A Debugging Support Based on Breakpoints for Distributed Programs Running Under Mach

Christy Yep

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

November 1992

ISBN 0-315-84635-6

ABSTRACT

A Debugging Support Based on Breakpoints for
Distributed Programs Running Under Mach

Christy Yep

Debugging a distributed program is a non-trivial task because of the inherent non-determinism and the real global time being absent in distributed systems. Also there are multiple threads of computation. As a result, specification of breakpoints, monitoring for their detection and halting, stepping the execution in the code space, and the assimilation of the vast amount of trace information all become difficult. Research in distributed systems is actively pursued in the last ten years by several active research groups. Experimental debuggers have been built to support various types of research in this field.

This thesis is a part of the whole project called CDB, which involves a team of graduate students whose collective aim is to design and implement a distributed debugger. Unlike many other implementations of distributed debuggers, CDB is an "integrated system". It integrates the concepts of non-deterministic replay, distributed breakpoints, rollback and recovery, white-box to black-box approach, and the user interface to support interactive debugging. Several theses have been written and the work is still on going, with regards to CDB.

Every concept discussed in this thesis is implemented under Mach, tested, and documented. Thus one of the contributions of this thesis is the implementation of ideas for conducting experimental research. Based on the research reported in literature, a specification language called PDL (Predicate Definition Language) is proposed in this thesis. Distributed breakpoints specified using PDL can be detected by the present implementation and the distributed computation can be halted. PDL assists the user to specify breakpoints at different levels of granularity, thus facilitating white-box or black-box type inspection of the computation. We have implemented two subsystems of CDB: breakpoint and user interface. They are designed such that future breakpoint and user interface methods can be incorporated. In developing these two subsystems, we have identified building blocks which can be re used in building other CDB subsystems. The two building blocks, vector clock server and local debugger are designed in such a way that their re-usability is maximized

Dedication

This thesis is dedicated

to my parents

Albert

and

Judy

Acknowledgement

First and foremost I would like to thank my supervisor, Dr. Radhakrishnan, who guided me through every step during the master's program. Without his help and patience, the work presented in this thesis would be of less value. I would also like to take this opportunity to share my experience in having Dr Radhakrishnan as my supervisor. Dr. Radhakrishnan was always available whenever I needed help on both academic and social levels. His knowledge in computer science and his methods of teaching inspired me where these two years seem to pass very quickly. Aside from computer science, Dr. Radhakrishnan taught me a lot about life and culture and gave me the opportunity to see life outside of North America. This is probably the most valuable learning experience he has ever given me Again I would like to thank Dr. Radhakrishnan for his supervision and for being a great person.

I would also like to thank Bell Northern Research for their financial assistance.

I would like to express my thanks to Alain Sarraf who is also part of the distributed debugging project. Alain and I had many brainstorming sessions, both theoretical and practical, that were beneficial for both of us. Also, his moral support and amicable personality made the project more enjoyable.

I would also like to give a special thanks to Juliette D'Almeida who was always there whenever times became difficult. I deeply appreciate her care, warmth, and moral support during both my bachelor and master degree.

Other very special people in my life also supported and encouraged me during the past two years. I'd like to thank Craig Mathewson, George Papaleonidopoulos, Mitsuo Matsushita, Le Huan Tran, and Edward Battershill for their irreplaceable friendship.

Last but not least, I'd like to express my deepest gratitude to Mom, Dad, YinLei, Tony, Dolly, Andy, and Sally for their encouragement, love, and patience.

Contents

List of Figures

Chapter 1

Introduction

The interest in parallel and distributed programming has grown dramatically in recent years. The added complexity of expressing concurrency has made debugging parallel and distributed programs even harder than debugging sequential programs

One of the most important facilities in any debugging system is the ability to halt a program during its execution. Once halted, the program state may be inspected to different levels of detail in order to find the error. This facility of halting is called a breakpoint. In sequential programs, specifying a breakpoint is straightforward since all program events are linearly ordered. A breakpoint would specify any one of those events where the program is expected to halt. Distributed programs, on the other hand, pose new problems for breakpointing. Since the nature of the program consists of many threads of control and determining the order of program events is difficult, specifying a distributed breakpoint becomes non-trivial. Detection of the occurrence and halting at a breakpoint are to be studied carefully since many threads of control are present. Certain protocols must be developed in order to make detection and halting meaningful to the user. Another factor in debugging is to provide an effective user interface. With the large number of processes being present, a user could be overwhelmed with unmanageable amounts of information. New methods of displaying information must be : ldressed.

This thesis is a design and implementation of solutions to the problems stated above. The implementation is within the context of CDB, Concordia Distributed deBugger, where the main objectives are to provide a debugger that will make use of the user's knowledge of the program, provide a set of flexible tools, and provide a user interface that will enable the user to effectively use these tools [LRK90].

1.1 Terminologies

Before introducing the issues of debugging, some of the key terminologies are explained below:

1.1.1 Sequential and Distributed Programs

A *sequential program* is a set of program statements or events that occur in sequential order, all within the same machine and address space. A *distributed program*, on the other hand, can be seen as a set of sequential programs dispersed among several sites executing simultaneously, cooperating, and communicating to achieve a common task. "Executing simultaneously" would refer to processes running in true parallel on separate sites or in pseudo-parallel on the same site. "Cooperation" refers to all processes coordinating their actions so that they do not nullify or contend with each other. "Communication" could be based on shared memory or message passing.

The IPC model considered in this thesis is *message passing*. Processes communicate with each other by sending messages to each other. The IPC primitives are made up of a non-blocking send, a blocking receive, and remote procedure calls (RPC). Messages are sent to *ports* which are essentially message queues. Messages received are taken from these ports.

1.1.2 Bug, Testing, and Debugging

Any experienced programmer has and will encounter a bug in his/her program. A *bug* is any deviation in the actual program behavior from the expected behavior. Finding a bug starts with *testing*. Testing is the action of exercising all possible paths a program may take during execution. It is this phase that will reveal if a program contains a bug. When a bug is known to exist, *debugging* is performed. Debugging is the action of pinpointing the exact location of the bug. This involves using different debugging techniques and tools which will be explained in the following sections.

1.2 Conventional Debugging Techniques

Essentially, there are two methods of debugging a program. They are known as *top down* and *bottom up*.

Top down approach requires that all characteristics of the program be well defined where a theoretical behavioral pattern may be developed. During debugging, the actual program behavior is compared with this theoretical pattern. A bug will is said to occur when one of the two behavioral patterns deviates from the other. The problem with this method is that the user must know all details about the behavior of the program. For a large software system, this may be too voluminous. Also, knowing exact behavioral characteristics of a program may be difficult to grasp. With this in mind, the bottom up method may be more advantageous.

The bottom up approach is more of the classical way of debugging as we commonly practice. It assumes that the user is not quite aware of all the details of the behavioral characteristics of the program. The user is expected to know that a bug exists and there is a hunch of where the bug could be. With this information, the user would execute the program to the anticipated point of the bug and perform a step by step analysis. In doing this, the user will gain more knowledge about the program and thus determine where the bug is. To make this method effective, *cyclic debugging* techniques is usually applied. Once the step by step analysis is done, a new anticipated point of the bug will be determined. Hence, the bottom up method is applied again. With every cycle, the user gains more knowledge about the program and spirals into the location of the bug.

1.3 Debugging Tools

Both methods described above require some tools. This section describes the main tools found in sequential debuggers.

Trace

The trace tool supports the top down method of debugging. Program events are gathered and stored in a file or displayed on the screen during the execution of the program. Usually the type of events may be predefined by the user such that only those events of interest will get recorded.

Breakpoint

The breakpoint tool is one of the tools that support the bottom up method. A breakpoint is a location, within the program, of where the program will be suspended. Essentially, it is a point in global program time.

Stepping

Stepping is a tool used in collaboration with breakpoints. Once a breakpoint is reached, the stepping tool may be used to incrementally advance the program to support detailed inspection.

State Recording

The state recording tool is simply a tool that will return the current state of the program. For example, state information may be gathered at a breakpoint as well as after each step of the execution.

User's Intuition

User's intuition is very useful in debugging. No matter what method of debugging is used, there is no guarantee that the bug will be found. Basically, debugging is an art because more than half the job of debugging relies on the user's intuition and his/her ability to understand the faulty program. A debugger is simply an extension to the user's intuition.

1.4 Problems in Distributed Debugging

The above tools are also needed in distributed debugging in order to apply the debugging methods. But, due to the nature of a distributed program, various problems arise which make these tools difficult to obtain. It also requires that other tools be created to deal with these problems. The following sections will outline the problems in distributed debugging.

Lack of Global Time

The concept of a global state or time becomes misleading or even non existent without some kind of synchronized global clock. Since each process within a distributed program conforms to its own local clock, the global clock would consist of the synchronization of these clocks. But, this synchronization is difficult to achieve since it would rely on an unpredictable communication network. However, without global time it would be difficult to determine the order of program events occurring on distinct parallel processors.

Non-Determinism

Since distributed programs rely on communication, unpredictable delays in the network will make the programs non-deterministic. Every run of the program may experience different message delays which may cause a different computational paths to be taken.

Breakpoints and Stepping

Basically, a breakpoint specification implies a reference in time. This is no problem a sequential program since it spans within a single process on a single processor. On the other hand, global time in a distributed program does not exist, which makes the specification of breakpoints difficult. Also, we need some way to specify breakpoints that do not overwhelm the user with too much detail but is effective.

Stepping experiences similar problems. A step in sequential programs refers to moving the execution to the next reference in local time. Stepping in distributed programs refers to advancing the execution of all processes to the next reference in global time. Since global time is difficult to achieve, stepping becomes a non-trivial task.

Probe Effect

Since the distributed debugger itself is another program that executes within the same system as the application program, the debugger will induce an extra load on the system. This extra load could cause more message delays and affect certain race conditions.

User Interface

Since a distributed program is made up of several tasks, there is a potential scaling up problem. For each task, debugging information must be displayed to the user where the size of the physical monitor and the user's ability to comprehend become the limitation. What is needed is a means of displaying this data in a meaningful manner within the limitations of the screen and human cognition.

1.5 Related Work

Distributed debugging has been researched for at least the past ten years where several experimental systems have been developed in both theory and practice. The following work concentrates on "complete debugging systems" that influenced the design of Concordia's Distributed deBugger, CDB. CDB is a set of debugging tools designed to debug distributed programs. Section 1.6 discusses CDB in more detail.

Garcia-Molina, Germano, and Kohler [GGK84] explains the key problems in distributed debugging and propose a bottom up approach methodology for debugging. Each single process is tested individually, then tested all together as a whole. Their system focuses mainly on a trace tool where events are logged during the execution of the program. The testing phase will let the user compare the expected behavior against the actual behavior. The disadvantage is that a user can not always know the expected behavior. Hence, testing against expected behavior becomes difficult

Miller and Choi [Miller-Choi88-1] propose a mechanism for debugging parallel programs where it incorporates replaying the system without re-execution. Detailed trace files are taken during one run of the program where they are used to recreate the distributed computation. Hence, a slow pace view of the system behavior can be inspected by the user. This method puts a high dependency on the trace files since it determines the replay detail in the recreation of the program.

Leblanc and Mellor-Crummey [Leblanc-Crummey87] state that the most difficult problem in distributed debugging is reproducing the execution behavior. They go on to describe a method of replay called Instant Replay where sufficient trace information is taken from the processes to ensure that subsequent program executions are the same. This method permits traditional methods of debugging (breakpoints, stepping, etc.) to be inserted since the execution may be slowed down without affecting the behavior.

Bates and Wileden [Bates-Wileden83] take a high level view of distributed debugging where they propose EDL. EDL, Event Definition Language, is a means of stating what events are of interest to the user during the computation. With this definition, monitoring is performed where the user is expected to compare actual behavior with the expected behavior. This work has been influential to our contribution to CDB

Baiardi, De Franchesco, and Vaglini [BDV86] describe a debugger for the concurrent language called ECSP. Like Bates and Wileden, they take a high level view where a behavioral specification of the system is supplied by the user and it is compared against the actual behavior during execution. Their behavioral specification language, called BS, provides a facility where a user may specify the program's behavior at different levels of abstraction. Depending on the user's knowledge of the program, a BS specification may be created corresponding to the user's view of the program. The BS specification may range from a coarse to a fine grain specification.

Joyce, Lomow, Slind, and Unger [JLSU87] propose a debugging system that con centrates on providing the user with a means to compose any program behavior. This form of debugging is more of a testing phase since it exercises all requested execution paths. The disadvantage of this system is that it is only suited for users who have a broad knowledge of the program. During the early stages of debugging where the user has limited knowledge of the program, the user may not know which execution path the bug may occur. Also, determining all possible execution paths is non-trivial. Another disadvantage is that there is a lack in assisting the user to gain more knowledge about the program such that their debugging system may be useful

Each of th above works is focused on one or more "basic" concepts of distributed debugging, namely, trace, record and replay, event specification, fine grain to coarse grain variability, and behavioral abstraction (top down). CDB, on the other hand, has taken an integrated approach to these concepts. A complete implementation under Mach is equally emphasized in CDB.

1.6 CDB - The Project

Before introducing CDB, the Mach operating system under which it executes will be described briefly.

1.6.1 Mach

Mach [MACH-1] [MACH-2] is a distributed operating system which provides an integrated coi puting environment that consists of networks of multiprocessors and uniprocessors. Mach's goal is to design a Unix compatible system as well as to add mechanisms that other distributed systems do not have. Major mechanisms involve:

- Support for multiprocessor architectures.
- A generic micro-kernel architecture where other operating systems may be built on top of.
- Distributed operation where the network is transparent to the user.
- Efficient memory management and interprocess communication modules.
- Organization according to the object oriented paradigm.

Mach provides an environment made up of simple basic abstractions. One of such abstractions is its process model of *tasks* and *threads*. A task is the address space and a collection of system resources. A thread is a basic unit of computation and can only run within the context of one task. A task may have many threads running, in parallel, within it. Another abstraction is the IPC model. Essentially, it is a message passing model made up of *messages* and *ports*. Messages are actual data in transit where ports are message destination queues. With this model tasks and threads may easily communicate transparently through the network, with a single construct.

In short, this operating system provides many facilities which directly support distributed computing environments. Also, it provides well defined abstractions and simple programming tools which make these facilities easily accessible. For these reasons Mach has been chosen as the operating environment for CDB.

1.6.2 The Project

Solutions for the problems of lack of global time, non-determinism, breakpoints and stepping, probe effect, and user interface are presented in the design of Concordia's Distributed deBugger, CDB [LRK90], project. CDB, is a project suited for the mach programming environment and it is designed to debug distributed programs written in C or C++. Essentially, CDB is made up of a set of debugging tools which are monitoring based tools, replay related tools, breakpoint related tools, display and interaction oriented tools, and blackbox and whitebox facilities.

- Monitoring based tools basically gather important information during the execution of the program.
- The replay related tools provide a means to re-execute the program from any point while ensuring that every run will exhibit the same behavior.
- Breakpoint related tools enable one to specify a breakpoint to halt from where step by step analysis may be performed.
- The display tools create a meaningful user interface.
- The blackbox and whitebox facilities are embedded in the other tools such as monitoring and breakpointing. Essentially, it is a facility which lets the user specify the granularity, fine to coarse grain, view of the system. For example,

monitoring the system with respect to the synchronization space would be a blackbox or coarse grain view since finer details are ignored. A finer grain or whitebox view would be a view with respect to the source code space.

1.6.3 Related Work (Concordia)

CDB is an on going project that started in 1988 and is supported by Bell Northern Research at Ottawa. Prior to CDB, other related work had be done in distributed systems that were influential to CDB. They are:

Ph.D. Thesis 1988: Krishnarao Venkatesh proposed a formalism for classifying different types of global states of a distributed system as well as developing new message efficient algorithms for detecting consistent and stable global states. In relation to this, he examined certain algorithms that depend on global states. More specifically he examined in detail two applications, discrete event simulation and backward error recovery, where he identified and proposed new solutions to the inherent problems within such algorithms.

Master Thesis 1988: Chris Passier provided experimental evaluations of two rollback and recovery (R&R) algorithms. He had implemented a R&R kernel which serves as a basis for building the R&R algorithms. The bulk of his work focussed on more theoretical issues of the algorithms.

Master Thesis 1988: Ioakim Hamamtzoglou also contributed to the implementation of the R&R kernel with Chris Passier. His work describes the problems related to distributed debugging and proposes solutions that are in the framework of distributed computations based on partial ordering. He also describes the usage of debugging tools and presents a model of a distributed debugger.

Master Thesis 1989: Minh Dang Bao proposed a methodology and tools for distributed debugging. His methodology consists of a two stage process. The first stage is a top down approach where a given synchronization specification is compared with the actual synchronization behavior until the error is located.

The second stage is a bottom up approach where examination of internal states is performed to locate the bug. He had also implemented a prototype debugger on a set of Unix based machined to exercise his methodology.

The implementations provided by Chris Passier, Ioakim Hamamtzoglou, and Minh Dang Bao were "throw away prototypes". Further extensions to these systems were not feasible. But their work and the work of Krishnarao Venkatesh provided a good base for the evolution of CDB. The following are direct contributions for the CDB project.

Master Thesis 1992: Victor Krawczuk proposed and implemented a record and replay module where it addressed the problem of non-determinism in a distributed program. Essentially, all non-deterministic choices are recorded during one run of the program where all subsequent replay will conform to the recorded information. Only partial implementation is achieved.

Master Thesis 1992: Honna Segel based her research in the area of monitoring. She developed a new logic for expressing safety properties in distributed predicates. Safety properties refer to predicates being stable or unstable throughout the process execution. She also proposed algorithms for detecting these safety properties so that they can be monitored efficiently.

Master Thesis: Alain Sarraf is currently working on a checkpoint and rollback mechanism. Once an error has been detected, his system will rollback the execution to a state or checkpoint that is considered error free. From the rolled back state, interactive debugging may be performed to locate the bug.

1.7 Contribution of this Thesis to CDB

The contributions of this thesis lie in the specification and detection of distributed breakpoints. An extendible user interface is also developed. All the proposed facilities are fully implemented and documented.

We propose a specification language called PDL (Predicate Definition Language) which enables the creation of hierarchical breakpoints ranging from a very fine to a very coarse granularity. We also implemented a breakpoint module which detects a PDL specification and halts the system upon its detection. The user interface is another problem that we addressed. Here, we implemented a user interface module that adopts the Space-Time diagram method of capturing meaningful data during the execution of the program. In order to make the breakpoint and user interface modules possible, two underlying generic building blocks were implemented. One building block is a vector clock module which provides a solution to the lack of global time. Here, we adopt the vector clock algorithm where temporal relations between events may be determined. The other building block is a local debugger module. Essentially, this module has total control of the execution of a single process to which it is assigned. The local debugger can be controlled from any site where it provides hooks to the application program.

Chapter 2

System Architecture of CDB

Distributed debugging systems vary in terms of their design. Some are designed such that their modules are highly dispersed among many sites and some are more centralized. The following chapter describes the architectural design of CDB which takes the latter approach.

2.1 CDB's Architecture

Essentially, CDB is made up of several interacting modules and "servers" which are organized in a centralized manner. The philosophy behind this architecture derives from the setting in which we expect the user to work in. We assume that the user doing the debugging is situated at only *one* site. This site, we call the *debugging site*, plays a very important role during debugging since it is expected to contain all debugging information. For this reason, we have organized CDB to be centralized at the debugging site. All non-debugging sites simply contain generic servers that control and extract information from local processes on that site. These generic servers have no dependency with any other modules and can be considered as building blocks. Modules of CDB depend on these servers in order to obtain control and information of application processes. Figure 2.1 illustrates the system architecture of CDB.

The modules on the debugging site form the actual debugger and consist of break point, stepping, monitor, record and replay, checkpoint and rollback, and user interface modules. The servers consist of a vector clock module and a local debugger

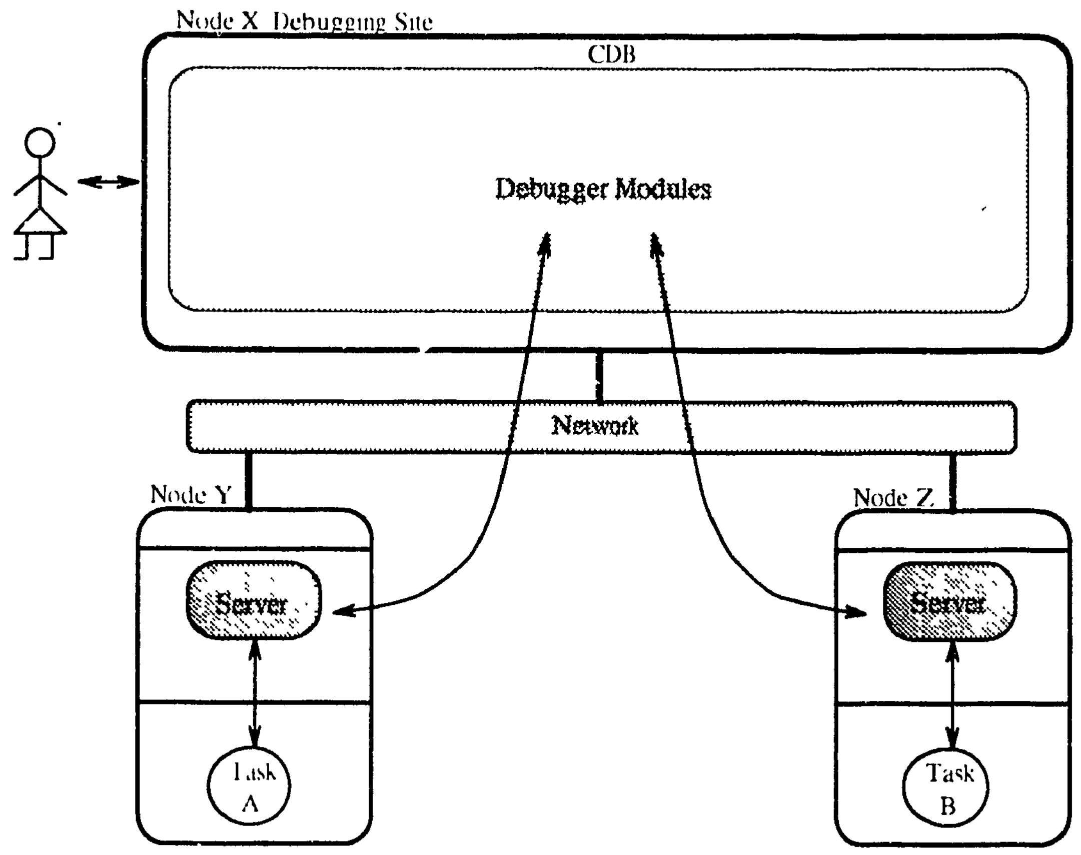


Figure 2.1: System Architecture of CDB

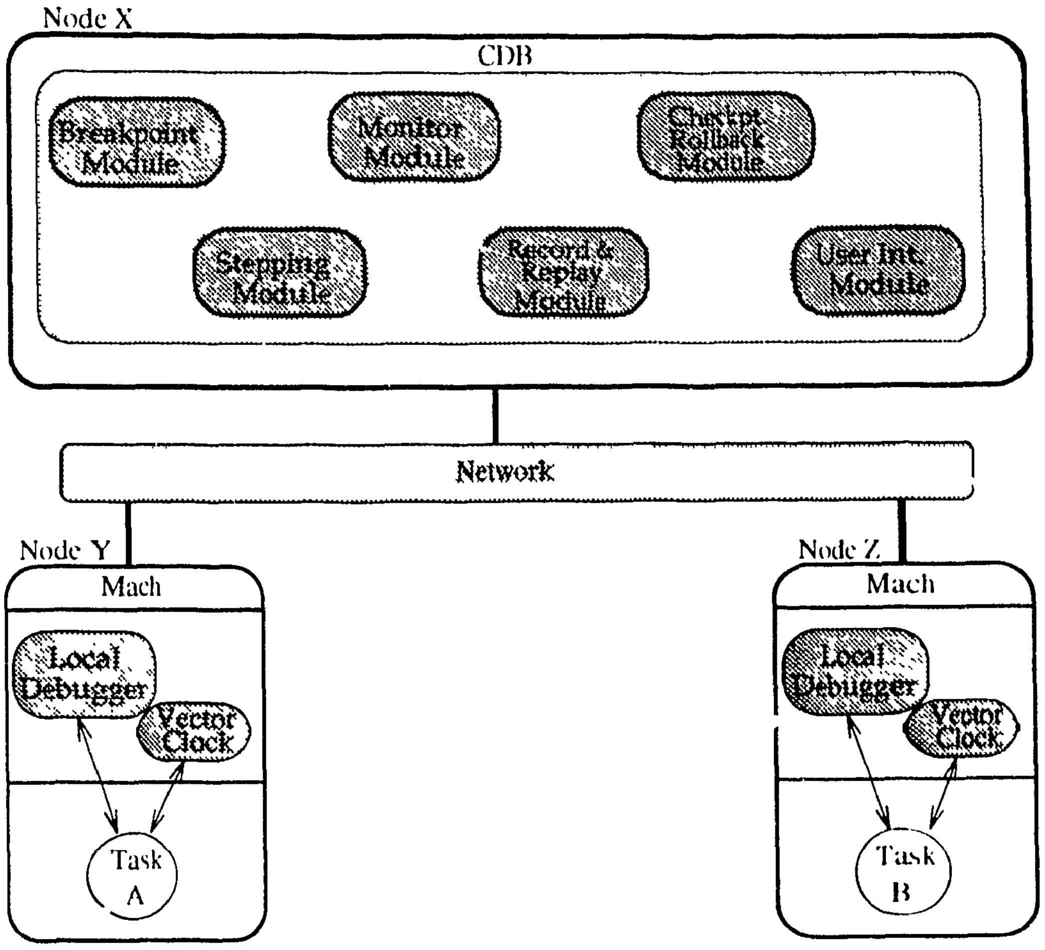


Figure 2.2: Modules of CDB

module. Figure 2.2 shows the placement of all the modules within CDB. These modules are described in more detail in the following sections.

2.2 USER-TO-CDB Session

In order to comprehend the facilities that each module provides, a general user to CDB debugging session is described.

Before doing any debugging, the user is required to compile his/her program in the debugging mode while linking in a special CDB library. This will prepare the program for the debugging session.

Because of the inherent problem of non-determinism in distributed programs, the debugging session can be viewed as two separate stages. The first stage involves recording the program execution and the second involves replaying.

During the recording stage the user is required to run his/her program until termination. Here, CDB will record and store all non-deterministic events in its own local database. This record facility will ensure a deterministic replay.

Having recorded all events, the user proceeds to the replay stage where the actual interactive debugging is performed. Here the program is executed and will replay exactly the same sequence as recorded. CDB provides many debugging tools similar to traditional debuggers and supports both top-down and bottom-up debugging techniques. The tools include monitoring, breakpoint, stepping, checkpoint, and rollback.

Monitoring involves specifying certain events to be monitored. Here, the user is required to specify the event, via Predicate Definition Language or PDL. CDB will detect such events and notify, the user, of its occurrence. The notification can either be in an interactive or file form. This facility supports the top-down debugging technique where the user may compare the monitor's trace against the expected behavior of the program.

Closely related to monitoring is the breakpoint tool. Again using Predicate Definition Language, the user may specify at the occurrence of what event(s) the program should halt (as opposed to what events should be monitored). Once the program is halted, the user may investigate more detailed aspects of the program. This facility, as well as the ones that follows, support the bottom-up technique of debugging.

In both monitoring and breakpoint, Predicate Definition Language, is used to specify events to be detected by CDB. PDL allows the user to construct hierarchical events from a very coarse grain to a very fine grain specification. Here, the user is expected to have some knowledge of PDL in order to interact with the monitoring and breakpoint facility.

Stepping is a tool that is used in collaboration with breakpoints. Once a breakpoint is reached, the user may incrementally advance the program's execution and examine the state of the program after each step (or group of steps).

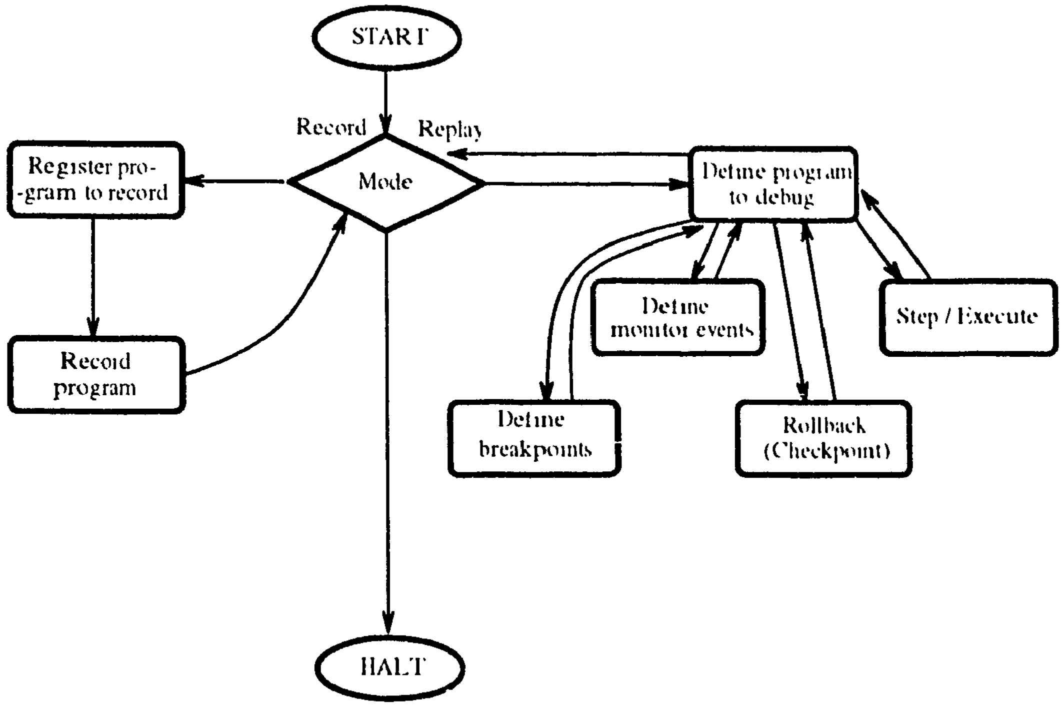


Figure 2.3: User Session with CDB

Checkpointing and rollback are two tools that work together to provide a means of bringing the execution of the program back to a previous state and to re-execute from that point. Checkpointing is the action of recording enough state information of a program so that re-execution starting at that point is possible. Checkpointing is transparent to the user and CDB, at this stage, arbitrarily chooses points where a checkpoint is taken. With these checkpoints, the user may use the rollback facility to re-execute the program from any one of the checkpoints. Also, when finding an error in the system, the user may ask CDB to find a checkpoint prior to the error.

To illustrate the user's perspective, figure 2.3 shows a flow diagram of a session with CDB.

2.3 Modules of CDB

The basis for CDB consists of two generic building blocks, organized as servers, vector clock server and the local debugger. These servers support the higher level modules

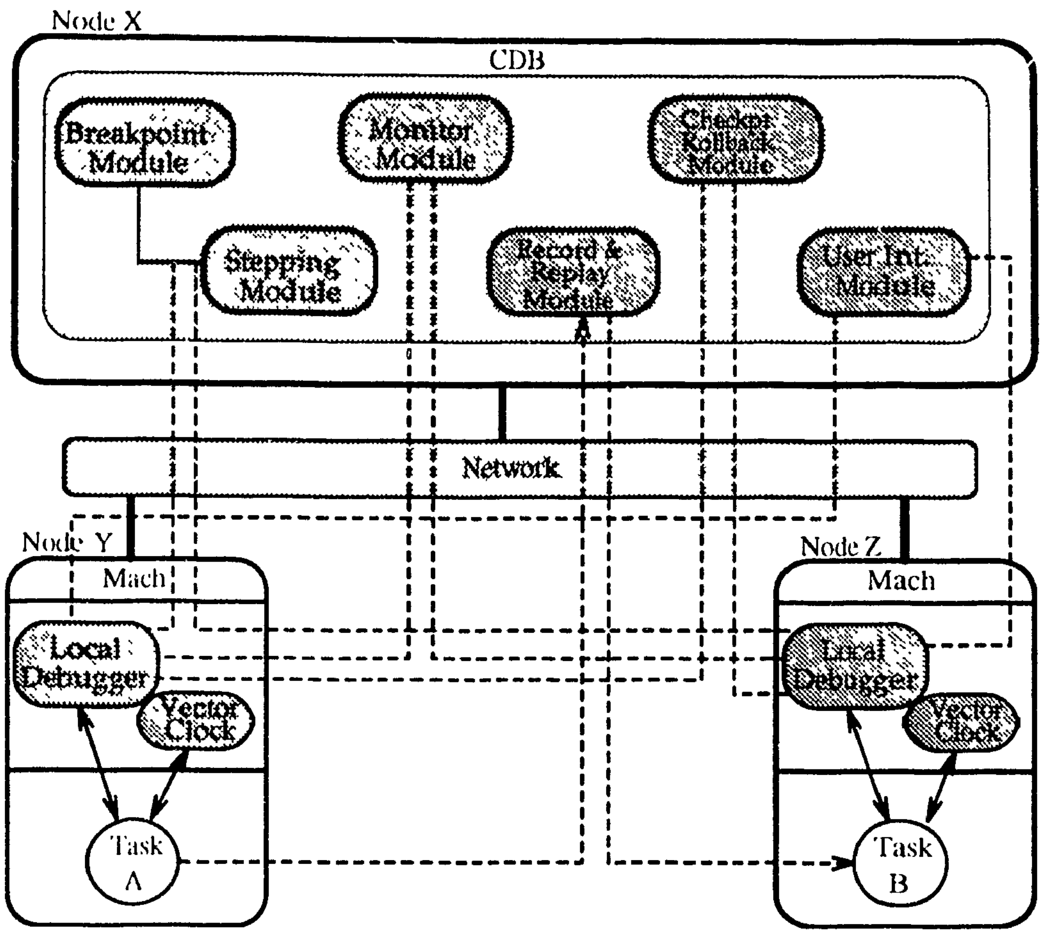


Figure 2.4: Interactions of Modules of CDB

of user interface, breakpoint, monitoring, checkpoint and rollback, and stepping. We intentionally left out the user interface module from this section because it deserves to be mentioned in a section of its own.

The following sections are all with reference to figure 2.4 which shows the interactions between all modules of CDB.

2.3.1 Vector Clock

As mentioned earlier, one of the problems of distributed debugging is the lack of global time. In order to obtain true global time we need a mechanism to capture the exact times when each and every event occurred relative to each other. This will give us the total order of the system. Unfortunately total ordering is impossible since the modules of a distributed program are dispersed and the concept of time is different at each site. However, providing a mechanism that can determine the partial order of events is useful and possible The term partial ordering refers to being able to determine the temporal relationships between some of the events (as opposed to all the events in total ordering).

This will be discussed in more detail in section 3.1

Vector Clock with CDB

The vector clock module provides an interface to CDB, to determine the partial order between events. It functions entirely transparent to the user, and consists of one server per machine. The user's program is augmented, at compile time, such that special vector clock commands are inserted. These commands interact with the servers.

2.3.2 Local Debugger

A distributed program is made up of several sequential tasks communicating with each other. For each sequential task a local debugger is attached to control that program's execution and to obtain local state information. The local debuggers are very important since they can be used to control the total execution of the distributed program.

This will be further discussed in more detail in section 3.2

Local Debugger with CDB

The local debugger module provides an interface where CDB has total control over the functionality of a sequential task. This module functions totally transparent to the user where for every sequential task within the distributed program, a local debugger module is created and attached to it. CDB will control all such modules and thus will have full control of the distributed program.

2.3.3 Record and Replay

One aspect in debugging sequential programs is that for a given a fixed input, the execution path will always be the same for every run. Distributed programs, on the other hand, do not exhibit this characteristic of determinism. A major cause for non-determinism is the IPC model as explained earlier. Messages passing through the communication medium may experience certain delays due to various loads on the system. These delays are non-deterministic which imply that any receiver of a message could wait indefinitely. A simple example of this non-determinism is where a process is expecting a messages from several other processes. With every run, the receiving process may receive the messages in different orders because of different communication delays. Hence, a different execution path may be taken for every run. This makes locating a bug difficult if it occurs in only one of the many paths, and that path is not followed during debugging.

What is needed here is a way to ensure that for every run, the same execution path will always be taken. The solution, adopted by the record and replay module, is to record all non-deterministic events during one run of the program. All subsequent runs will conform to the recorded information to ensure that the same execution path is taken during every debug run.

What is Recorded ?

Messages are recorded. Essentially, for every process, all message contents received are logged in sequential order. Though this method seems sensible, the overhead due to copying the message contents to its respective log must be considered. This overhead may cause a serious probe effect that may alter the execution path since extra debugging statements are included in the program. Also, there may be a potential overload in logging the messages, since they may tend to be very large. Another alternative is to tag every out going message with a unique "id" and to record only these id's within the log. This method will have less probe effect since there is less information to log and the log will be relatively smaller.

During re-execution of the program, every message received will be checked with the respective logs. If the message received does not match the message it should receive, according to the logs, then that message is put in storage until its turn arrives. This ensures that messages received are in the same order that they were recorded.

Record and Replay with CDB

The record and replay, R&R, module functions transparent to the user. During the record stage all out going messages are re-routed to the R&R module. These messages are then logged and re-sent to the actual destination. At the time of replay, all outgoing messages are re-routed to the R&R module where they are compared with the corresponding logs. This will ensure that incoming messages will be in the correct order.

2.3.4 Checkpoint and Rollback

Checkpoint and rollback are two facilities that support the *cyclical* approach to debugging. The checkpoint facility records enough state information of a distributed program so that the rollback facility may restart the execution from that point. Hence, certain critical areas of a program may be consistently re executed while under the observation of the user.

One characteristic of a checkpoint and rollback unit is to automatically rollback the execution to an *error free* checkpoint after detecting a bug. This is a non trivial task since it involves determining which is the correct checkpoint prior to the bug.

Checkpoint and Rollback with CDB

The checkpoint portion of the module is transparent to the user where checkpoints are issued by CDB. In taking checkpoints, the module instructs the local debugger module to return all data pertaining to the current status of the task where it is stored for future use. The rollback section provides the facility of either selectively rolling back to a checkpoint or to have the module locate a correct checkpoint prior to the bug. Essentially during rollback, the module supplies the local debugger module with the checkpoint data where the the task is re-fitted with a new execution environment. The local debugger will then re-attach to this new space.

2.3.5 Breakpoints

Essentially, a breakpoint represents a particular point during a program's execution. This is straight forward for sequential systems since programs consist of a single process on a single processor. But, since distributed systems involve multiple processes and multiple sites, breakpoints become unobvious. Breakpoint specification and detection must incorporate all processes where a "new notion" of a breakpoint in the distributed program must be addressed. We developed a Predicate Definition Language, PDL, as a means to defining events occurring on several tasks with temporal or non-temporal relationships so that a breakpoint can be stated. With this, the user may construct hierarchical breakpoint specifications ranging from very fine to very coarse granularity.

This will all be discussed in further detail in chapter 4.

Breakpoints with CDB

The breakpoint module requires that the user input a PDL specification. With this input, the breakpoint module will spawn jobs for the local debuggers for setting and detecting local breakpoints. Upon the occurrence of these events, the local debuggers will notify the breakpoint module and send to it the corresponding data of the event. With this information, the breakpoint module will determine if the breakpoint has been reached.

2.3.6 Stepping

Stepping is used in collaboration with breakpoints. Once a breakpoint is found and the system is halted, the stepping facility may be used to incrementally advance the program for detailed human inspection. This facility is straight forward in a sequential environments since a single step is local to one process only. But distributed systems pose new issues that must be addressed. One such issue is the granularity of each step With a large number of processes, stepping each process in the sequential conventional way will prove to have too much data for anyone to keep track of. Also, since stepping includes many processes, we need to decide what protocol should the processes follow, if only a subset of all processes are required to advance. Two protocols, causal and maximal, are explained in the following sections.

Fine to Coarse Grain Stepping

Stepping as we are familiar with involves advancing the program to the next line of execution. This would be considered as a fine grain step because it is the smallest possible program advancement that could be issued. But, this granularity may not be of any use if the distributed program is very large and performs many tedious operations. Perhaps a higher level of abstraction must be viewed in order to comprehend such a program. For example, one may take a view of the system with respect to only the IPC operations. This would be considered as a coarse grain view of the system since many of the finer details are ignored. The corresponding coarse grain step would involve advancing a process to the next IPC operation. We can even take a coarser grain view by considering only outgoing messages of the system. Similarly a step would advance the process to the next message output operation As we can see there are different levels of abstraction that a user may view his/her program in order to gain a better understanding. The stepping facility must also support these views in order to assist the user by providing stepping from a very fine to a very coarse level of granularity.

Causal and Maximal Stepping

Stepping in a sequential program, regardless of granularity, simply involves advancing one process to the next event. Distributed programs pose a new problem of knowing which processes should advance and which should not. For example, consider a distributed program made up of processes P1, P2, and P3. If P1 has been chosen to step to the next event, what protocol should P2 and P3 follow? Two protocols are explained below and are called causal stepping and maximal stepping [H.F. Li.].

Causal stepping functions is as follows : P2 and P3 are advanced to the earliest event possible which will permit P1 to reach its next event. In other words, if P1's next step has a dependency with P2 and P3, like P1 is waiting for a message from P2 and P3, then P2 and P3 are advanced only until the dependency has been satisfied, i.e., P2 and P3 are advance until the messages are sent to P1, thus permitting P1 to advance to the next step. This form of stepping is the most desirable since it is the natural extension for distributed programs of a step in conventional sequential programs. The problem with this method is that there must be some previous knowledge of the dependencies in order to know which process must advance and which should not.

The other method, maximal stepping, works the following way : P2 and P3 are advanced until it reaches a dependency on blocked P1. In other words, P1 is advanced to the next step and halted whereas P2 and P3 are proceeded until it hits an event that requires a future event to be issued from P1. For example, P2 and P3 are advanced and halted at a message receive event since they both need the corresponding message send event to be issued by P1. P1 has not yet reached the send events during the stepping. This form of stepping is simpler to obtain since no previous knowledge of the dependencies are needed.

Stepping with CDB

The stepping module of CDB provides all the features described above, namely fine to coarse grain steps, with causal and maximal stepping protocols. When the breakpoint module halts the system, the stepping module communicates to all the local debuggers where it decides which task should be stepped.

2.3.7 Monitoring

Monitoring supports top-down debugging where it notifies or extracts certain behavioral aspects, or events, of a program that are of interest to the user.

Conceptually, the front end of monitoring and breakpointing are similar. This front end involves the specification of the event and its detection. Following these two phases, monitoring would involve notifying or recording relevant information from the event without halting the system. Breakpointing, on the other hand, does halt the system at the occurrence of the specified event. All issues addressed by breakpointing also must be addressed by monitoring. These issues include specification and detection of the event.

Monitoring with CDB

The monitoring module requires that an input specification (PDL) be supplied such that jobs will be spawned to the local debuggers. The local debuggers are required to detect specific local events. Upon detection of a local event, the local debugger will notify the monitor module of its occurrence where it is then verified if the PDL specification has been satisfied. The monitoring module will either notify the user of the PDL occurrence, via user interface module, or retain data for further investigation.

2.4 User Interface

In common sequential debuggers, the user interface usually consists of a prompt where the user is required to enter a command. A more useful display would be to show the source listing while indicating which line is currently being executed. With this type of user interface one may follow the flow of control where its behavior is being analyzed with the help of the other debugging facilities. This form of user interface is simple to achieve since we are only dealing with one process. But, a distributed programming environment presents many foci of control and attempting to follow the flow of control of all these threads may become overwhelming an thus meaningless. Furthermore, even if following the flow of control were possible, representing all of this information in a meaningful way presents another problem.

Since the debugger is an interactive tool, any user interface development should always keep the user in mind. The main goal is to create an interface that is easy to use and represent debugging information in a meaningful way. Outlined in this section are related works in user interfaces and a description of CDB's user interface and how it achieves this goal.

2.4.1 Related Work

The following are related works in user interfaces for distributed debuggers. They provided a basis for the development of CDB's user interface.

McDowell and Helmbold [McDo-Helm89] present a debugging system that is best suited for distributed programs communicating via shared memory. Their user interface consists of sequential textual lists of processes that interacted with a particular shared object. The screen space is shared by multiple listings.

Harter, Heimbinger, and King [HHK85] describe the IDD debugging system where its user interface consists of a two dimensional grid: one axis represents processes and the other represents progression of time. They use diagonal arrows to represent messages. The problem with their system is that it was designed with the assumption that global time exists. This is really not a valid assumption for all distributed systems.

Socha, Bailey, and Notkin [SBN88] present a system called Voyeur which is a debugger whose user interface is based on animation. Here, whenever the state of the execution changes, Voyeur graphically redraws a new image representing the new state. Hence, an animation of the program is displayed during execution. This is suitable to view one aspect of a program in slow motion.

Hough and Cuny [Hough-Cuny87] present a system, called Belvedere, which is similar to that of Socha, Bailey, and Notkin [SBN88]. They support animation with multiple viewpoints. The viewpoints can be taken with reference to a processor, a channel, or a data item.

2.4.2 Types of User Interfaces

Essentially, there are three main types of user interfaces found in distributed debuggers which are textual representation, space-time diagram, and animation.

Textual Representation

Textual representation is probably the most common form of user interface found in all debuggers where it simply displays the debugging information by means of text. The main disadvantage of this technique is that the overall behavior of a complex distributed system can not be easily captured. More specifically, since common errors in distributed programs are due to message communication and synchronization, providing such information only by means of text is inadequate. On the contrary, textual representation also presents some advantages where states of particular processes may easily be inspected. Infact, most graphical user interfaces include some textual representation to display state information of processes.

Space Time Diagram

The space time diagram (S-T diagram) is a two dimensional representation where one axis indicates the processes while the other indicates time. Figure 2.5 illustrates the space time diagram. This form of display is very useful for displaying the overall behavior of the system where it shows all occurrences of events with respect to each other. Since global time is difficult to achieve, the distances between events do not represent time intervals to any particular scale. For example, referring to figure 2.5, process P0 had occurrences of A followed by D followed by E where it "seems" that interval A to D is larger than D to E. Since these intervals do not have any reference with time the interval A to D may not have taken longer time than D to E. What this diagram does show is the causal ordering of events occurring in the system, i.e., which event came before which other event. The characteristic of this method is that it is best suited to display the overall behavior of the system. State information as well as finer details of individual processes are not presented in this display.

Animation

Animation is another method that displays the overall behavior of the system. For every occurrence of a program event, a graphical image is drawn on the screen. As

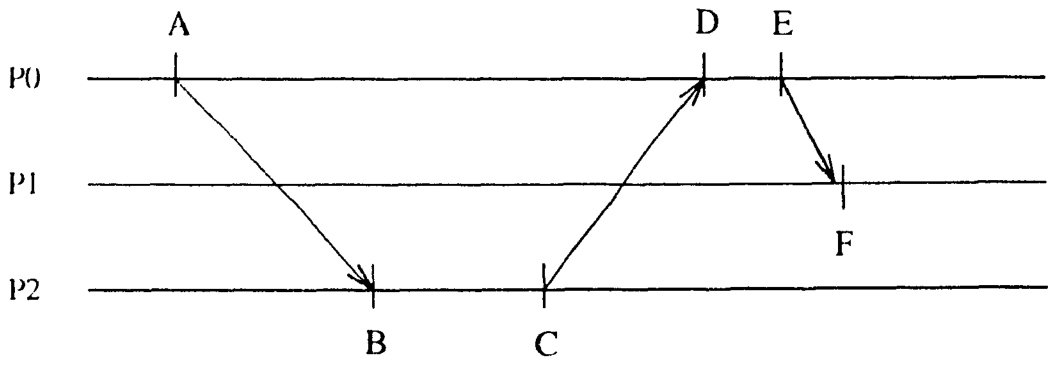


Figure 2.5: ST Diagram

many events occur during the program's execution, many graphical images are drawn which give the effect of an animated view of the programs execution. The animated view of the program behavior is done by placing objects on a two-dimensional display and having them graphically and dynamically interact with each other. For example, objects may represent processes where message send events cause the sender process to draw an arrow from itself to the receiver. Here, the user is always viewing the current state of the program. But a "rewind" facility may be added to view previous states. This may be considered as a disadvantage since a history of all states can not be viewed simultaneously. As in space time diagrams, animation is best suited for displaying an abstracted behavior of the system.

2.4.3 User Interface of CDB

Since textual representation best displays the states of a process and the space time diagram best represents the overall behavior of the program, we use both methods to achieve a meaningful user interface for CDB. Essentially we have three sub-interfaces that service to defining breakpoints, viewing state information, and viewing overall process interaction (space time diagram).

Since our breakpoint specification language consist of hierarchical breakpoint, we propose a graphical interface in which these breakpoints may be created. Basically, the user is able to interactively and graphically draw the desired breakpoint, see

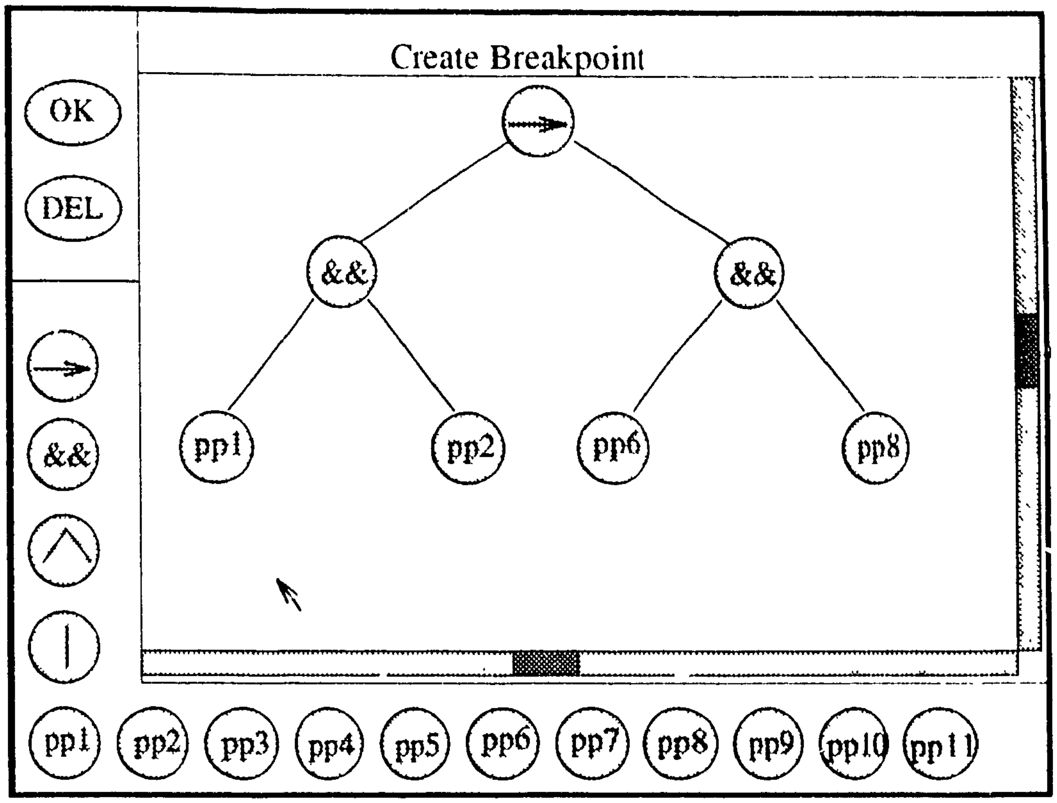


Figure 2.6: Breakpoint Specification User Interface

figure 2.6.

Since state information is best represented by text, we adopt this method by providing a textual display for the state information obtained from a breakpoint cut.

As for viewing the overall process interaction, we use the space time diagram method of representing the process behavior. The space time diagram will display all events that occurred during the execution.

Chapter 3

Two Building Blocks for CDB's Implementation

This chapter describes two generic servers called the vector clock server and the local debugger server that are used in supporting the functionality of other debugger modules. The vector clock provides a facility in which the order of program events may be deduced. The local debugger provides the total control of a single process assigned to it.

3.1 Vector Clock

Determining the global time in a program is a very important factor for debugging systems since most of its facilities depend on the notion of time. Issuing a breakpoint refers to a reference in global time. Stepping involves advancing the execution to the next time interval. Monitoring refers to testing certain predicates at specified times. Trace information is captured at certain times. Checkpoint and rollback involve taking checkpoints and rolling back the execution at specific times.

Thus, the global time plays a big role for any distributed debugging system. More specifically, programs are mapped against global time which allows the debugger to determine the order of events that occurred within the system, we call this *total ordering*. For sequential systems, the total ordering is straight forward since there is only one process and one clock. On the other hand, distributed systems introduce a new problem where there are many processes each containing their own clock. The problem arises in synchronizing these clocks. Since these processes communicate via messages and the communication medium imposes an arbitrary communication delay, perfect synchronization is difficult to achieve. Hence, the concept of a global clock in a distributed system becomes impossible which in turn makes obtaining the total ordering of the system difficult to achieve.

Partial ordering of events in a distributed system is possible to achieve and is also very useful for debugging systems.

3.1.1 Related Work

The following related work are relevant to clocks in distributed systems that were most influential to the implementation of CDB.

Leslie Lamport [Lamport78] introduces partial ordering, total ordering, and the happens before relation which lead to the notion of logical clocks. He then proposes an algorithm that determines a total ordering of the system. Simultaneity has not been addressed correctly since simultaneous events are forced to be ordered by his algorithm. In other words, all events have an ordering with respect to each other.

C. J. Fidge [Fidge] introduces Lamport's [Lamport78] happens before relation followed by an algorithm to determine the partial order of the system. This clocking system is an extension to Lamport's logical clock and it uses vectors to represent the local clock values of all processes. He later presents techniques for detecting temporal errors in both centralized and distributed systems.

Meldal Sankar and Vera [MSV91] propose a more efficient vector clock algorithm compared to that presented by C. J. Fidge [Fidge]. Fidge's method states that each element within the vector represents a process whether or not it does any communication. Meldal Sankar and Vera reduce the size of these vectors where each element in the vector represents only processes that actually do communicate. The problem with this method is that a dependency graph of messages must be created before hand in order to calculate the minimal vector size. Also, dynamically allocating processes require that a dependency graph be re-generated. Though this method saves space, it adds more computational overhead.

Haban and Weigel [Haban-Weigel88] propose a vector clock algorithm similar to that of C. J. Fidge [Fidge]. With this clocking mechanism, they provide definitions for happens before and simultaneity and they show their definitions can be used within a debugging environment.

3.1.2 Vector Clock Algorithm

Our algorithm is taken from C. J. Fidge [Fidge] and Haban and Weigel [Haban-Weigel88] which are both extensions of Lamport [Lamport78] logical clock system to provide partial ordering. It is as follows:

- Let $MAXPROC$ be the maximum number or processes.
- Every process has a local clock that ticks after every occurrence of an event.
- Each process, P_i, has a local vector clock, VC_i, of size $MAXPROC$ elements where each element of the vector refers to a local clock of a particular process and is initially set to zero. $VC_i[i]$ refers to the local clock of P_i and $VC_i[j]$, where $i \neq j$, refers to the local clock of P_j. For every occurrence of an event on P_i the local clock is updated.

  $VC_i[i] = VC_i[i] + 1$.

- Every message sent by P_i is appended with the local vector clock VC_i.

- Every message received by P_i uses the appended vector clock, VC_{rcv_msg}, to update its own vector clock with the following method.

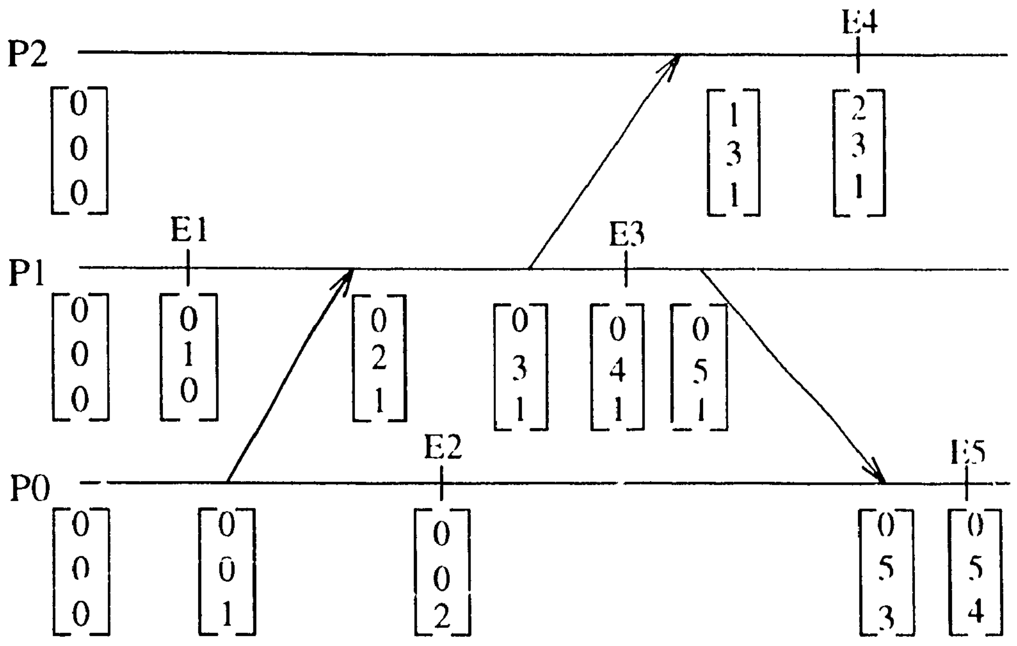
```
For $j = 1..MAXPROC$
    if $VC_{rcv\_msg}[j] > VC_i[j]$
        then $VC_i[j] = VC_{rcv\_msg}[j]$
```

Figure 3.1 illustrates a distributed program with the vector clock algorithm.

As seen from the figure, every occurrence of an event is associated with a unique clock tick called a *timestamp*. A timestamp can be defined as simply a snapshot of a vector clock for a given event. Or

- $TS_{i,j} = VC_i$ at the occurrence of event Ej on process P_i.

For example, referring to figure 3.1, event E1 is has the timestamp [2 3 1].



$$\text{Vector} == \begin{bmatrix} P2 \\ P1 \\ P0 \end{bmatrix}$$

Figure 3.1: Example of vector clock

Happens Before

The *happens before* relation was originally developed by Lamport [Lamport78] and is as follows:

- if E_i and E_j are events within the same process and E_i happens before E_j then $E_i \rightarrow E_j$.
- if E_i and E_j are events from different processes where E_i is the sending event and E_j is the receiving event, then $E_i \rightarrow E_j$.
- if $E_i \rightarrow E_j$ and $E_j \rightarrow E_k$, then $E_i \rightarrow E_k$.

Since our vector clock algorithm was based on Lamports [Lamport78] logical clock system and the happens before relation, determining the order of events are done in the following manner:

- Let event E_x occur on process P_i with timestamp $TS_i.x$ and E_y occur on process P_j with timestamp $TS_j.y$. Let $MAXPROC$ be the maximum number of elements in the timestamp.

Assertion 1 For $k = 1..MAXPROC$,

If $TS_i.x[k] \leq TS_j.y[k]$ then
$E_x \rightarrow E_y$

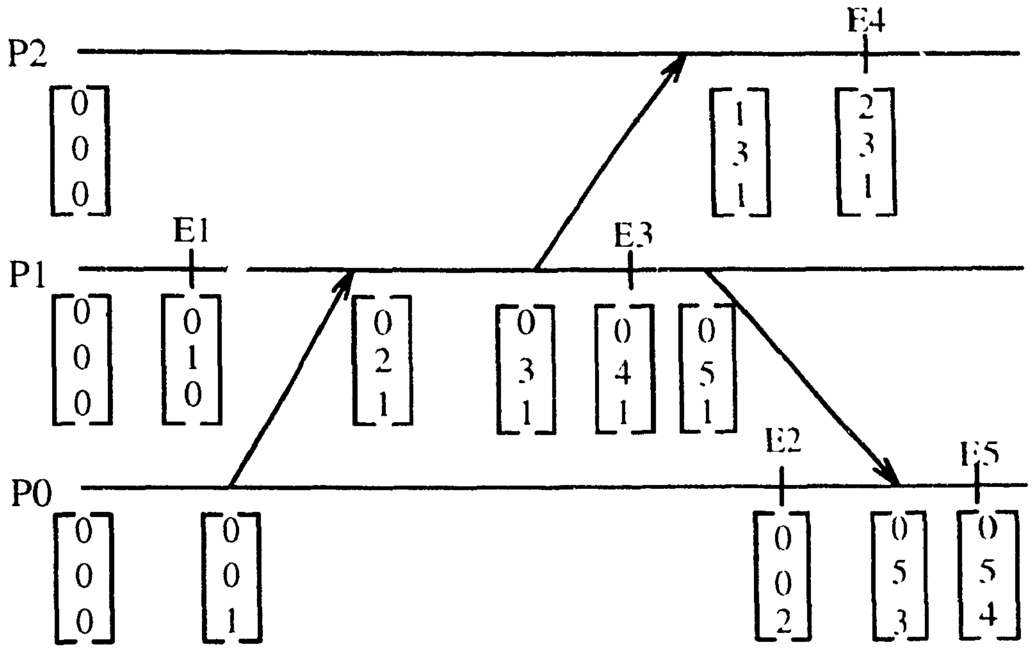
Assertion 2 For $k = 1..MAXPROC$,

If $TS_j.y[k] \leq TS_i.x[k]$ then
$E_y \rightarrow E_x$

Now, referring back to figure 3.1, we can say that $E1 \rightarrow E3$, $E1 \rightarrow E4$, $E1 \rightarrow E5$, $E3 \rightarrow E5$, and $E2 \rightarrow E5$.

Simultaneous

Since we are determining partial orders there may be a situation where Assertion 1 and Assertion 2 are not satisfied. Take for example E2 and E3 in figure 3.1. According to the diagram, it seems that E2 happened before E3. Without changing the diagram in figure 3.1 we can redraw the diagram as shown in figure 3.2. Now it seems that E3



$$\text{Vector} == \begin{bmatrix} P2 \\ P1 \\ P0 \end{bmatrix}$$

Figure 3.2: Vector Clock

happened before E2. In reality, perhaps E2 did happen before E3 or vise versa or E2 happened exactly at the same time as E3. According to the vector clock algorithm, there is no way of knowing their exact order. We define *simultaneous* relation as the case when both assertion 1 and 2 fail. Simultaneity refers to a region where two events may or may not have occurred at the same exact time. Figure 3.3 illustrates the notion.

3.1.3 Vector per Process Vs. Vector per Machine

One of the main disadvantages of the vector clock algorithm is that for every process there exist an element within the vector, *vector per process*. As distributed programs becomes very large, the vectors occupy more space. Also since the vectors are ap

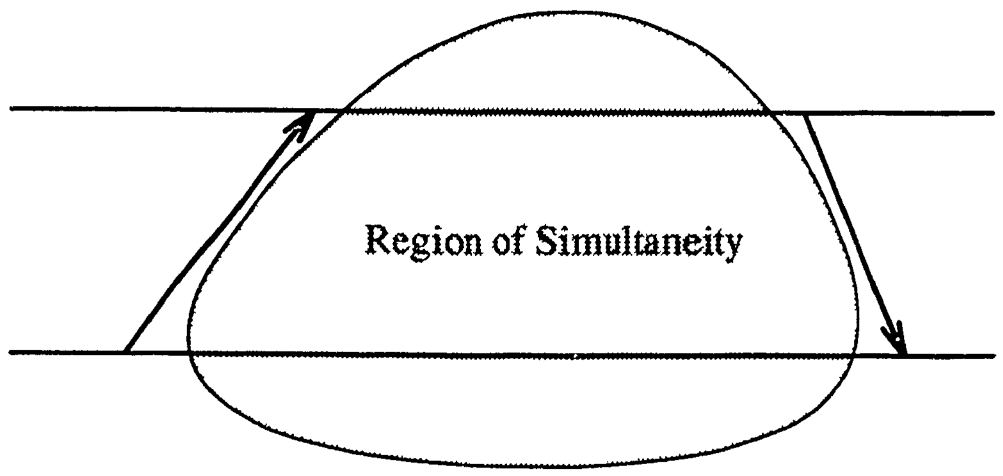


Figure 3.3: Simultaneous Region

pended to messages, the messages themselves grow which induces a greater probe effect. Another disadvantage is dynamically allocating processes. For every process dynamically allocated, all vectors must be re-adjusted in order to incorporate the new process's local clock. This again would induce a large probe effect since additional overhead is needed.

One remedy to this disadvantage is to have every process maintain a vector in which each element represents a machine, *vector per machine*. In other words, each physical machine will maintain one local clock which all processes of that machine will share. The size of the vector is thus reduced since the number of machines are usually less than the number of processes. Also dynamically allocating processes simply involves having another process share the local clock of the machine.

The disadvantage of this solution is that it forces processes on the same machine to be totally ordered. In other words, any two events from any two processes running in parallel on the same machine will always be ordered. This solution would be acceptable if the machine contains a single processing unit. Here, processes are really running in pseudo-parallelism where events are being executed sequentially anyways. But, on the other hand, if the machine is a multi-processor system, forcing total

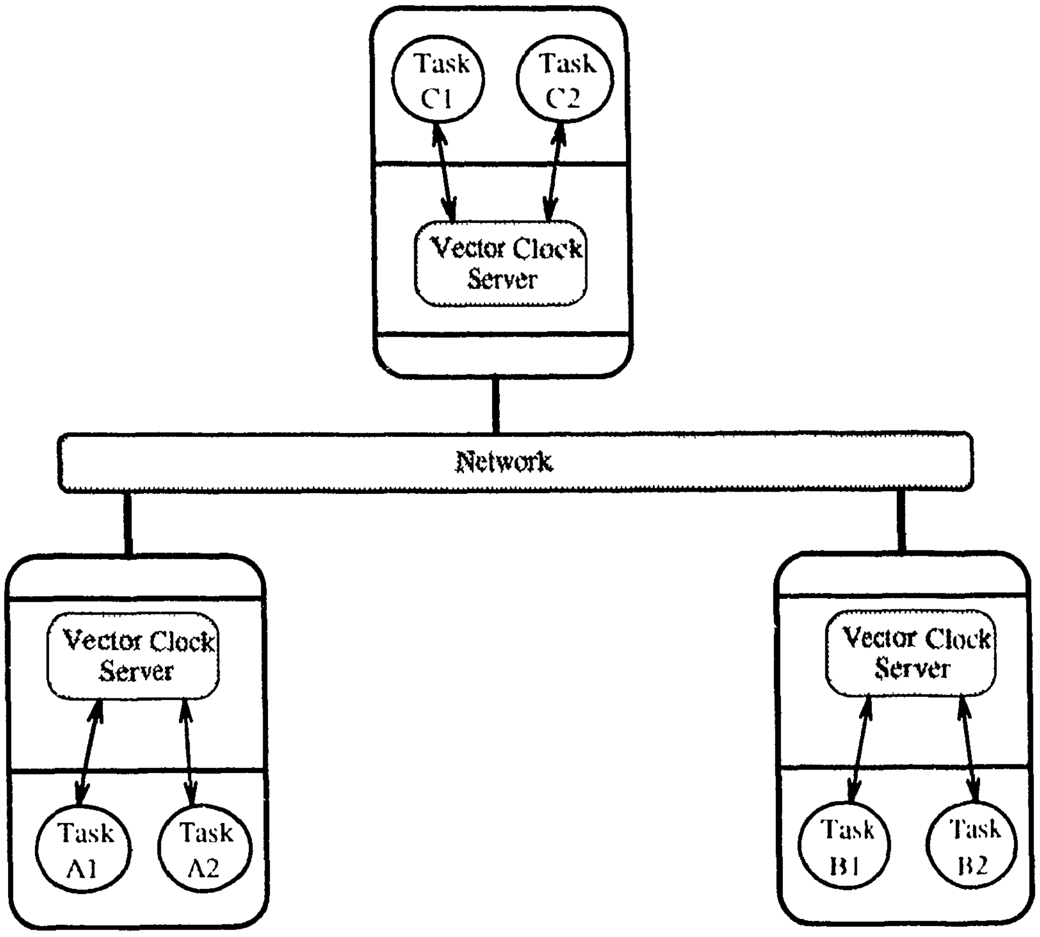


Figure 3.4: Vector Clock Servers

ordering is unacceptable.

3.1.4 Vector Clock Server

With respect to CDB, we have built a vector clock system that follows the vector clock algorithm stated above. Since CDB's environment consists of several single processor machines, we adopted the vector per machine solution. Essentially, for each machine, a vector clock server is created where it maintains one vector for all processes on the machine. Any occurrence of an event causes the process to poll the corresponding vector clock server to either cause the machine's local clock to tick or to get a timestamp of the event. Figure 3.4 illustrates the vector clock servers.

3.2 Local Debugger

Recall that a distributed program consists of a set of sequential programs communicating with each other. If we extend this definition to encompass distributed debuggers, we may define a distributed debugger as a set of sequential debuggers communicating with each other. We call these sequential debuggers as *local debuggers*.

Essentially, the job of a local debugger is to control the execution of one sequential program. Since these local debuggers are similar to sequential debuggers as we commonly know , they perform the functions of halting and continuing execution, breakpointing, stepping, examining data, obtaining state information, and altering execution.

Local debuggers play a big role in a distributed debugger since it services the development of all facilities such as distributed breakpoints, distributed stepping, checkpoint and rollback, monitoring, and vector clocks.

Figure 3.5 illustrates local debuggers with respect to the overall CDB.

3.2.1 GDB and its Functionalities

The local debuggers in CDB were built from a GNU product called GDB which is a source level debugger for C and C++. Essentially, we have modified the GDB source code while maintaining all of its original functionalities. The features of the standard GDB are as follows:

Breakpoints. Different forms of breakpoints may be set, such as break at a specific line number, break upon entry to a procedure, break at a particular address, and break at an offset from the current line. Conditions may also be used to augment the stated breakpoints, for example, break at procedure X if its arguments are less than 10. Also any breakpoint may be labelled as temporary where they will be deleted after the first occurrence of the breakpoint has been satisfied.

Stepping. Stepping involves advancing to the next line(s) of execution, advancing the program until a selected "stack frame" returns, executing through an entire procedure, advancing the program past a loop, or advancing the program to a selected label. It also provides execution of one machine instruction at a time.

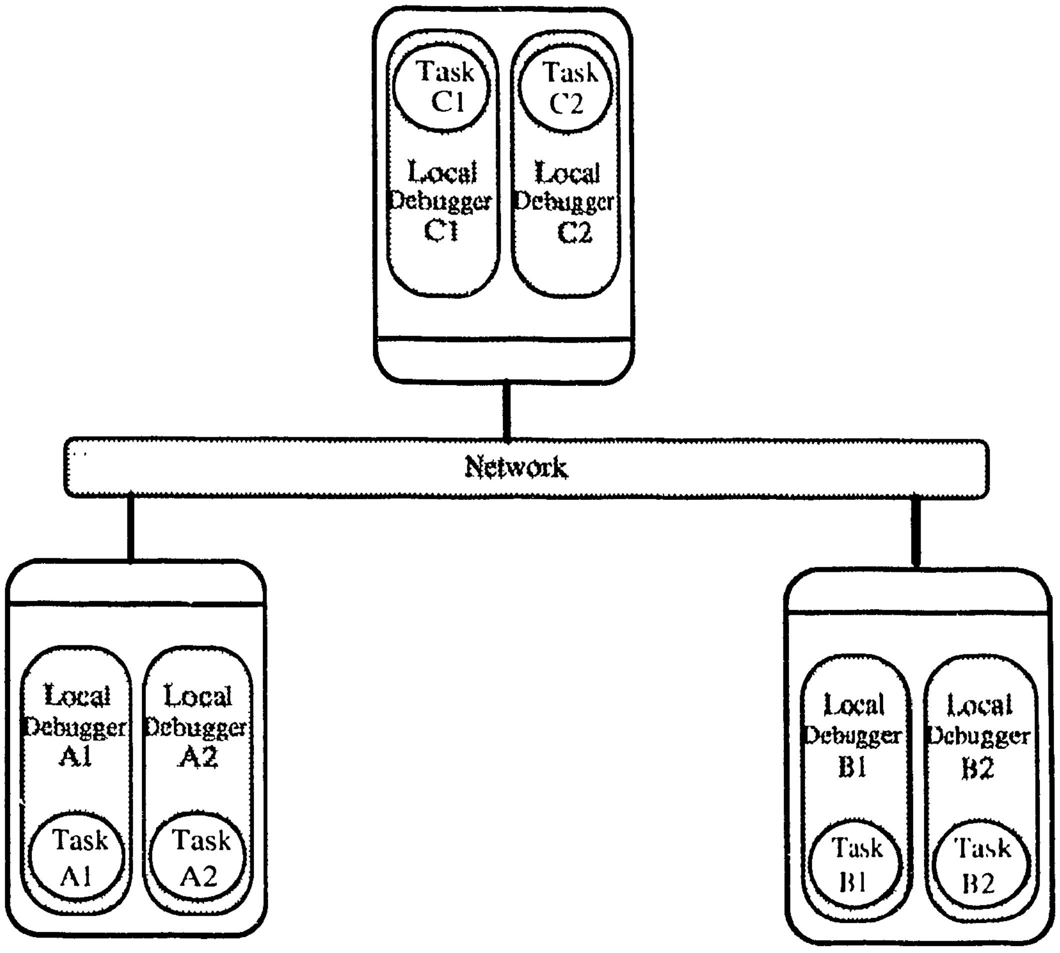


Figure 3.5: Local Debuggers

Stack Examination. This facility provides information on the stack frames where selected frames may be analyzed. Information such as address of the frame, addresses of the called by and the caller frames, arguments of the frame, and local variables of the frame are provided.

Source File Examination. Source code from any file may be displayed at anytime.

Data Examination. Examination of the data involves displaying values of certain variables, evaluating debugging expressions, and displaying arrays of specific types. Also examination of the memory may be done without reference to the program's data types where registers, such as the program counter, and memory contents may be obtained. GDB also saves all data examined throughout the debugging session.

Symbol Table Examination. Symbol table examination provides the information of addresses of certain symbols stored, the type of symbols, the names and data types of all defined functions or variables, and the names and data types of all functions or whose names match a given regular expression.

Altering Execution. This facility allows modification to existing program variables, modification to a specific memory location, and continuing the execution from a different address.

3.2.2 Mach's Extensions to GDB

Originally, GDB is designed to debug single threaded Unix applications with no extension for multi-threaded tasks that are commonly found in Mach. For this reason, Caswell and Black [Caswell-Black89] provided an implementation of a debugger that supports multiple threads. They modified GDB version 3.4 for the support. With the modified version, the user may choose any thread to analyze during a debugging session.

3.2.3 Our Extensions to GDB

Since Mach's version of GDB suits best the Mach environment for tasks and threads, our extensions were done on this version. Basically, we kept all functionalities of GDB and converted it to a local debugger where it can be executed and be controlled from a remote site

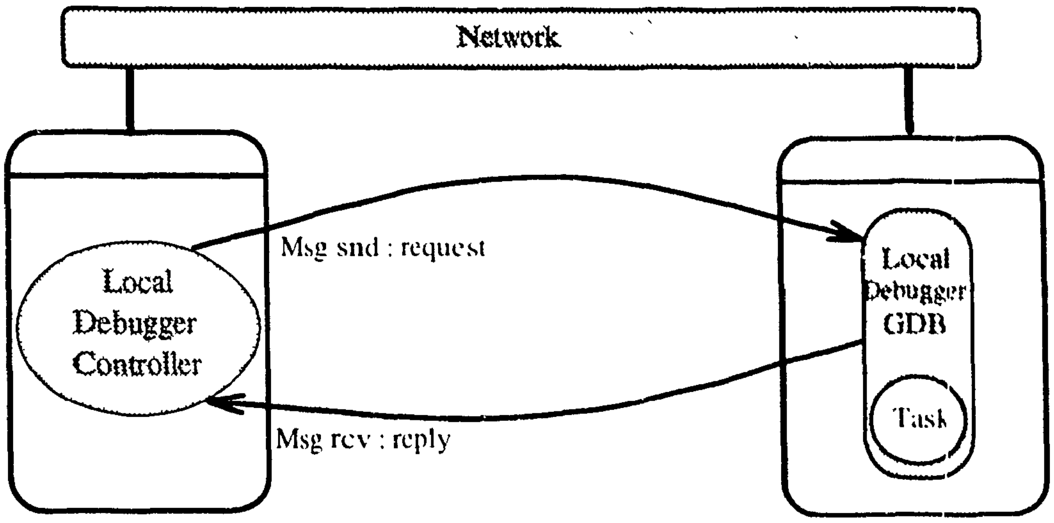


Figure 3.6: Our Extension to GDB

Basically, we removed the user interface of GDB and replaced it with Mach IPC primitives. All input commands are replaced by a message receive and all output commands are replaced by message sends. This will enable the control of GDB to be situated at any site and a GDB command would consist of request message to the local debugger and its reply would be received in the returning message. Figure 3.6 illustrates the our extension to GDB.

Chapter 4

Distributed Breakpoints

A *distributed breakpoint* can be viewed as a collection of sequential or local breakpoints. It is different from sequential breakpoints in specification, detection, and halting the underlying distributed program.

New Dimensions in Distributed Breakpoints

Since we are dealing with several concurrent processes, keeping track of the detailed aspects of all processes becomes an unmanageable task for the user. The user may not even know all the detailed aspects to define a precise breakpoint. However, the user will have a particular view which may be at a higher level of abstraction than the program's code space. For example, the user may have a view point with respect to the synchronization space. It is this view that will help specify a breakpoint that will be useful in debugging. Once the user gains more knowledge about the program, a more detailed breakpoint may be constructed. We define *fine breakpoints* to be those breakpoints that specify more details about the program and *coarse breakpoints* to be breakpoints that take a higher level abstracted view of the system.

Another type of breakpoint is called *conditional breakpoints*. The program is halted at some point in its execution if the specified condition is satisfied. In sequential systems, conditional breakpoints may be constructed as follows: "Break when variable X is 10". Distributed programs, on the other hand, introduce an additional dimension where conditions may span across multiple processes. For example, "break when $X < 10$ in process P1 *and* $Y < 9$ in process P2. This breakpoint can be viewed as two sequential (conditional) breakpoints with an added relation (condition) between them. This relation may be temporal or non temporal

In sequential programs, a breakpoint is associated with only one process. Distributed breakpoints, do not have this limitation. They may be associated with some or all of the processes. Then a question arises: What protocol should the processes, not associated with the breakpoint, follow when the breakpoint is detected? Should they continue to run or stop? Protocols outlined in this chapter are the *causal breakpoint* [Fowler-Zwaen90], *maximal breakpoint* [H.F. Li], and the *immediate breakpoint*

In summary, the distributed breakpoint is a subproblem in distributed debugging where it introduces four major new dimensions of specification, protocol for termination, halting, and managing the vast amount of information generated. The research done by various groups in the area of distributed breakpoints revolve around the above four issues. In this thesis we address the first three dimensions where the forth needs further study.

Related Work

Miller and Choi [Miller-Choi88-2] present a specification method for distributed breakpoints and algorithms for its detection and halting. Their breakpoint specification is made up of *simple predicates* which are typical predicates used in sequential debuggers. Simple predicates form the basis for other predicates called *disjunctive*, *conjunctive*, and *linked*. A disjunctive predicate is satisfied when one or more of its simple predicates occur. A conjunctive predicate is satisfied when all of its simple predicates occur. Linked predicates specify an ordering between disjunctive predicates and is satisfied when this ordering occurs. This specification method is sufficient if the user is only interested in breakpoints based on the happens before relation. What this method lacks is dealing with aspects of simultaneous predicates. Since linked predicates comprise simple and disjunctive predicates, Miller and Choi present a detection algorithm based on linked predicates. Their algorithm is as follows. The linked predicate starts on the process where the first simple predicate is expected to occur. After its occurrence, the linked predicate is sent to the next process where the next simple predicate will occur. This pattern continues until the last simple predicate has occurred where the system is halted. This algorithm provides a sufficient means of detecting a linked predicate but lacks in halting the system soon enough to be meaningful. Once a linked predicate is detected, the processes are halted at a state past the specification point.

Bates and Wileden [Bates-Wileden82] describe an Event Definition Language, EDL, for specifying important events in monitoring. Essentially, it provides users with a means of clustering and filtering a system's event stream in order to obtain a behavioral abstraction. EDL defines a set of *primitive events* that characterizes the lowest level events possible within the system. These events are clustered together to construct higher level events which again may be clustered to form even higher level events. As more events get clustered, the more detailed the specification becomes. Each defined event may also be qualified by placing conditions on certain *variables* associated with the event. A problem with this form of specification is the potentially vast amount of event variables. As the EDL specification grows large, the amount of event variables grows very large which may become unmanageable for the user. Though EDL is designed for monitoring, its semantics fit very well for breakpoint specification.

Fowler and Zwanepoel [Fowler-Zwaen90] present an algorithm for achieving causal breakpoints. They assume that a breakpoint specification consists of a set of independent local breakpoints on a set of processes. This is a very simple view of a breakpoint since it does not incorporate dependencies between local breakpoints. Their algorithm lets each process containing a local breakpoint advance until it reaches the breakpoint. All processes without local breakpoint advance to the *earliest state* which permits the other process to reach their local breakpoint. This algorithm assumes that some previous knowledge of all events exist such that the earliest state may be calculated. The algorithm also induces a large probe effect since all processes must verify for every event whether it should halt.

Haban and Weigel [Haban-Weigel88] implemented a debugging system based on breakpoints which consists of a special debugging hardware environment. This environment has two separate networks, one of which is dedicated for debugging purposes whereas the other is for the application. This eliminates the probe effect within the main application. They propose a method of specifying distributed breakpoints in a hierarchical format that is similar to EDL of Bates and Wileden [Bates-Wileden82]. The specification method provides a means to qualify primitive predicates but lacks in qualifying an event to more finer detail. For example, the *send* primitive can only be qualified by its *portID*. Further qualifiers such as message contents, message length, message type, etc. would be useful to further refine the event. Haban and Weigel also propose a detection and halting algorithm is as follows : Every process maintains a copy of the distributed breakpoint. Every occurrence of a primitive predicate results in a broadcast message sent to all other process to indicate that the event occurred. Eventually, as all primitive predicates occur, one of the processes will detect that the distributed breakpoint has been reached and the system is halted. This detection and halting algorithm is distributed which does not easily map to the typical centralized debugging model of a user performing the debugging at one site. The research of Haban and Weigel has been the most influential for CDB and is the closest work to our implementation.

4.1 PDL : Predicate Definition Language

As mentioned above, breakpoint specification is an issue that must be well addressed. It should be flexible enough so that different types of breakpoints may be formed as well as they be meaningful to the user. With this in mind, we propose PDL, Predicate Definition Language which is a breakpoint specification language. To put PDL into context with the needs of breakpoint specifications, we will first describe the relationship between the user and the breakpoint.

4.1.1 The User and the Breakpoint Relation

The breakpoint specification and data examination are two activities performed by a user who is debugging a program. The user may perform these two activities alternatingly and repeatedly. The user first starts off with a broad idea of where the bug may occur and sets breakpoints at those deduced areas. The system halts at these areas and displays *relevant* data. The user examines them to obtain more knowledge about the program. Then, new breakpoints can be re-inserted based on this knowledge. Thus a breakpoint can serve as a *coarse comb* to narrow down to the region of the anticipated bug.

In cyclic debugging, the two crucial parts are the definition of the breakpoint and the examination of program test results since these are the tools that the user will use to locate the bug. Ultimately, it is left to the user to use these tools effectively in order to locate the bug. Hence, debugging tools such as breakpoints assist a user in finding a bug but provides no guarantee that the bug will be found.

What the User Knows? Before attempting to specify a breakpoint in a distributed system, it should be clear what the user is expected to know about the program before execution. In general, the user is assumed to know:

1. The distributed program structure and behaviour of the program.
2. The imports and exports of each module.
3. Invariants.
4. What are communicated between processes.
5. Port structures.
6. Data structures.
7. Critical regions.

4.1.2 Types of Predicates

When a breakpoint is set, the debugger monitors the running program for the breakpoint's occurrence. A breakpoint is considered as a predicate as it evaluates to "true or false" depending on the program state. The term breakpoint and predicate are used synonymously hence forth.

A predicate is composed of *primitive predicates*, evaluation methods of which are known a priori to the system. A composed predicate is called a *global predicate*. For this thesis, the composition is represented in form of a binary tree. Predicates closer to the root of this tree encapsulate more information about the system whereas predicates closer to the leaf nodes which are primitive predicates specify less information.

Primitive Predicates

Essentially, primitive predicates describe code level system behavior such as an assignment to a variable, entry to a procedure, etc. At this level, one can define very many types of primitive predicates. But, not all of them may be useful within the context of defining a global predicate. One must keep in mind that the set of selected primitive predicates must be useful for constructing meaningful global predicates. The size of the primitive predicate set is another factor that must be considered. A very large set puts a burden on the user to learn more about them. A very small set may be easier to learn and manage but may not be adequate. One must must weigh these pros and cons and by experience come up with a useful set.

Based on what the user knows, section 4.1.1, we propose a first draft of fourteen primitive predicates which may expand or contract based on further research and experience. These predicates may be subdivided into three categories based on message transmission, internal state, and program code.

Type 1 (Based on Message Transmission) : send, receive, port allocate, port set allocate, port deallocate, port set deallocate, port set add, port set remove.

Type 2 (Based on Internal State of Process) : assignment to variable, process start, process stop.

Type 3 (Based on the Code Level) : function/procedure enter, function/procedure exit, line number execution.

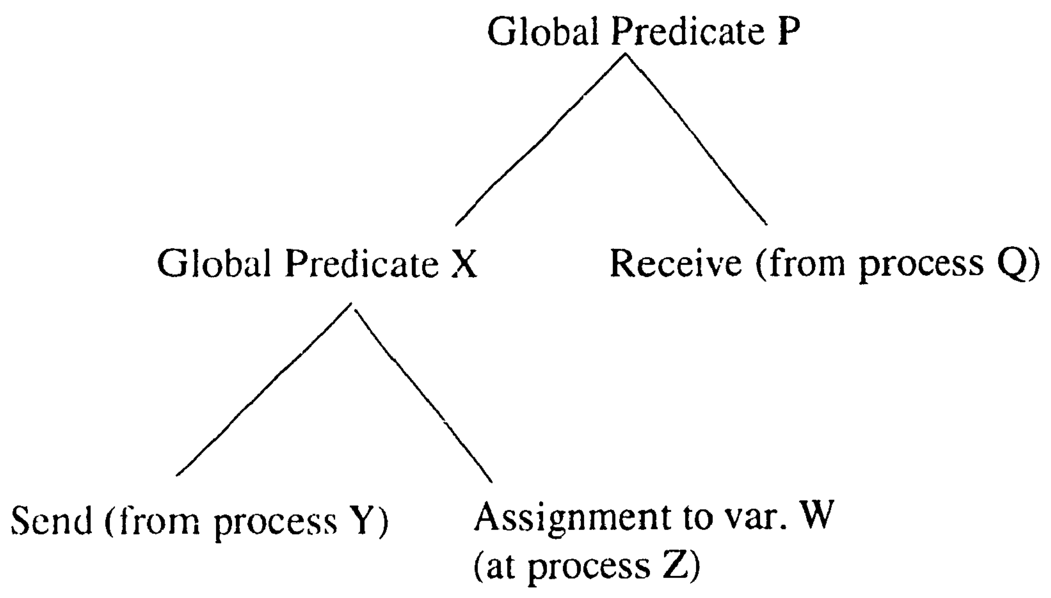


Figure 4.1: Global Predicate

Global Predicates

A global predicate is a tree of several predicates, the leaf nodes of which are primitive predicates. For example, see figure 4.1. The predicate X is composed of primitive predicates send (from process Y) and assignment to variable W (at process Z). Another example would be global predicate P is composed of global predicate X and primitive predicate receive (at process Q). One may intuitively feel that the higher the predicate tree, the more knowledge the user is required to have (in order to construct the tree), and the less output data will be produced by the debugger. We call this type of predicate as a *fine grain* predicate. Also, a flatter tree requires less user knowledge and results in more output data and it is called a *coarse grain* predicate.

4.1.3 Relation Amongst Predicates

Global predicates are formed from other predicates, primitive or global, by combining them using relations. These relations can be divided into two groups: temporal and non-temporal. Temporal relations involve the happens before and simultaneous relations. Non-temporal relations involve alternation, and conjunction.

Temporal Relations

In a sequential system, temporal relations are clear since the execution of a program is limited to (conceptually) one processor where the order of primitive predicates can easily be determined. On the other hand, determining order of primitive predicates for a distributed system is not as trivial since execution of a program is dispersed amongst many processors. We can make use of Lamport's [Lamport78] happens before relation that defines causal relations between predicates.

Happens Before Relation

We denote the happens before relation with a right arrow:

global predicate X := predicate A $\rightarrow$ predicate B

where predicate A and B can either be a primitive or a global predicates. This relation implies that global predicate X is true only if predicate A is true before predicate B becomes true.

Predicate A and predicate B both have timestamps indicating when they became true. In the case when global predicate X evaluates to true, it will inherit the timestamp of predicate B since it was the last predicate that made X true. Figure 4.2 illustrates this.

Simultaneous Relation

Since exact global time is impossible to obtain in a distributed system, simultaneous relation amongst predicates adopts the definition stated in section 3.1.2. Predicates are considered simultaneous if there are no causal (or temporal) relation between them. Take for example figure 4.3 where:
global predicate X := predicate A && predicate B.

In reality, A and B may not have occurred simultaneously. But since A and B have

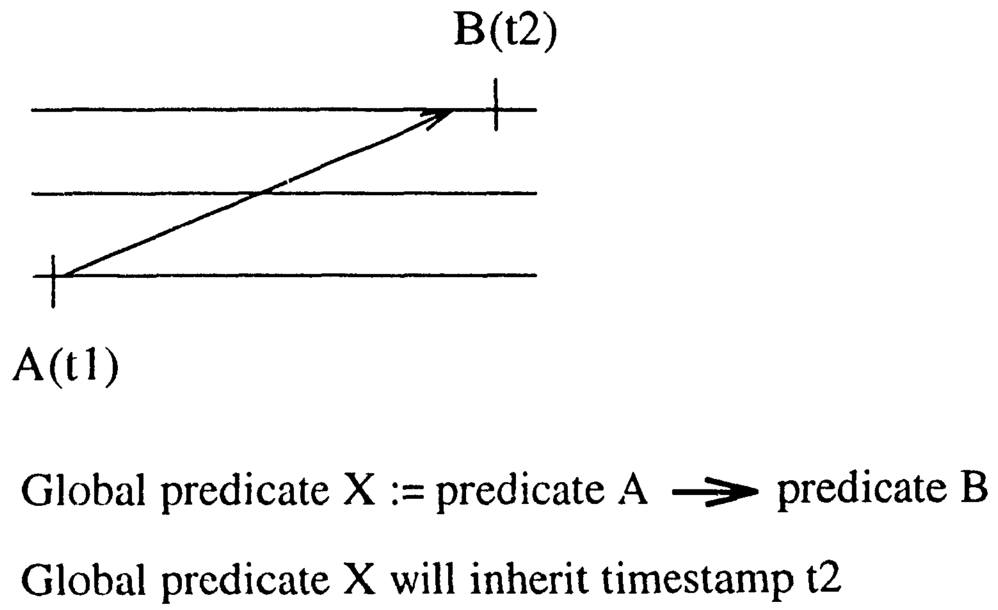

B(t2)
A(t1)
Global predicate X := predicate A → predicate B
Global predicate X will inherit timestamp t2

Figure 4.2: Time Stamp Inherit

no causal relation, i.e., their timestamps cannot indicate whether one came before the other, A and B are considered simultaneous. If global predicate X evaluates to true, it will inherit the last predicate that caused X to become true, which could be A or B.

4.1.4 Non-Temporal Relations

Two non-temporal relations used in PDL, as of now, are conjunction and alternation.

Conjunctive Relation

The conjunctive relation is denoted by a $\wedge$. For example,

Global predicate X := predicate A $\wedge$ predicate B

where predicate A and B can either be a primitive or global predicate. Global predicate X will evaluate to true if both predicate A and predicate B evaluate to true regardless of their causality.

The global predicate X will inherit the timestamp of the last predicate that caused X to become true.

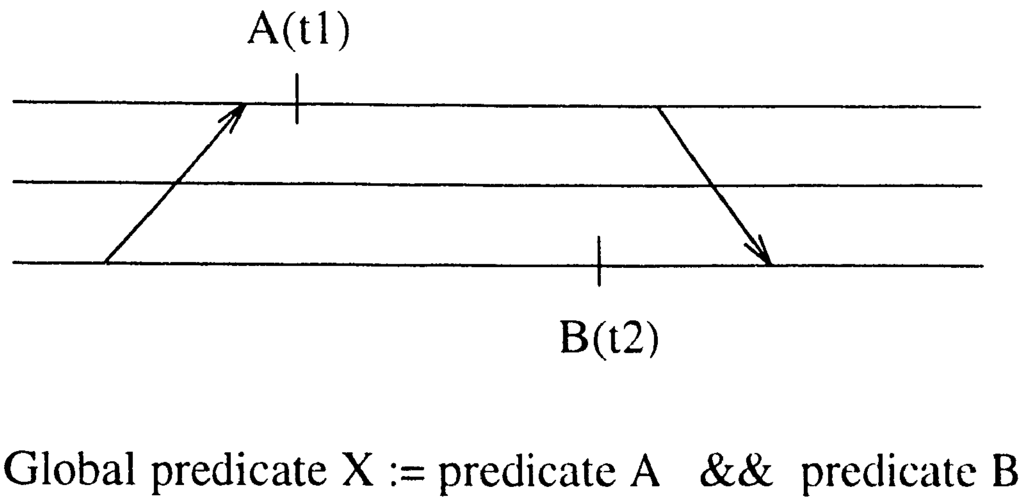


Figure 4.3: Simultaneous Predicate

Alternative Relation

The alternative relation describes the occurrence of either one or both predicates evaluating to true. For example,

Global predicate X := predicate A | predicate B

where predicate A and B can either be a primitive or global predicate. Global predicate X will evaluate to true if either A or B, or both, evaluate to true.

Global predicate X will inherit the timestamp of the first predicate that caused X to become true.

4.1.5 Filtering Facility

Filtering is a facility provided as part of PDL. The objective is to reduce the output data generated that has to be digested by the user. Filtering is accomplished by qualifying the primitive predicates. An unqualified primitive predicate has a tendancy to produce a large amount of output. Take for example the PDL predicate

```
predicate X := send
```

which means that user defined predicate X consists of the occurrence of primitive predicate send. This may not be detailed enough because for every send during the program execution the predicate will evaluate to true. What a user needs is a way to filter out unwanted sends. PDL provides this filtering facility where occurrences of predicates can further be refined. For example, a more meaningful PDL predicate could be

```
predicate X := send
            condition : send.length < 10
                        send.task_name == P1
```

This states that predicate X consists of the occurrence of primitive predicate send from process P1 and whose message length is less than 10.

Hence, for each primitive predicate, there are associated variables that are accessible by the user such as send.length and send.task_name.

In PDL, each primitive predicate is considered an object, much like an object in the object oriented paradigm. Each object encapsulates a set of variables which is updated every time the predicate evaluates to true and the values are accessible to users. For example, "send" has its associated variables of task name, destination task, destination port, message contents, message length, message type, message id, and message return port. When the send primitive occurs, all of the variables are updated and accessible through the notation send.α, where α is the name of the variable.

The following is a list of variables that each primitive predicate encapsulates.

Send: task name, destination task, destination port, message contents, message length, message type, message id, message return port.

Receive: task name, source task, message contents, message length, message type, message id, message return port.

Port allocate, Port deallocate, Port set allocate, and Port set deallocate: task name, port name.

Port set add, and Port set remove: task name, set name, port name.

Assignment to variable: task name, value of variable, encapsulating procedure/function.

Process start, and Process stop: task name, pid, parent pid, processor number.

Function/Procedure enter and exit: task name, passed parameters, the calling function/procedure, return value.

Line number execution: task name, source code at that line number.

PDL provides a filtering facility for the primitive predicates so that "meaningful" breakpoint specification of varying granularity may be created by the user.

4.2 Detection and Halting

Once the breakpoint specification using PDL is defined, the task remains to detect for its occurrence and halt the system.

4.2.1 Types of Halting

As mentioned earlier there are three types of halting protocols: Causal breakpoint, maximal breakpoint, and immediate breakpoint. Briefly, all three protocols require that all processes participating in the global predicate are required to halt at the location of the corresponding primitive predicates. And, the remaining processes, if any, (non-breakpoint processes) must halt at the location determined by the protocol.

Causal Breakpoint

To obtain a causal breakpoint, non-breakpoint processes are halted at the earliest point in their execution that will permit the breakpoint to be achieved. Consider figure 4.4 where the breakpoint is marked in processes P1 and P2 (square boxes).

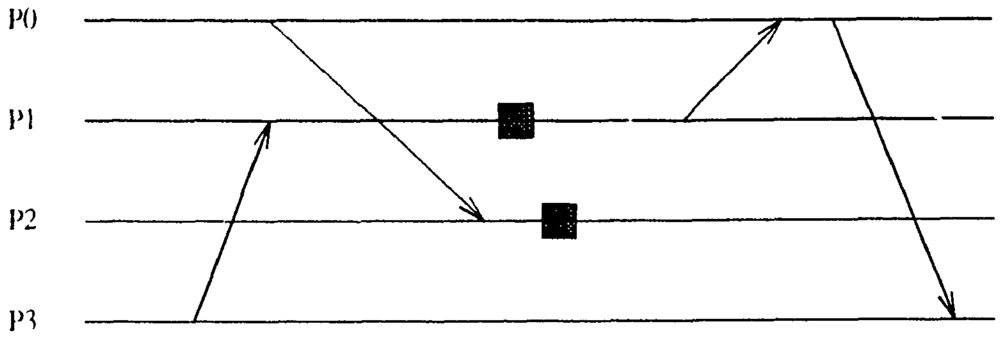


Figure 4.4: Distributed Breakpoint

The causal breakpoint is shown in figure 4.5 by the solid black line. Processes P0 and P3 are halted after the respective send command which Enabled P1 and P2 to reach their respective breakpoints.

Maximal Breakpoint

Maximal breakpoints are the opposite of the causal breakpoint. All processes not within the breakpoint are halted at the latest point possible in their execution. Halting may derive from one of two scenarios. The first is from a receive statement. If a process is waiting for a message from a process that is already halted due to the breakpoint, then this process is halted at the receive statement. The second scenario is if a process has no further dependencies, i.e., waiting for a message. from the processes halted due to the breakpoint, then the process may execute until termination. For the breakpoint in figure 4.4, the maximal breakpoint is illustrated in figure 4.6.

Immediate Breakpoint

A causal breakpoint shows the earliest point of execution and the maximal breakpoint shows the latest. It may be desired that all processes not within the breakpoint be

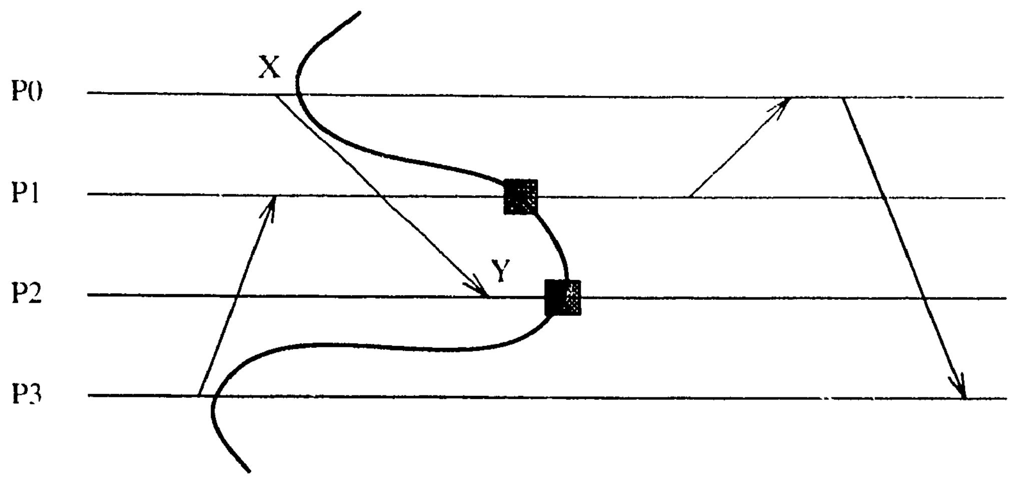


Figure 1.5: Causal Breakpoint

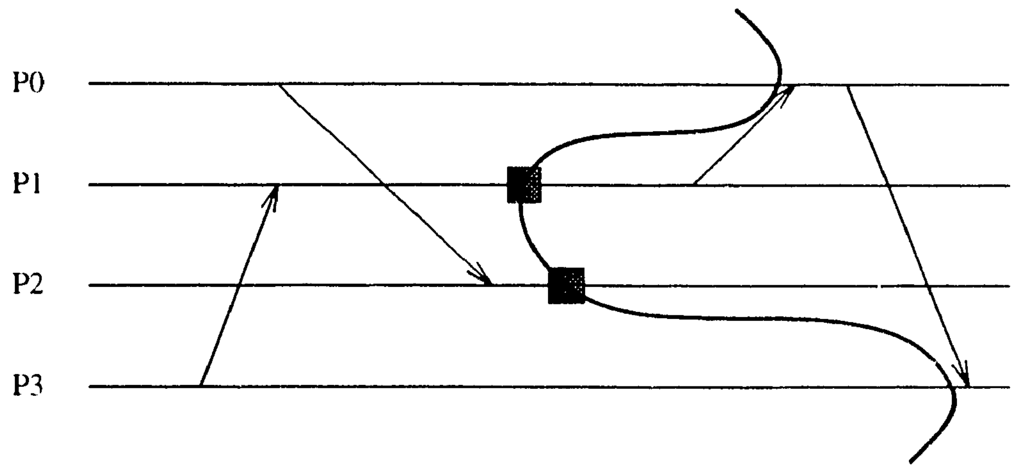


Figure 1.6: Maximal Breakpoint

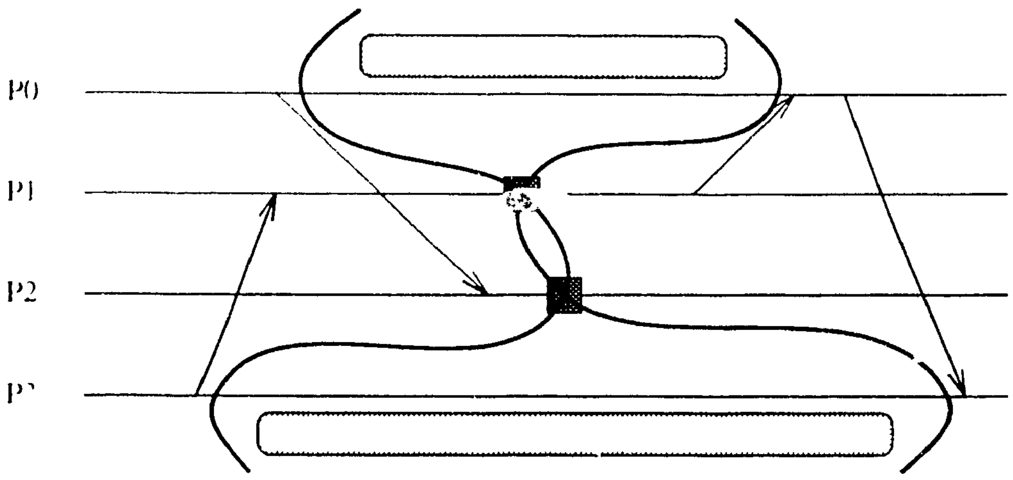


Figure 4.7: Immediate Breakpoint

halted **whenever** the breakpoint has been detected. This is called an immediate breakpoint. Here, the execution of the non-breakpoint processes may halt anywhere within the area denoted by the corresponding causal and maximal breakpoint cuts. Figure 4.7 shows the area, shaded rectangles, of where P0 and P3 may halt with respect to the original breakpoint in figure 4.4.

4.2.2 Difficulties in Obtaining these Cuts

Of all these, the causal breakpoint is the most difficult to obtain. The difficulty arises because a non breakpoint process is required to have knowledge of the future. In figure 4.5, P0 at point X should have knowledge that Y (which follows X) will cause a breakpoint and thus P0 should stop right after X. We call this halt point the *earliest state*. Even if this future knowledge were available, there would still be a considerable amount of probe effect since all non-breakpoint processes are required to perform some verification at every event occurrence. With the record and replay approach, future knowledge is achievable.

The maximal and immediate breakpoints are easier to obtain since no further knowledge is required.

One problem that all three protocols exhibit is when a breakpoint predicate eval-

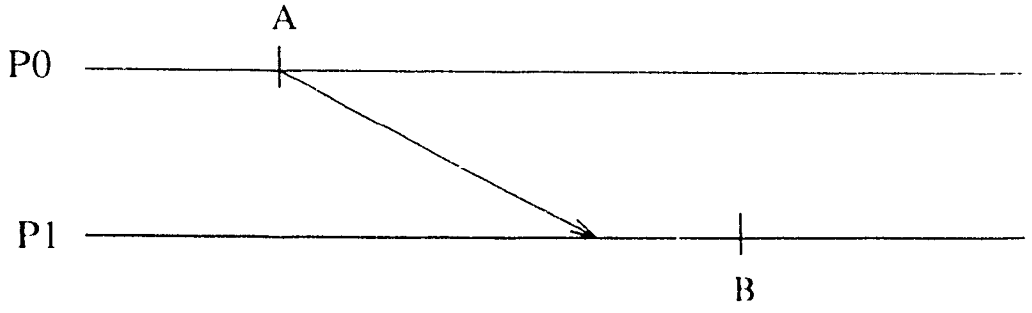


Figure 1.8: Global predicate := A → B

uates to false. Up until now we've talked about setting, detecting, and halting break points assuming that the breakpoints are reachable. In a realistic environment, an impossible or unattainable breakpoint may have been specified. Consider figure 1.8, where the global predicate is defined as

Global predicate := A → B.

Here we are looking for the occurrence of primitive predicate A followed by primitive predicate B. Since there is no way to know if A or B will occur, there is no guarantee that the breakpoint will be reached. Therefore, for any occurrence of a primitive predicate, the corresponding local process should not be halted. For example, referring back to figure 1.8, let primitive predicate A occur first. Since there is no guarantee that B will occur, A will be permitted to execute. Once B occurs, the global predicate is satisfied and the system is halted. The result is shown in figure 1.9.

This cut is neither causal, maximal, nor immediate. One may even classify this cut as meaningless since process P0 has advanced past point A and has potentially lost any state information at the desired location. Stepping on P0 now starts at a new unknown location which may again be meaningless.

A solution to this problem would be to use the checkpoint and rollback mechanism. Once we know that a global predicate is obtainable, we can rollback the execution and

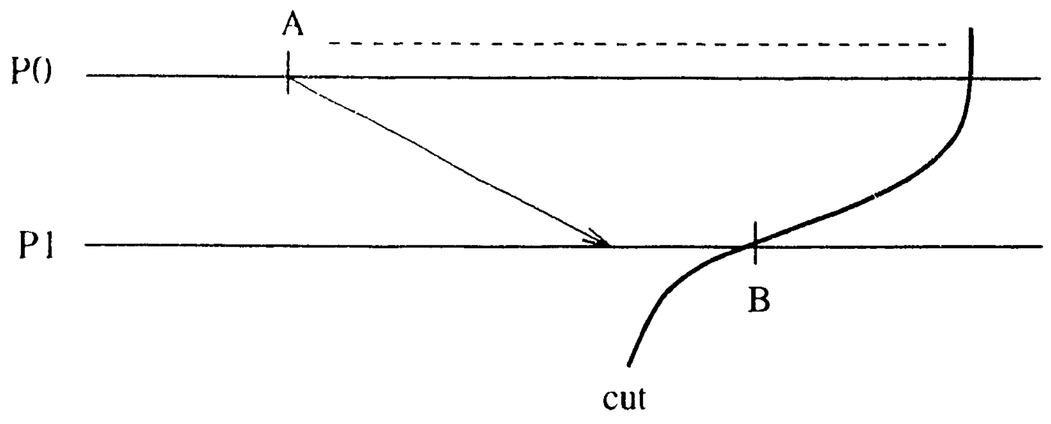


Figure 4.9: A is unaware of B's occurrence

re-execute under the assumption that all primitive predicates will occur in the same way. Breakpoint processes may then halt at respective locations to achieve causal, maximal, or immediate cuts.

An alternate solution which avoids rollback is implemented in CDB. It records relevant state information at the occurrence of every primitive event. For example, in figure 4.9, since there is no guarantee that B will occur, state information at point A will be recorded. Hence, when B does occur and the system is halted, we have meaningful data pertai. ng to the occurrence of A. Similarly, we can extend this solution by recording a trace from the primitive predicate, A, up to the halt point. This method does not allow us to obtain causal, maximal, or immediate cuts, but still provides meaningful information when the system does halt.

4.2.3 Detection/Halting Algorithm

The following algorithms are for detection and halting global predicates within CDB. Since global predicates are not guaranteed to occur and there is no checkpoint and rollback unit implemented, we do not provide causal, maximal, nor immediate forms of cuts. These are in fact considered as future extensions after the checkpoint and rollback unit has been built. Instead, we adopt the alternate solution of recording state information at the occurrence of the primitive predicates.

For every node within the breakpoint specification tree, a detection and halting task is created. This task manages all breakpoint information within the node. It communicates with other nodes to find the global predicate. There are two separate algorithms, called "operator" and "leaf". If the node is a non leaf node, then the task adopts the operator algorithm. If it is a leaf node, then the leaf algorithm is used. Since global predicates are constructed at the *debugging site*, all operator algorithms are expected to execute at this site. But, since primitive predicates are particular to processes, the leaf algorithms execute on the same sites as the processes. Communication between operator and leaf algorithms is through message passing.

Operator algorithm

```
loop
    if (child is leaf node) then
        receive message from child
    else (child is non-leaf node)
        sleep until waken by child
    evaluate operator
    if (NOT ROOT NODE) then
        if (Operator Satisfied) then
            record timestamp of child
            store timestamp and event type for parent (operator) node
            wake parent
            sleep until waken and notified by parent
            if (Signal == Continue) then
                if (child is leaf node) then
                    send Continue to child
                else (child is non-leaf node)
                    wake and notify child with Continue signal
            else (Signal != Continue)
                if (child is leaf node) then
                    send Break to child
                    else (child is non-leaf node)
                              wake and notify child with Break signal
          else (Operator Not Satisfied)
               if (child is leaf node) then
                    send Continue to child
               else (child is non-leaf node)
                    wake and notify child with Continue signal
     else (ROOT NODE)
          if (Operator Satified) then
               if (child is leaf node) then
                    send Break to child
                    send Break to all leaf nodes
               else (child is non-leaf node)
                    wake and notify child with Break signal
                    send Break to all leaf nodes
          else (Operator Not Satisfied)
               if (child is leaf node) then
                    send Continue to child
               else (child is non-leaf node)
                    wake and notify child with Continue signal
endloop
```

Leaf Algorithm

```
loop
    if (primitive predicate occurred) then
            Record timestamp
            Record state information of this predicate
            Send timestamp and event type to parent (operator) node
            Receive message from parent
            if (Message == Continue) then
                    continue task execution
                    return to main loop
            else (Message != Continue)
                    halt task execution
    if (Break signal received from ROOT) then
            halt task execution
endloop
```

4.3 Comparison with Haban and Weigel

Our solutions and that of Haban and Weigel [Haban-Weigel88] both view a distributed breakpoint as a hierarchical tree. But, our detection and halting algorithm differs from their algorithm. Essentially their detection and halting algorithm is as follows. Every process has a local debugger attached to it in which a copy of the distributed breakpoint is maintained. For every occurrence of a primitive predicate, it is marked within the local debugger and a message is broadcasted to all the local debuggers to notify of its occurrence. Eventually, the last primitive predicate will occur which will cause a halt signal to be broadcasted. Let us compare the message complexity of these two algorithms.

In both cases, let the distributed program consist of N tasks and the distributed breakpoint contain M primitive predicates.

Haban and Weigel

○ Occurrence of any one primitive predicate will send N-1 messages

The total number of messages is : $M(N-1)$

$\approx MN$

if $N \gg 1$

Our algorithm

○ Occurrence of any one primitive predicate will send 2 messages (node to central site and acknowledge back)

○ Occurrence of last primitive predicate will send N-1 messages

All primitive predicates, except for the last predicate will generate $2(M-1)$ messages.
The last primitive predicate will generate $(N-1)$ messages

Hence, the total number of messages is :$2(M-1)+(N-1)$

$\approx 2M+N$

if $M \gg 1$ and $N \gg 1$

Haban and Weigel's method generates many messages which may cause a large probe effect. But, since their system has a dedicated network for debugging messages only, their algorithm will induce virtually no probe effect. Our system, presently, consists of a single network where both application and debugging messages are sent. The probe effect in this system is an important factor which required us to design a detection and halting algorithm that has less message overhead.

This comparison is really not fair since their algorithm is decentralized whereas ours is centralized. The rationale for centralized system is derived from the fact that the user performing the debugging has to sit at one central site, unless computer supported cooperative debugging is done by many users working together

Chapter 5

Implementation Details

This chapter describes the major issues of the implementation and presents the design and functionality of each module. The implementation consists of the following four modules: vector clock, local debugger, breakpoint, and user interface. They have been organized in a layered format, see figure 5.1. There are two layers where the lower layer contains the vector clock and the local debugger modules and the higher layer contains the breakpoint and the user interface modules. The lower layer has been designed in a generic way so that distributed debuggers, like CDB, may be built using them. The higher layer contains more CDB specific modules.

5.1 Lower Layer

The lower re-usable layer consists of a vector clock server which maintains a distributed global clock and a local debugger which controls the execution of the local process. Together, these modules could support the implementation of breakpoints, stepping, monitoring, checkpoint and rollback, user interface, and other distributed processing concepts.

5.1.1 Vector Clock Server

The vector clock server is a multi-threaded server that is designed to be invoked on every machine. The executable is named *vect_server*.

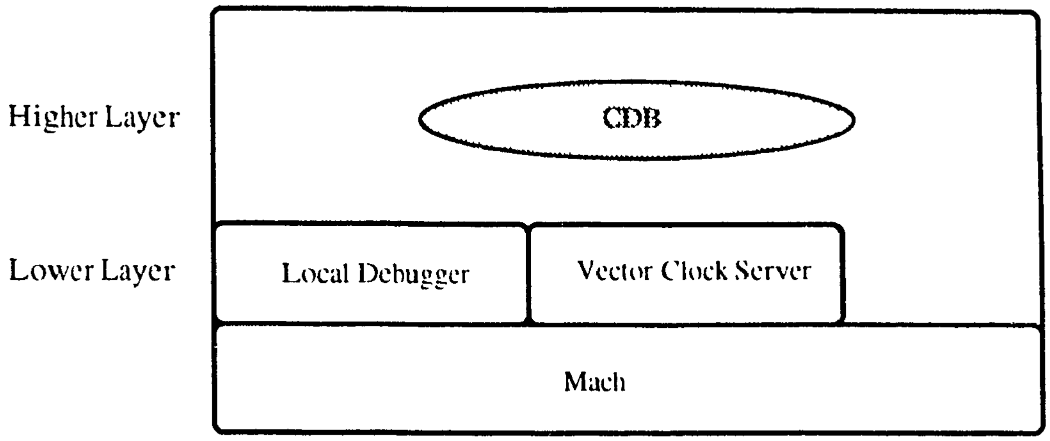


Figure 5.1: Layered Structure

Data Structures

The vector clock server maintains the following data structures:

Vector The vector is an array of five integers, one for each machine in our distributed system. In our case the five machines are: *jupiter*, *europa*, *carme*, *thebe*, and *ananke*.

Port There is only one port allocated to the server to which all requests are directed to. While invoking the vector clock server, the port name is required to be given as a parameter, i.e., vect_server < *port_name* >.

Operation of the Vector Clock Server

There are four types of requests that can be made to the vector clock server and they are: initialize, terminate, increment, and receive. Initialization causes all elements of the vector to be set to zero and termination causes the server to terminate. Increment causes appropriate element in the vector to be incremented and receive causes the current vector to be updated with the received vector. The algorithm used is as follows:

Vector Clock Server Algorithm

```
Receive next request from port
    If (request = initialize) then
        For i = 1 to 5
            vector[i] = 0


    If (request = terminate) then
        terminate server


    If (request = increment) then
        machine = current host machine
        If (machine = jupiter) then vector[1] = vector[1] + 1
        If (machine = europa) then vector[2] = vector[2] + 1
        If (machine = carme) then vector[3] = vector[3] + 1
        If (machine = thebe) then vector[4] = vector[4] + 1
        If (machine = ananke) then vector[5] = vector[5] + 1
        send vector to back requester


    If (request = receive) then
        machine = current host machine
        If (machine = jupiter) then vector[1] = vector[1] + 1
        If (machine = europa) then vector[2] = vector[2] + 1
        If (machine = carme) then vector[3] = vector[3] + 1
        If (machine = thebe) then vector[4] = vector[4] + 1
        If (machine = ananke) then vector[5] = vector[5] + 1
        For i = 1 to 5
            If (received_vector[i] > vector[i]) then
                    vector[i] = received_vector[i]
        send vector to back requester
```

Interface to the Vector Clock Server

All interactions with the server is done through Mach IPC. Hence, to properly communicate with the server, all messages must have the following Mach data structure:

```
struct msg_struct{
        msg_header_t h;
        msg_type_t t;
        long inline_data[6];
}
```

where they should be initialized to

```
h.msg_local_port = thread_reply();
h.msg_remote_port = port of sever.
h.msg_size = sizeof(struct msg_struct);
h.msg_id = 0x12345678;
h.msg_type = MSG_TYPE_NORMAL;
h.msg_simple = TRUE;

t.msg_type_name = MSG_TYPE_INTEGER_32;
t.msg_type_size = 32;
t.msg_type_number = 6;
t.msg_type_inline = TRUE;
t.msg_type_longform = FALSE;
t.msg_type_deallocate = FALSE;
```

The first element, inline_data[0], should be set to -1 for initialize request, -2 for terminate request, -3 for increment request, and -4 for receive request.

For initialize and terminate requests, a Mach "message send" is used to send the request. For increment and receive requests, a Mach "message rpc" is used to send the request and to receive the current vector clock values.

Determining Partial Order of Distributed Program

The vector clock server alone is not sufficient to implement the vector clock algorithm in section 3.1. In order to determine the partial ordering between events within a distributed program, the user's code must be augmented for interaction with the vector clock server, see figure 5.2. The vector clock server and the augmented user code will implement the vector clock algorithm.

To augment the user code, the user must first include the header file *debug.h* at the top of the code. This will allocate memory within the user's address space which will be used during interaction with the vector clock server. The header file will also redefine message send, receive, and rpc such that they will interact with the appropriate parts of the vector clock server. The user is also required to include an additional constant, *DEBUG_MSG*, within all message structures so that the vector clock values may be appended to outgoing messages. All message structures should be written as:

```
struct simp_msg_struct {
        msg_header_t h;
        DEBUG_MSG
        .
        .
        .
};
```

The user code is also required to be compiled with an additional library *ddebug*. Invoking the executable will result in the user's program **transparently** interacting with the vector clock servers.

Example of Vector Clock Server Interface

For the user program to interact with a vector clock server, we developed a utility program called *serverutil*. Essentially it is a program that sends either an initialize or terminate message to a specified server. Its semantics are

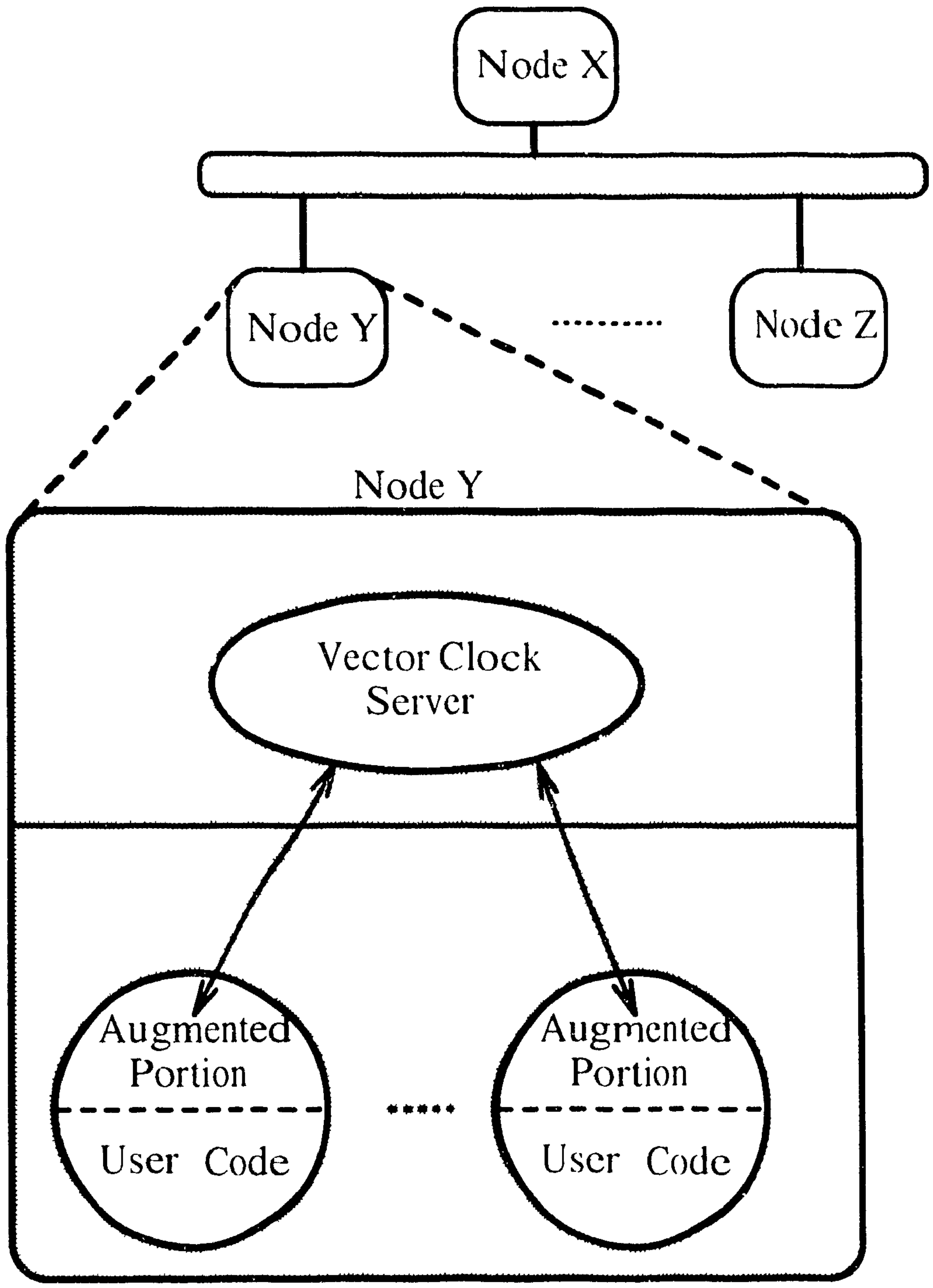


Figure 5.2: Vector Clock Server with augmented user code

servernutil < *INITTERMINATE* >< *nameofserverport* >

5.1.2 Local Debugger

The local debugger is based on GDB version 3.1. We had modified GDB so that all the input and output of GDB normally obtained from an interactive user are now done through Mach IPC. The executable is named *cdbgdb*

Data Structures

Port There is only one port allocated to the local debugger where all requests are directed to. Upon invoking the local debugger, the port name is required to be given as a parameter, i.e., cdbgdb < *port_name* >.

Operation of the Local Debugger

See figure 5.3. Essentially the operation of the local debugger is same as that of to GDB except that its requests are taken from incoming messages and its output is sent as an outgoing reply message. The overall algorithm is simple:

```
loop
    receive next message
    parse message to form a GDB command
    execute GDB command
    package GDB output into an outgoing message
    send outgoing message to requester
endloop
```

Interface to the Local Debugger

All requests sent to the local debugger have the following Mach structure.

```
struct simp msg snd struct{
        msg header t h,
```

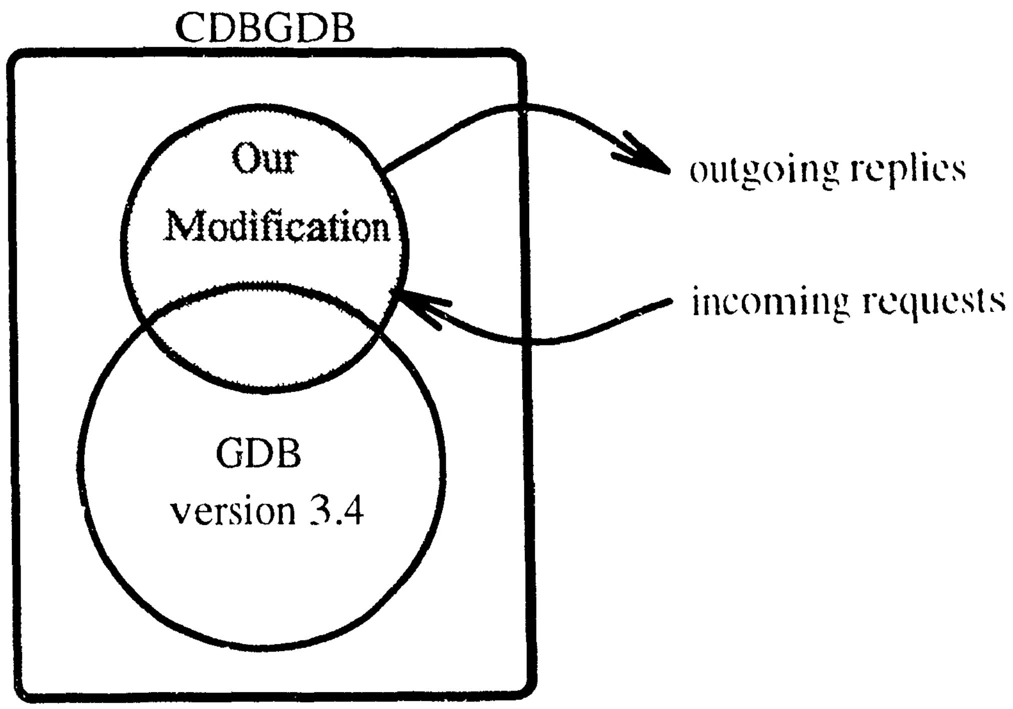


Figure 5.3: GDB with our modification

```
        msg_type_t t;
        char inline_data[128];
};
```

and should be initialized as.

```
h.msg_local_port = my_reply_port;
h.msg_remote_port = local_debugger_port;
h.msg_size = sizeof(struct simp_msg_snd_struct);
h.msg_id = 0x12345678;
h.msg_type = MSG_TYPE_NORMAL;
h.msg_simple = TRUE;

t.msg_type_name = MSG_TYPE_CHAR;
t.msg_type_size = 8;
t.msg_type_number = 128;
```

```
t.msg_type.inline = TRUE;
t.msg_type.longform = FALSE;
t.msg_type.deallocate = FALSE;
```

inline_data[128] is a string that contains any valid GDB command specified in [Stallman89].

The reply message from the local debugger is expected to be buffered in the following data structure:

```
struct simp_msg_rcv_struct{
        msg_header_t h;
        msg_type_t t;
        char inline_data[1024];
};
```

inline_data[1024] contains the output to corresponding request.

5.2 Higher Layer

The higher layer consists of two modules of breakpoint and user interface. Following the general design philosophy of CDB, we have organized these two modules in a modular format, see figure 5.1. Portions within these modules may be replaced so that new breakpoint or user interface related work may be rapidly supplemented. For example, detection and halting units may be replaced such that new detection and halting routines may be studied. This will encourage the re-usability of these modules in the future. Referring to figure 5.4, all units within the breakpoint module, specification, detection, and halting, have been implemented. Also, the first two modules, graphical representation of processes and I/O for breakpoint module, have been implemented for the user interface module.

5.2.1 Breakpoint Module

The breakpoint module is multi-threaded where it relies on both the vector clock and the local debuggers.

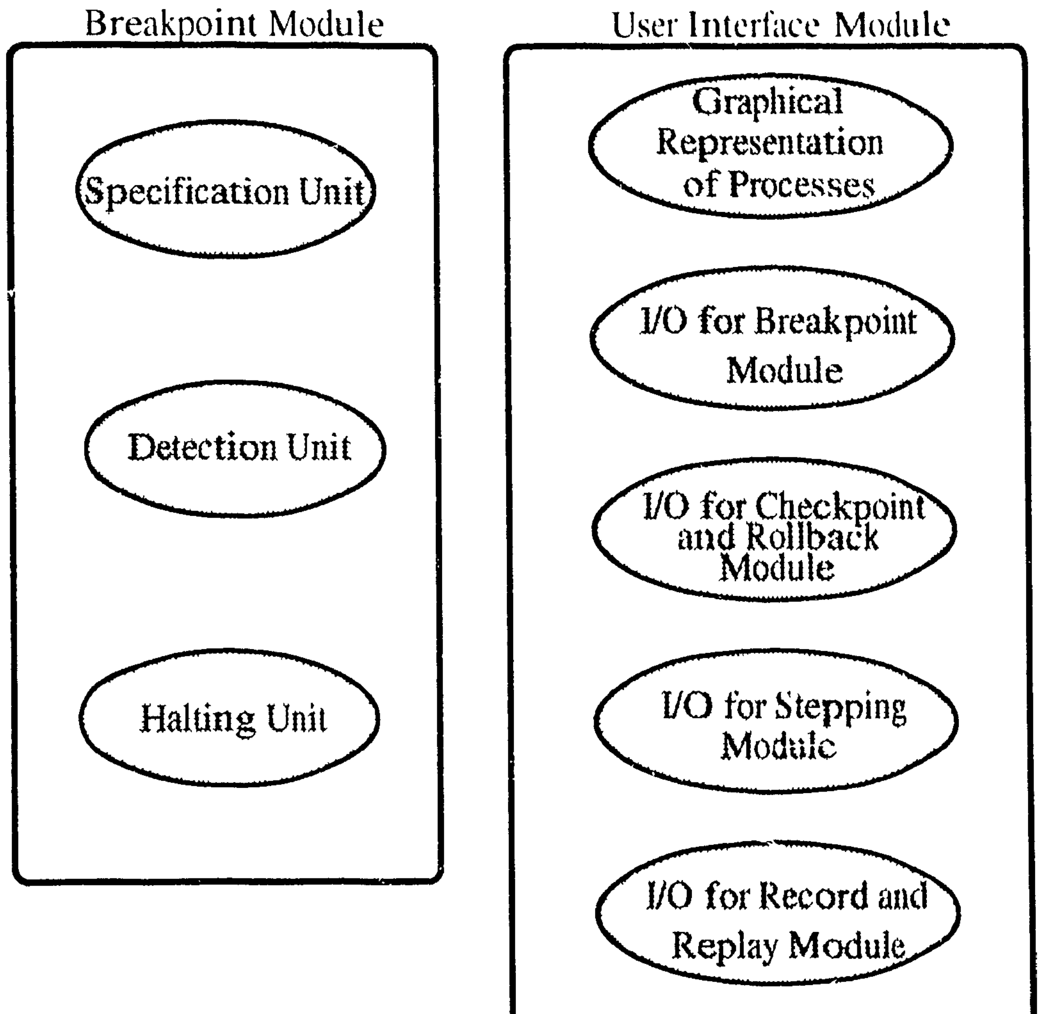

Breakpoint Module
Specification Unit
Detection Unit
Halting Unit
User Interface Module
Graphical Representation of Processes
I/O for Breakpoint Module
I/O for Checkpoint and Rollback Module
I/O for Stepping Module
I/O for Record and Replay Module
I/O for Monitor Module

Figure 5.1: Modular Design of Breakpoint and User Interface Modules

Data Structures

The breakpoint module maintains the following data structures:

Breakpoint Tree Given a PDL specification, the breakpoint module generates a data structure that represents the breakpoint. This data structure closely maps to the hierarchical structure of a PDL specification where it is built as a binary tree. Each node either represents an operator, such as → and &&, or a primitive predicate and has a vector clock value that will indicate the time of its occurrence. All leaf nodes maintain a linked list of state information of the given primitive predicate (i.e., when a primitive predicate occurs) and state information may be recorded in the linked list.

Trace During run-time, as the breakpoint module attempts to detect the given PDL specification, a data structure is dynamically generated and stored that reflects the events that occurred. For every event detected by this module, whether it be part of the PDL specification or not, a node is created and added to the data structure. This data structure will be the basis for a graphical representation of all events that occurred which is generated by the user interface module.

Operations of the Breakpoint Module

Basically, the breakpoint module performs two operations of breakpoint detection and building the trace data structure.

Breakpoint Detection Algorithm

The first step is to generate the breakpoint tree. This involves using the sub-algorithms, "operator" and "leaf", described in chapter 4. The algorithm is as follows:

```
Read PDL Specification
Generate breakpoint tree
For every node within the tree do
        If (node = operator) then
        Create a thread that will maintain the node and
            follow the operator algorithm in section 1.2.3
    endif
    If (node = leaf) then
        Create a thread that will maintain the node and
            follow the leaf algorithm in section 1.2.3.

        Invoke a local debugger which will assist leaf algorithm.
endfor
Wait for signal to commence detection of breakpoint.
```

Building Trace Data Structure:

The second operation is the generation of the trace data structure which proceeds as follows:

```
Set up all local debuggers to detect any primitive predicate
Loop
    Receive message upon occurrence of any primitive predicate
    Create node representing the primitive predicate and add
        it to the trace data structure.
    Mark node if primitive predicate is part of PDL specification.
endloop
```

Interface to the Breakpoint Module

The input is a PDL specification. As of now, this specification is in the form of an ASCII file and not in true form. Each line in the file represents a node within the PDL hierarchy and has a number assigned to it. These numbers determine the placement of each node, i.e., which nodes are parents of which other nodes. For example, referring to figure 5.5, this simple PDL specification would look like

1 function_name_type <<

```
1 node_type operator
1 indicator root
;
11 function_name_type &&
11 node_type operator
11 indicator left
;
12 function_name_type DUMMYg
12 node_type leaf
12 indicator right
12 machinename carme
12 directory /mitech/home/grad/christy/Machexamples/IPC
12 taskname cquad
12 parameters 4
12 state list
;
111 function_name_type <<
111 node_type operator
111 indicator left
;
112 function_name_type DUMMYc
112 node_type leaf
112 indicator right
112 machinename ananke
112 directory /mitech/home/grad/christy/Machexamples/IPC
112 taskname aquad
112 parameters 2
112 state list
;
1111 function_name_type DUMMYa
```

```
1111 node_type leaf
1111 indicator left
1111 machinename jupiter
1111 directory /mitech/home/grad/christy/Machexamples/IPC
1111 taskname jquad
1111 parameters 1
1111 state list
:
1112 function_name_type DUMMYb
1112 node_type leaf
1112 indicator right
1112 machinename thebe
1112 directory /mitech/home/grad/christy/Machexamples/IPC
1112 taskname tquad
1112 parameters 3
1112 state list
```

The output of this module are the two data structures namely the breakpoint tree and the trace data structure. The breakpoint tree will contain timestamps of all nodes as well as captured state information of primitive predicates that became true. The trace data structure contains a *history* of the user program execution

5.2.2 User Interface

The user interface module provides the user with a means of interacting with all CDB modules. Presently, referring to figure 5.4, this module contains only those units for breakpoint I/O and graphical representation of processes. The design of this module permits its re-usability when checkpoint and rollback, stepping, record and replay and monitoring are integrated together.

This module is based on the X windows system with the athena widget set. Essentially, we have designed the X portion of the user interface to be a frame where all

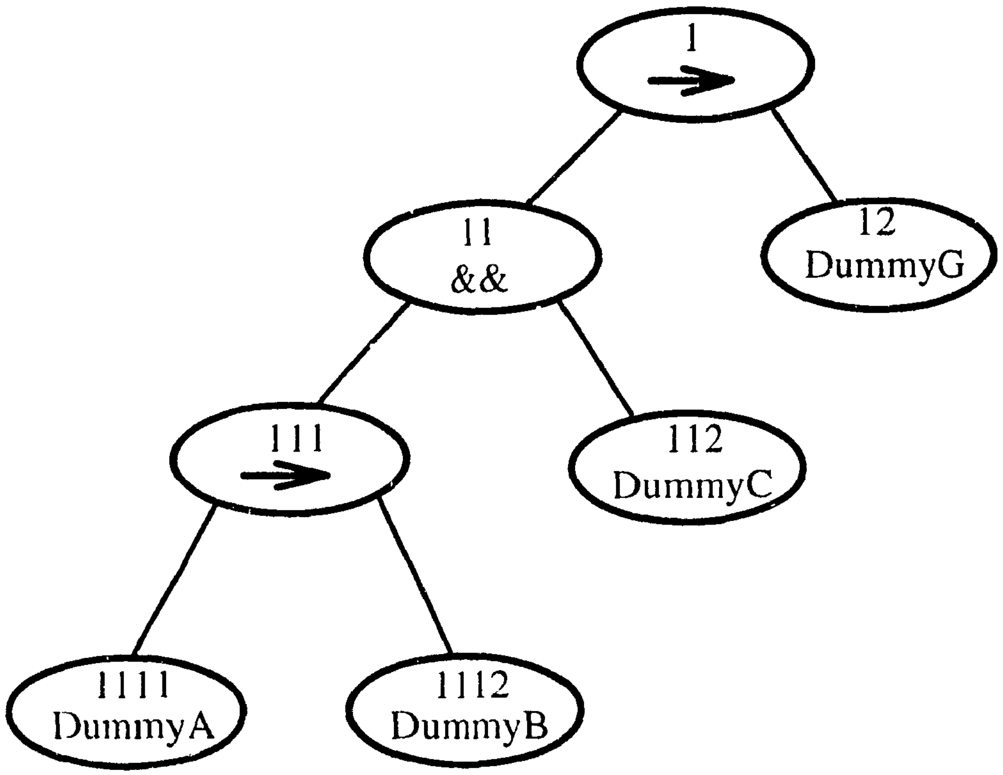


Figure 5.5: PDL Specification

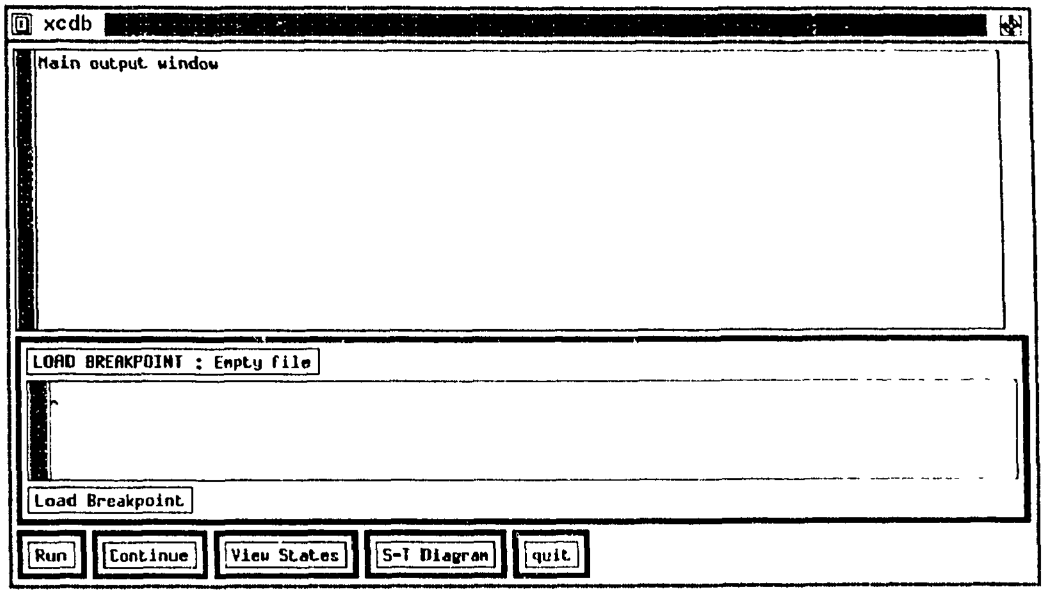


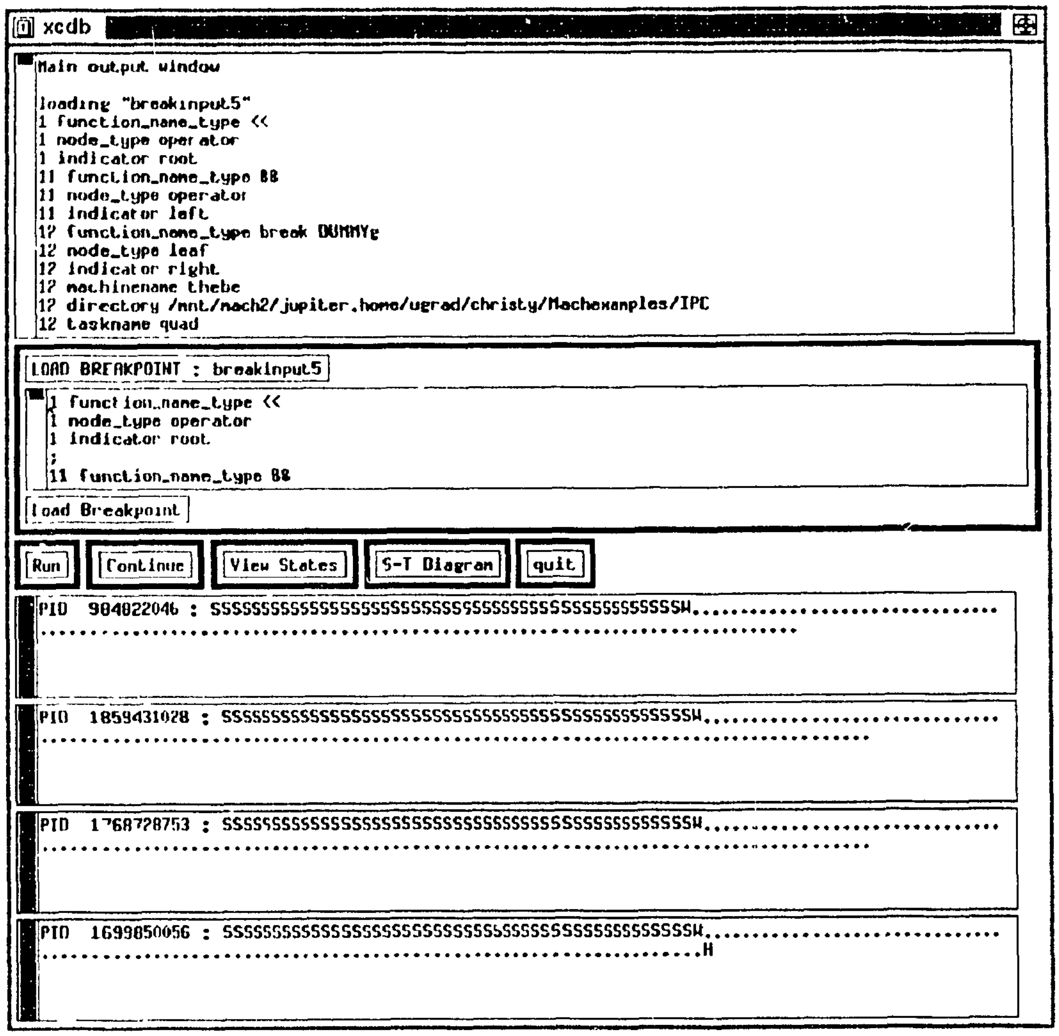
Figure 5.6: Main xcdb Window

the current and future buttons, windows, and sub-windows will be properly managed New buttons, windows, and sub-windows need only be specified within this frame.

Operation of the User Interface

Upon invoking CDB, the main window will appear as in figure 5.6, which contains two sub-windows and a set of buttons. The first window is called the *Main output window*. This contains all error, warning, or acknowledge messages displayed for the user. The second sub-window is the *current breakpoint display area* and contains a button for loading a breakpoint ASCII file. The display area displays the current breakpoint that CDB is looking for. The set of buttons are used to execute the program, continue the execution, view state information, view S T diagram, or to quit CDB. They are explained below.

The *Load Breakpoint* button reads a PDL specification file and feeds it to the breakpoint module. As an example we consider the PDL specification file to be that of section 5.2.1. Upon selecting this breakpoint, the PDL specification is displayed within current breakpoint display area. See figure 5.7 Since this PDL specification



```
Main output window

loading "breakinput5"
1 function_name_type <<
1 node_type operator
1 indicator root
11 function_name_type 88
11 node_type operator
11 indicator left
12 function_name_type break DUMMYg
12 node_type leaf
12 indicator right
12 machinename thebe
12 directory /mnt/mach2/jupiter.home/ugrad/christy/Machexamples/IPC
12 taskname quad
```

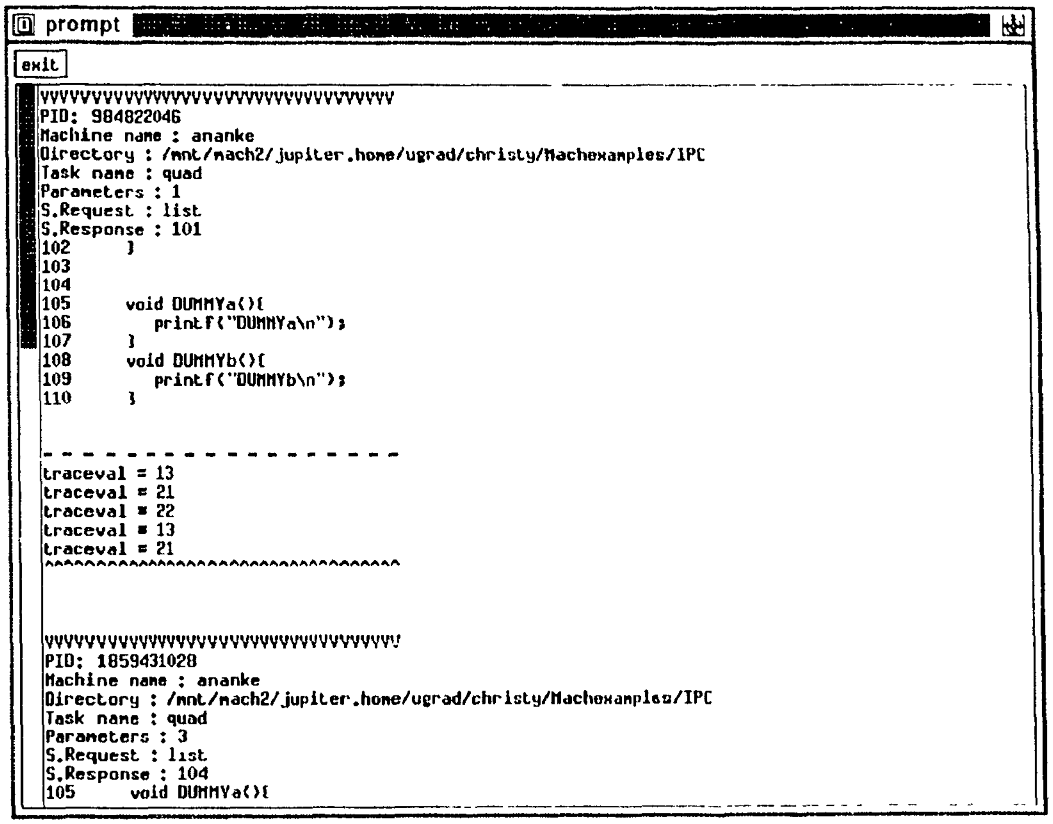
```
LOAD BREAKPOINT : breakinput5
1 function_name_type <<
1 node_type operator
1 indicator root
;
11 function_name_type 88
```

```
PID  984022046 : SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSW.............................
..........................................................................

PID  1859431028 : SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSW.............................
.................................................................................

PID  1768728753 : SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSW.............................
.................................................................................

PID  1699850056 : SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSW.............................
.................................................................H
```

Figure 5.7: Detection of Breakpoint

incorporates four processes, four *activity* sub-windows are created which indicate the activity of each process. Within these sub-windows, the S denotes that a process is setting up to detect the given primitive predicates and the W indicates that the process is ready and waiting to start detection.

The *Run* button simply raises a signal which will notify the local debuggers to commence execution of the program and to start detecting the current breakpoint. Referring to figure 5.7, the activity sub-windows contains a series of "." (dots) which are generated dynamically to indicate the process is in execution state. Eventually, the breakpoint will be reached where one of the processes will detect the final primitive



```
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
PID: 984822046
Machine name : ananke
Directory : /mnt/mach2/jupiter.home/ugrad/christy/Machexamples/IPC
Task name : quad
Parameters : 1
S.Request : list
S.Response : 101
102       }
103
104
105       void DUMMYa(){
106          printf("DUMMYa\n");
107       }
108       void DUMMYb(){
109          printf("DUMMYb\n");
110       }

- - - - - - - - - - - - - - - - - -
traceval = 13
traceval = 21
traceval = 22
traceval = 13
traceval = 21
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^



VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
PID: 1859431028
Machine name : ananke
Directory : /mnt/mach2/jupiter.home/ugrad/christy/Machexamples/IPC
Task name : quad
Parameters : 3
S.Request : list
S.Response : 104
105       void DUMMYa(){
```

Figure 5.8: State Sub Window

predicate and will result in the other processes to halt. This is denoted by an *H*.

The *View States* button pops up a textual sub-window in which the states of primitive predicates, from the PDL specification, are displayed, see figure 5.8. Basically the breakpoint tree, from the breakpoint module, is traversed and the state information retained in leaf nodes are printed. Also, a list of events from the primitive predicate location up until the halt point is displayed. Presently, referring back to figure 5.8, this list is displayed as *traceval = xx* where *xx* has a numerical value that represents a particular event. Future work would require that these numbers be parsed and the corresponding event names be displayed.

The *S-T Diagram* button pops up a sub-window where the S-T diagram is graphically drawn, see figure 5.9. The trace data structure generated by the breakpoint module is traversed where each node is mapped to the window. The dotted line rep

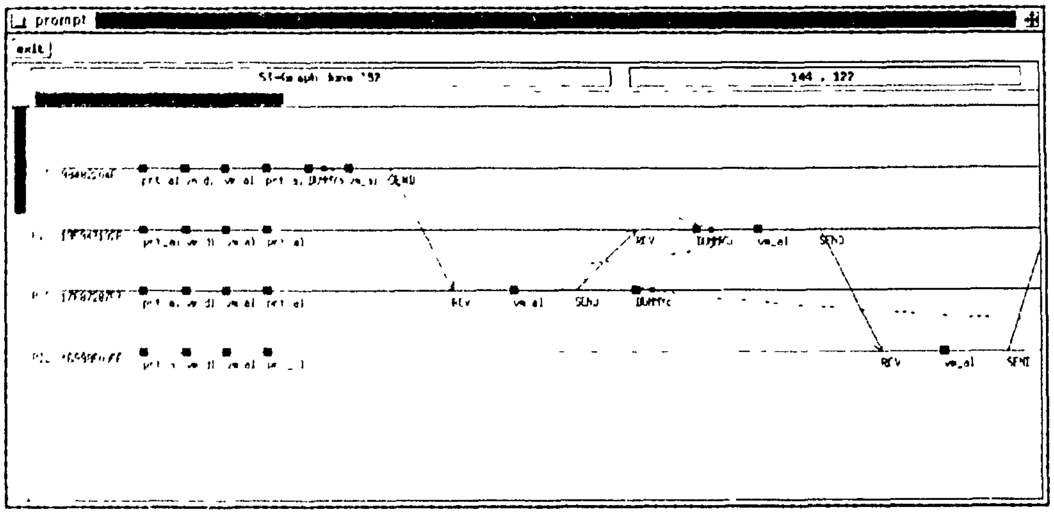


Figure 5.9: S-T Diagram Sub-Window

resents the breakpoint specification cut where a solid line, not shown, represents the actual system halt cut.

The *Continue* button raises a signal that indicates, to all the local debuggers, that the program may continue execution. The activity sub-windows will thus continue to dynamically create a series of "." (dots).

The *Quit* button terminates all CDB modules.

5.3 Code Structure

This section describes the contents of each file which is useful for those who continue implementation of other subsystems of CDB.

Vector Clock Server

debug.h is expected to be included in the users code. It contains all the declarations needed in the user code to include the vector clock.

debug5.c contains all the operations that the user code will transparently invoke in order to communicate with the vector clock server.

libddbug5.a is the library built from debug5.c.

vect_server.c is the code for the vector clock server. The executable is *vect server*

serverutil.c is the code for the utility program that may be used to initialize or terminate any server. The executable is *serverutil*

Local Debugger

main.c is originally from GDB 3.4. All modifications done to the user interface are found here.

cdbgdb is the executable for the local debugger.

Breakpoint Module

externals.h contains all the external declarations

global.h specifies all the global variables.

local.h contains all the common declarations that all files have.

struct.h contains all data structure declarations.

breakpoint.c contains operations to build breakpoint tree, spawn detection threads, and invoke local debuggers.

gdb_leaf.c is the code that implements the leaf algorithm from section 4.2.3

gdb_operator.c is the code that implements the operator algorithm from section 4.2.3

init.c contains all initialization routines.

invoker.c invokes local debugger on remote site.

message_oper.c contains all message operation routines

port_oper.c contains all port operation routines.

utils.c contains all miscellaneous support routines.

User Interface Module

ST_INT.h contains all declarations for the ST diagram

xcdb.h contains header information for xcdb.c.

xexternals.h contains all the X window external declarations

xglobal.h specifies all X window global variables.

xlocal.h contains all the common declarations that all X window files have

xcdb.c is code for generating the user interface

xutils.c contains all X window miscellaneous support routines

Chapter 6

Summary and Future Work

6.1 Summary

CDB, Concordia Distributed deBugger, is an ongoing project that started in 1988 and is expected to continue for some more years. Its main goal is to develop a set of debugging tools catered for experimental research in distributed debugging in the Mach environment. These tools perform monitoring, record and replay, breakpoint, checkpoint and rollback, stepping, and user interface. The research support that these tools provide will be to experiment with new algorithms or compare algorithms by taking measurements on certain variables. Of these tools, our work concentrated on breakpoint and user interface and we have developed two CDB modules, breakpoint and user interface, and two reusable generic support modules vector clock and local debugger as a contribution to the CDB project.

One of the problems in distributed breakpoints is its specification. Yet another difficulty of distributed breakpoints stems from the potentially large scalability of a distributed program. As the program increases in size, a breakpoint specification requires a larger amount of information and also provides a larger amount of output data for inspection by the user. It would be unrealistic to assume that the user is capable of knowing his/her needs exactly so that a meaningful breakpoint may be specified. Thus, interactive use is inevitable. For this reason we presented the PDL breakpoint specification language and it allows the user to view the program from various levels of abstraction and construct a breakpoint based on such abstracted views. As the user gains more information about the program, other viewpoints may be taken such that a finer breakpoint may be constructed. Along with PDL, we presented halting and detection algorithms that closely resemble the work of Haban and Weigel [Haban-Weigel88] but their algorithms are less efficient compared to ours for the distributed computing environment we have chosen.

The user interface is another area that we addressed. Basically, there are two different types of user interfaces: textual and graphical. Both interfaces have their advantages and disadvantages but neither by itself seems to be ideal for distributed debuggers. We feel that a combination of both interfaces will suit the needs of the debugger. Essentially, our user interface module adopted the graphical Space-Time diagram technique for displaying the message level interaction between multiple processes. It displays all the events that occurred from all processes as well as the interactions between them. Our textual representation is reserved for more intricate details of a program.

Global time is an important issue in debugging distributed systems since it determines the temporal relations between process events. The vector clock algorithm reported by [Fidge] and [Haban-Weigel88] is one form of keeping track of global time where it determines the partial order of the system. We found that the algorithm tends to take a large amount of space and have little support for dynamically allocated tasks. For this reason we adopted the algorithm reported by [Haban-Weigel88] and implemented a vector clock module that reduces the space requirements and permits dynamic task creation.

Local debuggers are necessary modules in any distributed debugging system since they control the execution of a given sequential task. We have implemented a local debugger module which is essentially a conventional debugger, GDB, with a Mach interface. Processes run within the context of this local debugger which can be invoked and controlled remotely.

In the scope of our work, we can conclude that this thesis presents the development of a flexible and integrated software system, based on breakpoints, for debugging distributed programs running under Mach. Our development resulted in a set of reusable modules that are integratable into the CDB project and they are flexible enough to support new algorithms and the development of new modules.

6.2 Future Work

The following are suggestions for future work on the current implementation of the breakpoint and user interface modules.

Breakpoint module

- Presently, the breakpoint module can only accept PDL specifications that include the happens before and simultaneous relation. Further extensions to the module are needed to incorporate alternation, conjunction, and the filtering facility.
- Presently, the breakpoint module only detects one breakpoint for every run of the program. Extension to this module should allow multiple breakpoint specification and detection.
- Presently, when a breakpoint is detected, the system is halted as soon as possible forming an arbitrary cut past the point where the breakpoint was specified. Other form of cuts, such as causal and maximal, should also be implemented given a breakpoint specification.
- Presently, it is assumed that the distributed program being debugged does not dynamically spawn tasks. Hence, breakpoint module has no means to bind to spawned tasks. Future versions of this module should incorporate this.

User Interface

- Presently the breakpoint module expects the user interface to input a breakpoint specification file from the user. A more meaningful interface would be to provide a means to graphically enter the breakpoint.

- The S-T diagram simply displays all primitive predicates that occurred with all message interaction. More data should be made available so that the user may view finer aspects of the program. For example, extra mouse driven windows should display the code space of each individual program.

CDB-The Project

Since CDB is an ongoing project, there is much work left to be done. Our modules are the second set of modules completed for CDB which provided a solid platform for future modules to be added on. Presently, aside from our modules, the record and replay module has also been completed and integration of these two sets of modules should be considered in the near future. Other modules such a checkpoint and roll back, monitoring, and stepping remain to be built in the future. Also, algorithms implemented in each module should be compared with other related work and evaluated based on key metrics such as message complexity, space complexity, or other suitable metrics.

Bibliography

[MACH 1] Adadis Tevanian, Jr., Richard F. Rashid, "MACH: A Basis for Future UNIX Development",*Department of Computer Science, Carnegie Mellon University*

[MACH 2] A. Silberschatz, J. Peterson, P. Galvin, "Operating System Concepts, 3rd ed.", *published by Addison-Wesley*, Third Edition, pp 597-629, 1992.

[MSV91] Sigurd Meldal, Sriram Sankar, and James Vera, "Exploiting Locality in Maintaining Potential Causality", *ACM*, 1991.

[LRK90] H F. Li, T Radhakrishnan, and V. Krawczuk, "A Toolkit for Debugging Distributed Programs", *Concordia University*, December, 1990.

[Fowler Zwaen90] Jerry Fowler and Willy Zwaenepoel, "Causal Distributed Breakpoint", *IEEE 10th International Conference on Distributed Computing Systems*, Rice University, Houston, Texas, 1990.

[Stallman89] Richard M. Stallman, "GDB Manual", The GNU Source-Level Debugger, Third Edition, Version 3.4, Oct 1989.

[Caswell Black89] Deborah Caswell and David Black, "Implementing a Mach Debugger For Multi-threaded Applications", *USENIX*, Winter ,1990.

[McDo-Helm89] Charles E. McDowell and David P.Helmbold, "Debugging Concurrent Programs", *Computing Surveys*, Vol. 21, No. 4, pp. 593-622, December 1989.

[SBN88] David Socha, Mary L. Baily, and David Notkin, "Voyeur: Graphical views of Parallel Programs", *Proceedings of Workshop on Parallel and Distributed Debugging*, ACM, pp. 206-215, 1988

[Haban-Weigel88] . Dieter Haban and Wolfgang Weigel, "Global Events and Global Breakpoints in Distributed Systems", *21st Hawaii International Conference of System Sciences*, University of Kaiserslautern, Kaiserslautern, West Germany, 1988.

[Miller-Choi88-1] Barton P. Miller and Jong-Deok Choi, "A Mechanism for Efficient Debugging of Parallel Programs", *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, University of Wisconsin, May 5-6, 1988.

[Miller-Choi88-2] Barton P. Miller and Jong-Deok Choi, "Breakpoints and Halting in Distributed Programs", *IEEE 8th International Conference on Distributed Computing Systems*, University of Wisconsin, 1988.

[Hough-Cuny87] A.A. Hough and J. Cuny, "Belvedere: Prototype of a pattern oriented debugger for highly parallel computation", *Proceedings of the International Conference on Parallel Processing*, Penn. State University, pp. 735-738, 1987.

[Leblanc-Crummey87] Thomas J. Leblanc and John M. Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, Vol. 36, No. 4, pp. 471-482, April 1987.

[JLSU87] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger, 'Monitoring Distributed Systems", *acm Transactions on Computer Systems*, Vol. 5, No. 2, pp. 121-150, May 1987.

[BDV86] Fabrizio Baiardi, Nicoletta De Francesco, and Giglioal Vaglini, "Development of a Debugger for a Concurrent Language", *IEEE Transactions on Software Engineering*, Vol. 12, No. 4, pp. 547-553, April 1986

[HHK85] P.K. Harter Jr., D.M Heimbigner and R.King, "IDD: an interactive distributed debugger", *Proceedings of the 5th International Conference on Distributed Computing Systems*, IEEE, pp498-506, 1985.

[GGK84] Hector Garcia Molina, Frank Germano Jr., Walter H. Kohler, "Debugging a Distributed Computing System", *IEEE Transaction on Software Engineering*, Vol. 10, No. 2, pp. 210-219, March 1984.

[Bates Wileden83] Peter Bates, and Jack C. Wileden, "An Approach to High-Level Debugging of Distributed Systems", *Proceedings of ACM SIGSOFT / SIGPLAN Symposium on High-Level Debugging*, pp. 107-111, March 1983.

[Bates Wileden82] Peter Bates, and Jack C. Wileden, "EDL: A Basis For Distributed System Debugging Tools", *Proceedings of the 15th International Conference on System Sciences*, University of Massachusetts, Amherst, Massachusetts, 1982.

[Lamport78] L Lamport, "Time, Clocks, and Ordering of Events in Distributed System", *Communications of the ACM*, Vol 21(7), pp. 558-565, July 1978.

[Fidge] C. J. Fidge, "Partial Orders for Parallel Debugging", *Australian National University*, Canberra, ACT, Australia.

[H.F. Li.] Personal communication with Dr. H.F. Li, Professor of Computer Science, *Concordia University*.

[ALAIN] Personal communication with Alain Sarraf, Computer Science Graduate Student, *Concordia University*.