Debuggers.* Ph.D. Dissertation, Technical Report CMU-CS-85-174, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn., September 1985.

B. Bruegge and P. Hibbard. "Generalized Path Expressions: A High Level Debugging Mechanism." *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, August 1983, p. 34 (published as *Software Engineering Notes,* **4**(4) and SIGPLAN Notices, **8**(8)).

T. A. Cargill. "Implementation of the Blit Debugger." *Software—Practice and Experience*, **15**(2), February 1985, p. 153.

T. A. Cargill. "Pi: A Case Study in Object-Oriented Programming." *Proc. of OOPSLA '86*, Portland, Oregon, September 1986, p. 350 (published as *SIGPLAN Notices,* **21**(11), November 1986).

R. H. Carver and K. Tai. "Reproduceable Testing of Concurrent Programs Based on Shared Variables." *Proc. of the 6th International Conference on Distributed Computing Systems*, Cambridge, Mass., May 1986, IEEE Computer Society Press, Washington D.C., p. 428.

K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Trans. on Computer Systems*, **3**(1), February 1985, p. 63.

S. Y. Chiu. *Debugging Distributed Computations in a Nested Atomic Action System.* Ph.D. Dissertation, Technical Report MIT/LCS/TR-327, Laboratory for Computer Science, MIT, Cambridge, Mass., December 1984.

R. C. B. Cooper. "Pilgrim: A Debugger for Distributed Systems." *Proc. of the 7th International Conference on Distributed Computing Systems*, Berlin, September 1987, IEEE Computer Society Press, Washington D.C., p. 458.

R. C. B. Cooper and K. G. Hamilton. "Preserving Abstraction in Concurrent Programming." *IEEE Trans. on Software Engineering* (*to appear* February 1988).

D. H. Craft. *Resource Management in a Distributed Computing System.* Ph.D. Dissertation, Technical Report 77, Computer Laboratory, Cambridge University, March 1985.

R. Curtis and L. Wittie. "Bugnet: A debugging system for parallel programming environments." *Proc. of the 3rd International Conference on Distributed Computing systems*, Miami, Florida, October 1982, IEEE Computer Society Press, Silver Spring, Maryland, p. 394.

N. De Francesco, D. Latella, G. Vaglini. "An Interactive Debugger for a Concurrent Language." *Proc. of the 8th International Conference on Software Engineering*, London, August 1985, IEEE Computer Society Press, Washington D.C., p. 320.

A. Di Maio, S. Ceri, and S. Crespi Reghizzi. "Execution Monitoring and Debugging Tool for Ada Using Relational Algebra." *Ada in Use, Proc. of the Ada International Conference*, Paris, (*Ada Letters*, **5**(2)), September 1985, p. 109.

P. Fritzson. "Preliminary Experience from the DICE system, a Distributed Incremental Compiling Environment." *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, August 1983, p. 113 (published as *Software Engineering Notes,* **4**(4) and SIGPLAN Notices, **8**(8)).

J. Gait. "A Debugger for Concurrent Programs." *Software—Practice and Experience*, **15**(6), June 1985, p. 539.

H. Garcia Molina, F. Germano Jr., and W. H. Kohler. "Debugging a Distributed Computing System." *IEEE Trans. on Software Engineering*, **SE-10**(2), March 1984, p. 210.

N. H. Garnett. *Intelligent Network Interfaces.* Ph.D. Dissertation, Computer Laboratory, Cambridge University, 1983.

R. Gusella and S. Zatti. "TEMPO: A Network Time Controller for a Distributed Berkeley UNIX System." *Proc. of the USENIX Summer Conference,* June 1984.

K. G. Hamilton. *A Remote Procedure Call System.* Ph.D. Dissertation, Technical Report 70, Computer Laboratory, Cambridge University, December 1984.

P. K. Harter, D. M. Heimbigner and R. King. "IDD: An Interactive Distributed Debugger." *Proc. of the 5th International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985, IEEE Computer Society Press, Silver Spring, Maryland, p. 498.

J. Hennessy. "Symbolic Debugging of Optimized Code." *ACM Trans. on Programming Languages and Systems*, **4**(3), July 1982, p. 323.

D. R. Jefferson. "Virtual Time." *ACM Trans. on Programming Languages and Systems*, **7**(3), July 1985, p. 404.

M. S. Johnson. "A Software Debugging Glossary." *SIGPLAN Notices*, **17**(2), February 1982, p. 53.

P. R. Johnson. *Personal communication*, July 25, 1986.

L. Lamport. "Time, Clocks and the Ordering of Events in a Distributed System." *Comm. of the ACM*, **21**(7), July 1978, p. 546.

B. W. Lampson and D. D. Redell. "Experience with Processes and Monitors in Mesa." *Comm. of the ACM*, **23**(2), February 1980, p. 105.

H. E. Lauer and R. M. Needham. "On the Duality of Operating System Structures." *Proc. of the 2nd International Symposium on Operating Systems, Theory and Practice.* IRIA, Rocquencourt, France, October 1978. (Reprinted in *Operating Systems Review*, **13**(2), April 1979, p. 3.)

T. J. LeBlanc and M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay." *IEEE Trans. on Computers*, **C-36**(4), April 1987, p. 471.

C. H. LeDoux and D. S. Parker Jr. "Saving Traces for Ada Debugging (YODA)." *Ada in Use, Proc. of the Ada International Conference*, Paris, (*Ada Letters*, **5**(2)), September 1985, p. 97.

B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Manual. Lecture Notes in Computer Science,* **114**, Springer Verlag, Berlin, 1981.

B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, Mass., 1986.

B. Liskov and M. Herlihy. "Issues in Process and Communication Structure for Distributed Programs." *Proc. of the 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems,* October 1983, p. 123.

C. Mauger and K. Pammet. "An Event-Driven Debugger for Ada." *Ada in Use, Proc. of the Ada International Conference*, Paris, (*Ada Letters*, **5**(2)), September 1985, p. 124.

B. P. Miller and J. Choi. *Breakpoints and Halting in Distributed Programs.* Technical Report 648, Computer Sciences Dept., University of Wisconsin-Madison, Wisc., July 1986.

P. V. Mockapetris. "Analysis of Reliable Multicast Algorithms for Local Networks." *Proc. Eighth Data Communications Symposium*, ACM, October 1983, p. 150.

S. J. Mullender and A. S. Tanenbaum. *The Design of a Capability-Based Distributed Operating System.* Technical Report CS-R8418, Dept. of Computer Science, Centrum voor Wiskunde en Informatica, Amsterdam, October 1985.

R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System.* Addison-Wesley, London, 1982.

R. M. Needham and M. D. Schroeder. "Using Encryption for Authentication in Large Networks of Computers." *Comm. of the ACM*, **21**(12), December 1978, p. 993.

B. J. Nelson. *Remote Procedure Call.* Ph.D. Dissertation, Technical Report CMU-CS-81-119, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn., May 1981.

D. Otway and O. Rees. "Efficient and Timely Mutual Authentication." *Operating Systems Review*, **21**(1), January 1987, p. 8.

M. Richards, A. R. Aylward, P. Bond, R. D. Evans, and B. J. Knight. "TRIPOS—A Portable Operating System for Mini-computers." *Software—Practice and Experience*, **9**(7), July 1985, p. 513.

P. Rovner, R. Levin, J. Wick. *On Extending Modula-2 For Building Large, Integrated Systems.* Technical Report 3, DEC Systems Research Center, Palo Alto, Calif., January 1985.

R. D. Schiffenbauer. *Interactive Debugging in a Distributed Computational Environment.* M.S. Dissertation, Technical Report MIT/LCS/TR-264, Laboratory for Computer Science, MIT, Cambridge, Mass., September 1981.

E. T. Smith. "A debugger for message-based processes." *Software—Practice and Experience*, **15**(11), November 1985, p. 1073.

W. Teitelman. "A Tour Through Cedar." *Proc. of the 7th International Conference on Software Engineering*, Orlando, Florida, March 1984, IEEE Computer Society Press, p. 181.

M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System.* North Holland, New York, 1979.

L. Wittie and R. Curtis. "Time Management for Debugging Distributed Systems." *Proc. of the 5th International Conference on Distributed Computing Systems*, Denver, Colorado, May 1985, IEEE Computer Society Press, Silver Spring, Maryland, p. 549.

P. T. Zellweger. "An Interactive High-Level Debugger for Control-Flow Optimized Programs." *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, August 1983, p. 159 (published as *Software Engineering Notes,* **4**(4) and SIGPLAN Notices, **8**(8)).

P. T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs.* Ph.D. Dissertation, Technical Report CSL-84-5, Xerox PARC, Palo Alto, Calif., May 1984.