A DISTRIBUTED DEBUGGER BASED ON DETERMINISTIC REPLAY

Joanna Sienkiewicz

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

January 1996

ISBN 0-612-10894-5

Abstract

A Distributed Debugger Based on Deterministic Replay

Joanna Sienkiewicz

The growing popularity of distributed computing creates a demand for tools to support development of distributed software. A debugger is a basic software development tool needed by all programmers. Development of a distributed program requires a "good" debugging tool since distributed programs are more difficult to develop and debug than sequential programs. The difficulty in writing distributed software systems results from non-determinism inherent in program behaviour, lack of precise global states, complex patterns of interaction, large state space, probe effect, and communication limitations. Additional complexity arises due to race conditions and violation of implicit synchronization requirements.

In this thesis we describe DDB (Distributed DeBugger) a debugger for distributed and multithreaded programs running under Mach operating systems based on the well known technique of instant replay. The DDB debugger addresses the issue of non-determinism in distributed programs by dividing the debugging activities into two phases - record and replay. In the record phase the distributed program is executed with the debugging mechanism present which collects and stores all the needed information for "replay". This information is used in two ways. First, it is used to create a high level view of the program execution in the form of time-process diagram, then, in the replay phase, this information is used to ensure that the execution takes the same path as the previous one. In the replay phase the programmer can obtain low level details of program execution by using source code level debugging techniques offered by the popular GDB debugger that is attached to each process of the distributed program.

The present prototype implementation of DDB can be enhanced in the future with tools providing functionalities like checkpoint and rollback, global breakpoint or various graphical interpretations of the execution, or sophisticated stepping semantics.

This thesis is dedicated

to my husband
Robert

Acknowledgements

First, I would like to express my sincere gratitude to Dr.T Radhakrishnan, my thesis supervisor, for his guidance and valuable insight throughout this research. His wisdom and infinite patience have given me great support during preparation of this thesis.

Next, I would like to thank Dr. H.F. Li for his consultations, advice and suggestions which were always of great help.

I am very grateful to the members of the Analysts Pool of the Computer Science Department, particularly to Stan Swiercz, who helped us in our daily struggles with the computing equipment. Without their help many things in this project would not be possible. Thank you all again.

I also want to acknowledge the Fonds pour la formation de Chercheurs et l'Aide à la Recherche for providing financial support during my graduate studies.

The time at university would not pass so quickly without having great friends like Judit Barki, Alicja Celer, Maria Janto, Ilya Umansky, Sai Rani Vallurpalli. I would like to thank them for the good times we had together.

Next, I would like to acknowledge the international network of my family members and friends for their unending support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed computing has a potential to satisfy the growing demand for computational power. By dividing a computational task into several parts and executing each part on a different computer simultaneously, the time needed to complete the overall task can be significantly reduced. To take full advantage of the potential of a distributed system, good quality software is required. Debugging tools are essential to develop quality software. The techniques used in distributed debugging are in most cases based on sequential debugging but there have been also new methodologies and approaches proposed.

This thesis describes a distributed debugger that works under Mach operating system. The Distributed deBugger (DDB) is a conceptual continuation of the previous work done in the area of distributed debugging at the Computer Science Department at Concordia University.

1.1 Distributed Computing

A distributed program can be described as a set of sequential programs (processes) executing simultaneously on different computers and "cooperating" with each other to achieve the common computational goal. Distributed computation consists of instructions local to each process as well as instructions involving communication between processes.

Distributed programs are more difficult to develop, test, and debug than sequential programs. The most relevant problems that the programmer must face when attempting to create distributed software programs are as follows:

- Non-determinism inherent in progra n behavior
- Multiple threads of control
- Lack of precise global states
- Complex patterns of interactions
- Large state space
- Communication limitations
- Probe effect

Non-determinism

Unlike sequential programs, due to several random factors, concurrent execution can take different paths each time the program is executed. The interaction between the processes taking part in the computation, and race conditions present in the system cause this non-determinism. Non-determinism makes debugging difficult because the programmer can never be sure that the bug occuring in one execution will ever repeat itself, giving another opportunity for its examination. This problem is specially relevant for bugs that occur infrequently.

Multiple threads of control

The level of difficulty is greatly increased when there are more than one thread of control participating in the computation. Multithreaded programs are more difficult to understand and write. It is also more difficult to follow their execution path. The interaction between threads operating on shared resources creates potential for new types of errors that would be absent in sequential programs. These errors include race conditions and synchronization problems.

Lack of precise global states

It is one of the fundamental problems of distributed debugging. Global clock synchronization is also a classical research problem in distributed systems, but obtaining the accurately synchronized global clocks alone does not guarantee the accurate global state of the system. Because of different communication delays, various speeds, and and multiple machine states, collecting of the global information can proceed at different rates. Also it is difficult to effect "control changes" on the different processors at the same instant. For these reasons, only approximate global state can be obtained, like one consisting of local states of all machines plus states of the communication channels within certain time.

Complex patterns of interaction

Distributed programming introduces new types of bugs that have source in interaction among multiple asynchronous processes. The difficult errors in a concurrent programs often result from improper synchronization among the processes or from race conditions. The interaction between processes can be data dependent and order (sequence of messages) dependent. Bugs of this type are often hard to reproduce, because they depend not only on input data, but also on the relative timing of interactions among processes.

Large state space

In most cases the data needed for analysis of execution of a concurrent program creates a large state space. It often includes the machine state on each processor and a record of interactions among different processors, creating the problem of manipulating large quantities of state data information to be viewed while debugging.

Communication limitations

Unpredictable communication delays and limited communication bandwidth may make some of the debugging techniques (central manipulation of information) impractical.

Probe effect

Probe effect is defined as the undesired side-effects that the debugging activity can have on program behavior. This occurs when the debugging actions affect timing, and therefore the ordering, of events. The probe effect can hide otherwise present errors, or introduce new errors that do not occur when the program runs without a debugger. This problem is related to non-determinism inherent in distributed programs. Elimination of the probe effect is not possible, but care should be taken to make it as minimal as possible.

From the above discussion it is evident that concurrent programs require as good or better debugging tools than sequential programs. The techniques developed for sequential debugging are not adequate for the new requirements of the concurrent program debugging, and new solutions must be introduced.

1.2 Distributed Debugging

Although distributed debugging is an young discipline, its development has been quite dynamic due to the high demand for distributed debugging tools. Over the years several different methodologies and approaches to distributed debugging have been developed, often by extending the concepts from use sequential debugging to distributed debugging.

The additional complexity of distributed programs have created a need for methodologies that would allow for simplifying the debugging process. There are three types of such methodologies:

- Top-down debugging
- Bottom-up debugging
- Two-phase debugging

Top-down approach

Top-down debugging is often applied for fully developed systems where all processes are already implemented. When faced with debugging of a large distributed program, the programmer will first consider the behavior of a chosen cluster of processes and try to identify the erroneous modules. After the bad module is identified, the search starts for the faulty processes, and so on until the bug is located.

Bottom-up approach

In the bottom-up debugging each process is first debugged in isolation. After the correctness of single processes is ensured, the processes are systematically merged and the emphasis is put on examining the interaction between the processes. This procedure is repeated until all the clusters are formed and debugged. This methodology is very useful for a new systems being developed since it allows for step by step development and testing of all components.

Two-phase debugging

Two-phase debugging divides the debugging activities into monitoring and debugging. In the first phase the software continues to execute until an error is observed. During the execution some information about the program behavior is collected and stored. In the second phase the erroneous process is tested in the environment that is identical to that in which the error originally occured. The contexts of the original execution are reconstructed with use of data collected during the monitoring phase. The process can be replayed many times until the error is found. This approach is very good since it introduces probe effect only in the monitoring phase. The probe effect in the second phase is not relevant since the execution is controlled so it has to follow the original path.

The methodologies described above are not independent of each other and one often appears as a part of another. For example, bottom-up debugging can be used during the second phase of two-phase debugging.

In the attempt to make the distributed debugging more effective, new approaches have been proposed. The following describes some of the more promising approaches to distributed debugging that appeared in the technical literature.

- Database approach
- Behavioral approach
- Artificial-intelligence approach

Database approach

In the database approach the debugging process is viewed as performing queries on a database containing program information (system specification,source code) and execution information. The program information is loaded into the database when the debugging starts. During the program execution the debugging information is forwarded to the debugging system, which updates the database. The bug is found by creating queries to the database. The answers to the queries are supposed to provide clues to the bug. One of the advantages to this approach is the availability of distributed database systems and the possibility of use of the database system's query interface as the debugger's user interface. Database systems also allow for rich set of queries on the values and relationships among the data.

Behavioral approach
The behavioral approach views debugging as a process of performing comparisons of the execution behavior to the expected behavior. The expected behavior is defined by some formal specification. The debugger monitors the execution, extracts the actual execution events and compares them to the expected behavior. The programmer is notified if there is a discrepancy between the expected and the actual behavior. The current research concentrates on the development of debugging tools that would help the programmer in understanding the program behavior. This can be achieved by creating different views of the execution and viewing the execution at different levels of abstraction.

Artificial-intelligence approach
The artificial-intelligence approach uses techniques developed for AI to find errors in distributed programs. One method is to use an expert system. The expertise of an experienced programmer is captured in the knowledge base in the form of rules. This knowledge would be available to all programmers when the expert system acts as an intelligent assistant for reasoning and developing fault hypothesis. Although the methodology is very promising there are no debuggers so far that use the AI techniques directly to debug distributed programs.

1.3 History of distributed debugging research at Concordia University

The distributed debugging research at Concordia University has been conducted for several years through doctoral and masters projects. During these years, different problems of distributed debugging have been addressed and some prototype debugger implementations have been attempted:

- **Ph.D Thesis 1988: Krishnarao Venkatesh** proposed a formalism for classifying different types of global states of a distributed system and developing new message efficient algorithms for detecting consistent global states.

- **Master Thesis 1988: Chris Passier** provided experimental evaluations of two rollback and recovery (R&R) algorithms. He had implemented a R&R kernel which can serve as a basis for building the R&R algorithms.

- **Master Thesis 1988: Ioakim Hamamtzogulu** 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.

- **Master Thesis 1989: Minh Dang Bao** proposed a methodology and tools for distributed debugging in a two stage process. The first stage is a top-down debugging where a given synchronization specification is compared with the actual synchronization behavior until the error is detected. The second stage is a bottom-up debugging where examination of internal states is performed to locate a bug. He also implemented a prototype debugger to exercise his methodology.

The four master projects listed below have contributed to creation of CDB (Concordia Distributed deBugger). The implementation of the CDB debugger, called **xcdb** runs under the *X-Window system* and can be used for debugging of distributed programs written for a network of SUN 3/50 running the Mach operating system. The debugger provides two facilities : breakpoint detection and halting, and checkpoint creation and rollback.

- **Master Thesis 1993: Victor Krawczuk** proposed and implemented a record and replay module which addressed the problem of non-determinism in distributed programs. Only partial implementation was achieved.

- **Master Thesis 1992: Honna Segel** based her research in the area of monitoring. Developed a new logic for expressing safety properties using distributed predicates. She also proposed algorithms for detecting the violation of these safety properties.

- **Master Thesis 1992: Christy Yep** proposed and implemented module for specification and detection of distributed breakpoints. He proposed a specification language which enables the creation of hierarchical breakpoints ranging from fine to coarse granularity.

- **Master Thesis 1993: Alain Sarraf** created a checkpoint and rollback mechanism. When an error is detected, the system will rollback the execution to a state or a checkpoint that is considered error free. From this point, interactive debugging can be used to locate the bug.

1.4 What is DDB

This thesis work contributes to the distributed debugging research at Concordia University by creating a new distributed debugger based on deterministic replay. Unlike CBD this is a simple debugger without the rollback component.

The software requirements of this debugger have been extensively researched. The research conducted took the form of a survey. By examination of the functional properties of concurrent debuggers that have been proposed, designed or implemented, we have been able to develop a functional model of a debugger that can satisfy the very basic needs of debugging. It is simple enough to make the complete prototype implementation within the scope of a master's thesis work. Its design, implementation and testing took about a year.

DDB (Distributed deBugger) is a debugging tool for distributed and multithreaded programs running under Mach operating system. The debugger is to assist the programmer in locating errors in concurrent programs by providing relevant information about execution of the program. DDB addresses the issue of non-determinism inherent in distributed systems by taking the two-phase debugging approach. The activities in the two phases are as follows :

- **Monitoring Phase:** The distributed program is executed and the relevant information for the execution control is collected in history files. The information is extracted and stored in non-intrusive manner. The high level view of the execution path in the form of time-process diagram assists programmer in identifying erroneous execution, if any.

- **Replay Phase:** The program is re-executed in such a way that the faulty execution path is repeated, so the bug can manifest itself. A source level debugger, operating within a single machine (modified GDB) can be used to find the precise location of the bug.

The DDB debugger combines sequential and concurrent debugging techniques by providing the deterministic replay, giving a high level view of the program (presented through time-process diagrams), and at the same time allowing for source level debugging. The graphical user interface makes the debugger easy to use.

1.5 Thesis outline

The remaining part of this thesis describes different aspects related to the design and implementation of the DDB debugger.

Chapter two presents a survey of distributed and parallel debuggers that were designed and implemented during the last five years. The first part gives a classification of the debuggers according to the functionalities provided. This is followed by detailed descriptions of sample debuggers presented in the literature belonging to each group in the classification. The last part gives the functional properties of about thirty debuggers in the form of tables.

The next chapter describes concepts related to deterministic replay in distributed environments. The sources of non-determinism in distributed systems in general, and in Mach operating system in particular are identified. Then the application of deterministic replay to distributed debugging is examined; different debugging techniques are evaluated according to their applicability in distributed environments, and the possible improvements in them with introduction of the deterministic replay facility are discussed. This is followed by the description of functional needs of a "minimal" concurrent debugger. The last section presents several debuggers that have been based on deterministic replay.

Chapter four describes the method used in DDB for capturing the non-deterministic behavior of distributed programs. The presentation of the general approach to developing the debugger opens the chapter. This is followed by description of recorded information in the *record* phase and the ways of enforcing required order of events in *replay* phase.

Chapter five gives a detailed description of the design and implementation of the DDB debugger. The programming environment of DDB is described, followed by the description of the layered structure of the system. Next, comes the presentaion of system architecture, and description of the system modules. The implementation concepts are described at the end.

The following chapter presents a sample debugging session with DDB. The User Interface is presented in detail by following each step of the debugging session of a program with injected bugs. This walk-through illustrates how software bugs can be identified with the use of the DDB debugger. The appendix contains source code of the distributed program used for this demonstration.

Chapter seven presents summary of the work described in the thesis.

Chapter 2

Survey of Distributed and Parallel Debuggers

This chapter presents a survey of parallel and distributed debuggers that were described in literature in the recent years, from 1985 to 1994.

The first section gives a discussion of the needs of the concurrent programmers in the area of debugging tools, and the classes of available debuggers. The next three sections describe respectively: integration of sequential and concurrent debugging techniques, high-level event based debuggers, and extensions of traditional debuggers. In each case the overall characteristic of the group is followed by detailed descriptions of some interesting debuggers belonging to the group. The next chapter gives summary tables, containing concisely presented characteristics of about thirty interesting concurrent debuggers.

The selection of the debuggers to be presented in this survey was influenced by several factors. First of them was the time of the publication - I was trying to collect descriptions of debuggers that appeared in the literature in the last five years in order to have good view of the recent research in this field. Another criterion was the availability of the material, although I was able to collect most of the material that seemed to be promising, some of the papers were not available. The criterion in choosing between available descriptions from the last five years was the completeness of the debugger and/or debugging facilities provided by the debugger.

From the above, it should be clear that this survey does not have the ambition to be complete in the number of debuggers included nor the details of the description of the included debuggers. It is rather an attempt to identify some tendencies present in the distributed debugging area.

For obvious reasons, I could not experiment with many of the debuggers described, so the information included in this survey is mostly based on the publications that I was able to obtain. Although I was always trying to clarify all doubtful points with the authors of these debuggers, I was often not able to contact them. It means that data included in this survey is as accurate as my understanding of the descriptions of the debuggers in the papers. So, if any inaccuracy is spotted by anyone, please, do not hesitate to inform me.

I would also like to take this opportunity to express my deep gratitude for everybody who has helped me in searching for the literature, and kindly answered my numerous questions.

2.1 Requirements of Concurrent Debugging

There is no doubt that every programmer will want to use a debugger at some point during software development. The question is: what debugger will s/he need. In the recent years there have been much discussion in the concurrent computing community about the problem of defining the required capabilities of the debugging tools. The features of the "dream debugger" defined by Charlie McDowell [32] during the Supercomputing Debugging Workshop '91 include the following:

- Fast conditional breakpoints and fast memory watch points.
- A memory map in a user friendly format.
- 2D and 3D plots of multidimensional data
- Dynamic display of data (i.e. continuously updated).
- A memory reference trace of shared variables.
- A trace of message passing activity.
- Multiprocess event detection.
- Source level debugging of optimized/parallelized code.

- Algorithm level debugging (e.g. better abstraction to higher levels).

- Deterministic replay and reverse execution.

Other features not mentioned above are in the category required for software systems in general, not only specific to concurrent debugging. These include, in the first place, the intuitive and familiar user interface that would make the debugger easy to use for the first casual user and give easy access to complex features [31]. A good interface can significantly increase the usability of the debugger, when even very sophisticated features hidden from the user behind unfriendly user interface will not be very helpful.

Another important requirement is the efficiency and speed of the debugger. Even when the probe effect is ignored, a very slow debugger can discourage some users, specially when long programs or large number processes are involved.

The speed and efficiency are related to another important issue - the scalability of the debugging tools. Can the debugger effectively be used with programs consisting of hundreds of parallel processes ? In some cases the debugger design is geared towards certain types of architectures and cannot be effectively used with others.

The above requirements for concurrent debuggers are answered by different debuggers in various degrees. Some of the debuggers are trying to accommodate all possible needs of the user, others concentrate on solving few specific problems. There are systems that have been built with one specific architecture in mind, and others that place high among their goals portability to other platforms. This survey is presenting different debuggers from the functional point of view. We will show what services the debuggers provide and how they can be used in locating bugs, rather than considering different debuggers' architectures or debugging methods used. According to this criterion, the available debuggers have been divided into following categories :

- **Integrated approach** - the sequential and concurrent debugging techniques are combined in one debugging tool.

- **High-level event based approach** - the debugger is trying to create abstract models of program behavior.

- **Extensions of sequential debuggers** - a sequential debugger is attached to each process of a concurrent program to provide source level debugging.

The types of debuggers excluded from the survey are the ones based on static analysis and the performance monitors.

2.2 Integration of Sequential and Concurrent Debugging Tools

The debuggers in this category are designed to accommodate most needs of concurrent debugging. They combine the top-down and bottom-up approaches, and address the problem of non-determinism inherent in concurrent problems. This is achieved by using the traditional debugging techniques and concurrent debugging specific techniques combined in one tool.

The traditional techniques include cyclic debugging, when the erroneous program is repeatedly executed and in each execution more information is gathered that leads to discovery to the error (simple output statements can be sometimes added to the program code to achieve it). Another technique is breakpoint insertion. When breakpoints are added to the program at interesting points in the code and the execution is suspended, the programmer is allowed to examine the system state. Stepping can also be used to observe the program behavior in greater detail.

The technique more specific to concurrent debugging is top-down interactive debugging, when the programmer considers different views of the program. Starting with the high-level view of the program's behavior, the programmer can refine that viewpoint based on the information collected during successive replayed executions, until the desired level of detail is reached. This technique is specially useful for parallel programs, because they tend to produce large volumes of data that are difficult to deal with when presented in an unstructured manner.

The applicability of the above techniques to concurrent debugging depends on solving the problem of non-determinism. This is achieved by providing the replay facility, that makes repeated executions of the program in deterministic manner possible. One of the commonly used techniques is the Instant Replay, proposed by Thomas J. LeBlanc and John M. Mellor-Crummey in [30], where the relative order of accesses to shared data is recorded and used later to reproduce the same behavior. Another method, used in IGOR [14], is based on supporting reversible execution of a single process by periodically saving execution states using virtual memory support.

The approach of combining the traditional and parallel debugging techniques is promising because it allows for using methods that the programmers learned when debugging sequential program, and in the same time address the problems resulting from complexity of concurrent programs. Unfortunately there are not many debuggers that take this approach, and often they have been proposed rather than implemented. The remainder of this section presents three debuggers that represent the integrated approach described above.

2.2.1 Debugger for Amoeba

This section will describe the debugger for Amoeba, the capability based distributed operating system developed at the Centre for Mathematics and Computer Science (CWI) and the Free University (VU) in Amsterdam. The programs in Amoeba are collections of *clusters*, which communicate with their environment through messages. A cluster is an address space with a collection of tasks. Tasks can communicate by sending messages, through shared memory, semaphores or condition variables (only tasks in the same cluster). Another way of communication is by use of signals. A cluster receiving a signal from a task is interrupted and the signal is passed to the exception handling process.

Functionality

The debugger for Amoeba is an event-based debugger. It sees the user program by the stream of events the program generates. The events correspond to such program activities as sending and receiving of messages, creating tasks, reaching a breakpoint, or generating an exception. Each event is represented by a 4-tuple including event type, identity of the task generating the event, the capability of the cluster containing the task, and the arguments of the event. The use of *filters* and *recognizers* helps with processing of the data associated with events. The filters are applied to reduce the volume of data by eliminating the events that are of no interest to the user, so only the relevant events are passed to the subsequent stages of the debugger. The recognizers are used to search for specified sequences of events. Filters and recognizers use *patterns* (combinations of 4-tuples) to select events. A match between a pattern and an event is found when each field in the event is matched by the corresponding field in the pattern.

The event based approach is augmented with use of other facilities, often found in sequential debuggers. These include the ability to examine or alter the contents of memory, obtain a stack trace of procedure calls, set breakpoints, and to refer to the program on the source-level basis.

The debugger also provides execution replay facility based on the Instant Replay method. To recreate the progra n's execution, only the order of relevant events occurring in each cluster is recorded, without the data associated with the events. The recording of events is done locally to each cluster to capture the order of the cluster internal events. The interaction with the external environment is done only through messages, so the relevant events that need to be recorded and later controlled concerning this interaction are the message events.

The recreating of the exact execution is made difficult by the existence of the cluster signaling mechanism. Because the signals are asynchronous evens that can occur at any time, they cannot be reproduced without extensive kernel support.

The checkpoints of all clusters are taken during the recorded execution. This allows the user to roll the execution of the program back to any place, what might be useful in case of long programs.

Evaluation

The above described debugger for Amoeba integrates source level debugging (access to memory, breakpoints) with methods specific for distributed debuggers (execution replay, checkpoints, creation of event-based view of the program execution). This rich set of functional capabilities is presented to the user by a window-based user interface that allows for examination and control of individual clusters and tasks, as well as dealing with the target program as a whole.

The latest description of the debugger presents the prototype version, where many problems, like handling of signals or timeouts, are still to be addressed.

2.2.2 Panorama

The Panorama debugger was developed at the University of California, San Diego. The main goal of this software system is to provide an integrated debugging environment for parallel programs, that could be easily ported to a variety of MIMD message-passing multicomputer architectures and also extended to implement new debugging techniques. The debugger runs on top of existing multicomputer debuggers (like IPD or ndb) that are usually provided by the hardware vendors. Panorama uses the data provided by the base debuggers to create various graphical views of the debugged program. The user is also able to access the base debugger directly through a textual window.

Program views in Panorama

The execution of the parallel program is presented by Panorama using different graphical views derived from the information provided by the base debuggers. The views can be chosen from the default set or created by the user. The following views are provided by Panorama:

- message queues of each processor
- time line of events on the processors
- map of data

The above views can be modified by the programmer.

The new views are created with the use of Tcl(Tool Command Language) and Tk. Tcl is a software library developed for customization of programming tools like debuggers and editors. Tk is a package of X11 user interface objects that can be controlled with the use of Tcl script. The programmer can create a new view of the execution by writing Tcl/Tk script and placing its name in a file that contains names of all active features.

Portability

One of the main design principles of the Panorama debugger is the ease of porting the system to new parallel platforms. The use of the base debuggers for handling the low level interactions with the program makes portability achievable. The information about the commands used in each of the base debuggers is stored in the "platform file " that Panorama uses to translate the generic commands into the base debuggers commands. This file is also used to interpret the responses from the debuggers. In some cases there is no straightforward correspondence between the generic and the debugger specific commands. To handle this problem, platform specific functions are called to perform the required action. These special functions are written as scripts in Tcl and stored in a special script file.

This simple arrangement makes it possible to adapt the debugger to a new parallel computer by writing the platform file and the script file, and placing them in the Panorama directory. There is no need to recompile the debugger since the information from the files is read at run-time.

Status

The future enhancements to the Panorama debugger include creating new views to augment the default set, exploring ways of integrating Panorama with other programming tools, and improving communication with the base debuggers. At present, the debugger has versions working with IPD, and ndb base debuggers. There are plans of porting the debugger to M-5 and Intel Paragon.

Evaluation

The approach taken by Panorama is very appealing since it allows for integration of many debugging tools as well as using the same set of tools on different platforms. This eliminates the need to learn new techniques for each machine.

The possibility of extending the debugger is also very attractive, provided that it would be not too difficult to write the code needed for definition of the new program view. Some tools to assist the users in defining new views would be very appreciated.

2.2.3 PPUTT

The PPUTT toolkit was developed by the Parallel Program Understanding Tools and Techniques group at the University of Rochester. It is intended to address the problems related to debugging and analysis of parallel programs running on large scale, shared memory multiprocessors, such as the BBN Butterfly Parallel processor. The toolkit contains an interactive debugger, a graphical execution browser, performance analysis packages, and a programmable interface.

Methodology

The methodology applied by the PPUTT debugger extends the techniques of the sequential debugging and performance analysis to parallel programs analysis, to create a toollkit, that addresses the problems of traditional debugging, as well as the issues introduced by parallel program execution. Below we describe the main characteristics of this methodology along with the tools that implement it:

1. **Repeatable analysis** gives possibility of re-executing the same program in order to gain more information. This is achieved by dividing the program analysis into two phases. In the first phase the debugger collects data from the program execution that is used in the off-line analysis of the program in the second phase. The history of the execution is represented by the records of each process accesses to shared objects augmented with time stamps. The execution history browser, called *Maviola* gathers process-local histories and combines them into a single, global execution history.

2. **Top - down analysis** allows for both, viewing the program as whole and referring to single processes within the program or statements within the process. The top level execution view is represented by directed acyclic graphs (DAG) of process interactions. Nodes in the graph correspond to monitored events that took place during execution. Events within the process are linked by arcs denoting a temporal relation based on a local time scale. The interprocess communication is represented by arcs between events belonging to different processes. The graphs are created by *Maviola* based on the previously recorded execution history.

3. **Interactive and fine-grain analysis** allows the user to shift the focus of interest to display only the relevant information. The fine-grain analysis provides access to very detailed information about the debugged program. The program replay allows for examination of the execution at an arbitrary level of detail. Breakpoints can be used to stop the execution during replay, so the programmer can monitor processes, events or statements.

4. **Extensibility of the set of the possible analysis** allows for creating new, application specific analysis when such need arises. The programmable tools

included in the toolkit can be used to examine and manipulate the execution histories. For instance, the user can write Lisp code to traverse the execution graph built by *Maviola* to gather detailed, application specific performance statistics.

Evaluation

The integrated approach of this toolkit is very promising because it allows for addressing all problems related to debugging of a parallel program. The cyclic debugging techniques can be applied since the executions are repeatable and all the necessary information can be gathered during successive deterministic reexecutions. The high level views of the debugged program can be effectively used to examine the patterns of interprocess communications. The toolkit provides the user with all standard debugging and performance analysis techniques and, in addition, by means of programmable interface allows for extending the toolkit for application-specific analyses.

2.3 High-Level Event Based Debuggers

A large group of concurrent debuggers take the event based approach, which is appropriate since the complex behavior of parallel and distributed programs often requires more structured approach than the source level debugging can offer. The event based debuggers view program execution as a stream of events, and try to analyze the behavior of the program based on the information conveyed by these events.

The definition of an event differs from one system to another. In many cases the events represent interprocess communication operations. In shared memory systems the events will most probably represent accesses to shared variables, and in the message passing systems the message send and receive operations. But, in addition to the interprocess communication events, any relevant occurrence can be defined as some systems allow the user to specify the events that are of interest. In EBBA [1] the *primitive events* "represent the lowest observable level of system behavior or characterize some particular aspect of a system's activity ". Examples of these can be process operations (creation, suspension, resume) or file operations (open, create). The *primitive events* in EBBA form a basis for *view points*, which can be used to create high level models of the system behavior.

In addition to deciding on the type of events that are to be processed by the debugging system, there also must be some notation for representing the events or event patterns that the user is interested in. One of the languages created for this purpose is the *Event Description Language* (EDL), that provides a method of defining multiple levels of abstract events from primitive events generated by the program [2].

There are different methods of processing events that are identified during program execution. Some of the systems display the information as it is acquired, and some store it for future use. The Ariadne debugger, described below, represents a group of *post mortem* debuggers that gather information in one run, and then use different tools to process it. There are also different ways of using the gathered information that include browsing with some tools, creating visual images of the execution based on the generated events, and verifying assertions provided by the user.

The visualizing debuggers use different techniques to create good representation of the debugged program. The graphical images are sometimes accompanied by aural presentation. Some researchers suggest that pattern recognition using sound is superior to visual techniques, because the human ear is capable of processing tremendous amounts of data [17]. The important issues in creating useful visualization include providing multiple views, hierarchical presentation, application-specific visualization, and user-defined presentation [37]. The remainder of this section presents two event-based debuggers that take different approaches to presenting the debugging information to the programmer. The Ariadne debugger uses the information gathered during program execution to verify models of program behavior created by the programmer. The VISIT debugger provides multiple views of program execution, allowing this way for identification of the errors in the program. It is difficult to say which approach is better, maybe the two should be combined in one tool to give the best effect.

2.3.1 Ariadne

Ariadne is a *post mortem* debugger for massively parallel, MIMD message passing systems. The debugger takes the event-based behavioral abstraction approach; it supports the user in investigating global interprocess communication patterns.

Debugging method

The event based-behavioral abstraction allows the user to specify models of intended behaviour that are automatically compared to the actual program behaviour. The comparing of the models is performed on the traces produced by parallel programs. The sequences of events within the traces are identified by process ids. Currently, the debugger supports three types of primitive events: *Reads*, *Writes*, and *Phase Markers*. The *Reads* and *Writes* events denote interprocess communication, the Phase Markers denote the end of logical phases of computation. The traces are stored in *execution history graph*. The nodes in this graph represent events and the edges represent communication events. The debugger allows the users to view the execution history graph, but does not rely on visualization. It supports interactive, textual explorations of the graph.

A very important aspect of this type of system is the language used to define behavioral abstractions as patterns of events in logical time. The language used by Ariadne is simple. It employs a three level description of communication patterns:

- *Chains* represent the local views of communication. They are described by regular expressions. To obtain a match in an execution history graph, all events in the chain must occur exactly in the order specified in the pattern.

- *p-Chains* represent the concurrent execution of a chain by a set of processes. They are described by binding a chain to a process set. When a p-chain is matched against the behaviour, a copy of its chain is matched on each element of its process set.

- *pt-Chains* represent the logical, temporal composition of a set of p-chains. The matching process has two steps: first events matching p-chins are located in the graph and then the specified logical relations between those events are verified.

The simplicity of the language used to define the behavioral abstractions sometimes prevents the user from describing the intended behaviour precisely. To compensate for this inconvenience, the set of functions is provided that returns the characteristics of the obtained match.

Evaluation

The approach taken by Ariadne is very interesting because of the simplicity of the modeling language. Complex languages used in other debuggers of this type require very sophisticated matching algorithms on the debugger developers side, and considerable learning effort on the users side.

The limitations of this debugger include coarse granularity of the possible models description which results from the simplicity of the modeling language. The debugger also lacks any graphical utilities. Graphics could be used in creating the behavioral models as well as for illustration of the program behaviour.

2.3.2 Visit

VISIT is a parallel debugging environment developed at Siemens in Germany. It is used to debug programs written in parallel extensions of common Lisp and C++. The debugger takes a high level, event based approach to debugging.

Features

VISIT provides several different views of the debugged program. The views are based on events generated by the debugged program. The events are at source language level. They are grouped in classes (process, mailbox, critical section). In cases when the predefined events do not provide sufficient information, the user can define the events according to the actual debugging needs.
There are several visualizers available :

- **Process tree visualization.** The main visualization available in VISIT gives the overall view of the parallel program. It represents the process tree in two forms - the miniaturized overview of the entire tree, and an enlarged part of the tree. Each process in the tree is represented by a node with the process name and several icons denoting the process state. Clicking on an icon displays state information or triggers certain action. There can be several windows open for the same program, each displaying different part of the tree.

- **Sequence chart visualization.** This visualization provides a time line of each process and its events displayed as icons.

- **Critical section and mailbox visualization.** The view of the queue of waiting messages and/or processes is provided with the complete access history.
- **Processor load visualization.** This visualization is used to tune the load balancing algorithms.

The process tree visualizer also provides control over the executed program. For example, each process in the program can be stopped by clicking at the state icon. Then a new debugger window is open for the stopped process. The window can be closed by clicking again at the state icon. Other debugging tools, like tracers or steppers can be invoked in the some way.

The replay facility allows for recreating the behaviour of the program by the visualizers as many times as required without execution of the program. The visualizers use the trace information collected during the program run. This type of replay can be used for granularity analysis and tuning, since the full program state is not available.

Evaluation

The approach taken by VISIT is very interesting, since it allows someone to see both the global view of the parallel program and the individual process state. The view of the entire parallel program is given by the visualizers, the state of the particular process within the program can be examined with use of the debugging tools. The replay facility, although limited to fine grain events, is a very valuable feature in a concurrent debugger.

There are plans for porting the debugger to distributed systems.

2.4 Extensions of Traditional Debuggers

This section presents a category of debuggers that can be described as extensions of traditional debuggers. Most commercially available concurrent debuggers belong to that category. In the majority of cases, the debuggers are constructed by attaching a sequential debugger, like gdb or dbx to each process of the parallel program.

The factor deciding about the usability of the debuggers of this type is the handling on the problems specific to concurrent debugging. These problems include controlling of output coming from multiple processes at the same time, directing the sequential debugging commands to the desired set of processes, or managing large volumes of data present in many parallel programs.

Some problems remain unaddressed by this type of debugging tools. One of them is the probe effect that can make the traditional debugging ineffective against time dependent errors. Another important issue here is the low level that the debuggers operate on. For large programs consisting of many concurrently executing processes, it can be very difficult to understand what is happening at the interprocess level, when the behavior is presented at the instruction level.

The advantage of this class of debuggers is their simplicity and ease of use. The programmers often know the commands of the popular sequential debuggers so there is no need to learn new debugging commands when switching to a parallel environment. Another big advantage is that these debuggers exist in full implementation and are ready to be used as opposed to more sophisticated tools that have been proposed but not or only partially implemented.

The debuggers described below present different approaches to handling the problems resulting from concurrency present in the debugged programs. Two of them are commercial products, debuggers that are delivered to the user together with the computer system, the third one is a university project. All of them have been fully implemented and used by many programmers.

2.4.1 IPD

The IPD (Interactive Parallel Debugger) was developed by Intel corporation to provide debugging facilities for parallel programs running on iPSC parallel computers. The debugger is capable of debugging applications written in C language and Fortran.

Techniques

The IPD debugger is based on the traditional debugging techniques but it also has the necessary extensions for parallel debugging since the multiple processes present in a parallel program as well as the message passing between them create problems that cannot be solved by traditional methods. These extensions include handling of communication between the debugger and multiple processes, dealing with large volumes of debugging information, and providing access to the interprocess communication data.

One of these problems is the potential hitting of the breakpoint by many processes at the some time - if every process tried to print the breakpoint message on the debugger screen, the user would not have chance to see what happened because the screen would scroll too fast. The use of an asynchronous interface can solve the problem of notification, so the user can only be given the event information when requested. The form of this notification is also adapted to the new requirements.

Another problem is controlling the application I/O, so the programs input and output do not interfere with the debugger I/O. The solution chosen by the designers of the IPD is to take control over the application's terminal I/O, allowing terminal output before every user prompt, and both input and output when the user brings the requested process to the foreground.

Groups of processes in the parallel application can be addressed by listing ordered pairs (node_list:pid_list) that specify the debugging context. The debugging commands are applied only to processes included in the debugging context.

The problem of displaying voluminous information generated by some debugging commands is solved through data reduction. The debugger scans data to be displayed for each process in the debugging context to identify the groups of processes with the same data. This way only unique sets of data have to be displayed.

The communication between the multiple processes in the iPSC application depends on message passing. It is often important for the user debugging the program to know the state of the message passing network. The IPD provides commands allowing the user to display the contents of the system queues, and to get information about the messages that have arrived on the node but have not yet been received and receive requests that have been passed but have not been satisfied.

Future plans

Future plans for enhancing the debugger include optimization of the performance of the message tracing and exception tracing, including support for system programmers, and providing a graphical user interface.

Evaluation

The approach of extending the sequential techniques to parallel environment has the advantage of eliminating the need to learn new tools by the users. However, it is evident that these techniques can hardly be sufficient in debugging of concurrent programs. One major issue that is not addressed by IPD is the non-determinism inherent in concurrent programs. The IPD is a live-detection debugger; the errors present in one execution are not guaranteed to reappear in the next run, because the execution can take different path.

Another problem is lack on any graphical representation of the program execution. It can be very difficult for a programmer to interpret data coming from many processes displayed in textual form. Some interpretation tools could be very helpful.

The IPD debugger provides very restricted services but it could be used as a base debugger for more sophisticated debuggers like Panorama.

2.4.2 LPdbx

The LPdbx debugger was developed at Brown University. It provides debugging capabilities for programs written in C language, running on loosely coupled parallel processors like Armstrong, on which the debugger was implemented.

Features

The debugger offers most of the features present in the sequential debuggers plus message tracing capabilities.

The LPdbx has graphical user interface based on X Window widgets. The debugger window can display one process at a time, but the monitored process can be changed at any time by clicking the mouse on one of the pushbuttons. The window contains the source code of the selected process, the variables and structures, the command buttons and the select buttons. When the debugger is in the trace mode, the highlighted bar moves through the source code lines, just like in any sequential debugger.

The LPdbx allows for accessing variables at any location within the distributed program. It is also possible to examine the contents of a linkers list by tracing through the chain. Structures of complex data types are presented in graphical form.

The debugger allows for setting breakpoints in the application before the start of the execution as well as after the execution has started.

Evaluation

The big advantage of the LPdbx debugger is its very carefully designed user interface. It is very simple, and easy to understand and learn, what is not very common in applications of this type.

The techniques used in debugging are also very easy to comprehend by users familiar with sequential debuggers. The major problem with this debugger is lack of any global view of the distributed progıam. Since the debugger window is only capable of displaying the status of one process, the user is not able to observe the progress of the computation as a whole.

The second problem is lack of any facility addressing the problem of non-determinism in distributed programs. The execution of the debugged program cannot always be repeated in the same way, this can create problems when the bugs are appearing only in some executions.

2.4.3 UDB

UBD is a parallel debugger developed at Kendall Square Research, Waltham, Massachusetts. The debugger runs on the KSR1, which is a massively parallel supercomputer with shared memory. The debugger takes the bottom up approach to debugging, extending the traditional debugging techniques and adding new features.

Debugging facilities

The command set of UDB contains commands from the GDB debugger, the trace and assertion commands found in dbx, and some new features designed specially for parallel debugging. The new features include:

- full support for debugging Fortran programs
- command lists for signals
- arguments for user defined commands
- new control commands "if" and "for"
- command and session recording
- command file debugging

- vi mode for line editing

The UDB debugger provides extensive windowing facility. There are several types of windows. The source code window displays the source code of one selected thread or, like in the case of the shared source window, the breakpoints and the program counters for all the threads. The instruction windows can be created to display the disassembled program instructions and the current program counter. The shared instruction windows work the same as the shared source windows with minor limitations. There is a separate window containing program I/O. The user can also create prompt windows associated with different threads and create multiple command streams, each identified by thread number.

There are two types of breakpoints in UDB. The regular breakpoint suspends the entire program when a breakpoint in some thread is reached. The user can then switch between the threads to examine their state or change the variables. The continue command will resume all the suspended threads. This arrangement ensures that the program will execute normally when it is restarted after the breakpoint. The other type of breakpoint available in UDB is the synchronous breakpoint. When a thread hits a synchronous breakpoint, it stops, but if other barrier threads are running, the execution of the program continues. Only when all affected threads reach the breakpoint, the task is suspended and the control is returned to the programmer.

Evaluation

The UDB debugger has a command set that is based on popular sequential debuggers GDB and dbx. This fact will be considered as a big asset by everyone who is familiar with sequential debugging and steps into the world of concurrent computing, because the debugger will be easy to learn.

Another good point of this debugger is the windowing support that often makes manipulation of multiple threads simpler.

The deficiencies of UDB are common to the debuggers extending traditional techniques to parallel debugging: lack of some global view of the parallel program, and the fact that the debugger does not address the problem of non-determinism inherent in parallel computing.

2.5 Comparison tables

The following four tables collect information about the surveyed parallel and distributed debuggers. In order to include as much information as possible, the entries in the tables are abbreviated to one or two words. The detailed explanation of the notation precedes each table. The following entries are uniform across all of the tables:

- **n/s** - Information was not available
- **any** - Debugger does not require specific value
- **dbf** - Debugger for

2.5.1 General Information

This section presents general information about the debuggers. The data has been presented in two tables. The tables are different in type of information they provide and the order of presentation. The tables are followed by analysis.

General Information Table - Part I

Table 1 presented in this section gives general chracteristics of the debugging systems. The first column presents the reference numbers to the bibliography of sources of information for this survay that apperars at the end. The following columns give the operating system the debugger runs under, the hardware and the languages it supports. The entries are ordered alphabetically, according to the name of the debugger. The following notation is used in the table:

- **Ref** - Reference number
- **Operating System** the debugger runs under
- **Hardware** configuration the debugger works on
- **Languages** that the debugger supports

Debugger	*Ref*	*Operating System*	*Hardware*	*Languages*
dbf Agora	[15]	Agora	LAN	n/s
dbf Amoeba	[11]	Amoeba	n/s	any
Ariadne	[10]	n/s	MIMD message passing	any
bdh	[53], [54]	CSOS	Cray 3	C, Fortran
Bugnet	[49]	Unix	Network(Workstations)	C
CDB	[40]	CHORUS	486DX PC	any
CXdb	[46]	ConvexOS	Convex	any
EBBA	[1]	n/s	Network(Workstations)	any
HeNCE	[3]	Unix	Network(Workstations)	Fortran, C
IDD	[20]	Unix	Network (Sun)	C, Modula, Ada
IGOR	[14]	DUNE	Motorola 68000	C
Instant Replay	[30]	Chrysalis	BBN butterfly	any
IPD	[6]	NX2	iPSC/2 Intel	C, Fortran
ldb	[7]	UNICOS	Cray	fortran
LPdbx	[45]	Amstrong	Amstrong	C
MDB	[12],[13]	Xylem	Cedar	C, Fortran
dbf ML	[47]	any	any	ML
MpD	[38], [39]	Mach	8CE	C
MULTVISON	[19]	n/s	n/s	Multilisp
NodePrism	[43]	UNIX	CM-5	n/s
Panorama	[35], [34]	any	MIMD message passing	any
Parasight	[18]	Unix	Multimax	C
ParaGraph	[22]	any	any	any
PPUTT	[16]	Chrysalis/Elmwood	BBN Butterfly	Lynx
Recap	[36]	n/s	n/s	n/s
dbf RP3	[27]	Mach	RP3	any
UDB	[51]	OSF	KSR1	C, Fortran
VISIT	[23], [25]	n/s	EDS	Lisp, C++
Voyeur	[44]	n/s	n/s	any

Table 1: General Information - Part I

General Information Table - Part II

Table 2 presents the second part of the general information about the concurrent debuggers. The information in this table has been ordered differently than the information in the previous table. The first criterion for ordering is the completeness of the included debuggers. Accordingly, the debuggers are divided into groups corresponding to production, prototype, partial, and proposed implementations. Inside the groups defined by the completeness, the order follows the model of communication used in the computer system that the debugger operates under. First the debuggers for the message passing systems are listed, then the debuggers for the shared memory systems, followed by hybrid communication systems, and the debuggers for which the information about the communication scheme was not available. Inside these groups the alphabetical order is followed.
The following notation was used to present information in this table:

- **Status** - completeness of debugger implementation
  - *production* - production version available
  - *prototype* - not complete in house system
  - *partial* - prototype missing major features
  - *proposed* - no implementation exits
- **Communications** - model in the underlying architecture
  - *messages* - message passing
  - *shmem* - shared memory accessible by each processor
  - *hybrid* - shared memory and message passing can be used
  - *rpc* - remote procedure call
  - *rndzv* - Ada randezvous
- **Global clock** in the underlying architecture:
  - *assumed* - debugger assumes existence of global clock
  - *self-timed* - debugger provides its own clock

Debugger	*Status*	*Communications*	*Global Clock*
IPD	production	messages	assumed
LPdbx	production	messages	none
Paragraph	production	messages	self-timed
bdb	production	shmem	assumed
ldb	production	shmem	assumed
MDB	production	shmem	n/s
dbf RP3	production	shmem	assumed
UDB	production	shmem	assumed
CXdb	production	n/s	assumed
dbf Amoeba	prototype	messages	none
CDB	prototype	messages, shmem hybrid	self-timed
EBBA	prototype	messages	n/s
HeNCE	prototype	messages	self-timed
IGOR	prototype	messages	self-timed
Panorama	prototype	messages	none
dbf Agora	prototype	shmem	self-timed
Instant Replay	prototype	shmem	none
MpD	prototype	shmem	assumed
Parasight	prototype	shmem	none
PPUTT	prototype	shmem	self-timed
Voyeur	prototype	hybrid	assumed
VISIT	prototype	n/s	assumed
Ariadne	patrial	messages	n/s
Bugnet	partial	messages, rpc, rndzv	self-timed
IDD	partial	messages	none
MULTVISION	partial	shmem	n/s
Recap	proposed	hybrid	none
dbf ML	proposed	n/s	self-timed
NodePrism	n/s	messages	n/s

Table 2: General Information - Part II

Analysis

The fact that can be easily observed when looking at the first table is that while some of the debuggers run only on specific architectures, and support only specific programming languages, there are debuggers that are architecture independent, language independent, or both. These systems, like debugger for ML, Panorama, or ParaGraph, are likely to attract larger population of users due to their portability.

The debuggers in the second table are ordered according to the state of their implementation. The first place take the debuggers that are complete and ready to be used; their status is described as as "production". Next we placed the debuggers that exist in prototype implementation, where some features are missing. Finally, a few interesting debuggers that have been partially implemented or only proposed are also included. The completeness of the debugger is a very important characteristic since it determines the usability of the system. A programmer faced with a problem of locating some software bug will not be interested in sophisticated tools that exist only on paper. A simplest, but working tool would be better. For this reason, we have made the state of the implementation of the debugger one of the criteriums for selecting the debuggers to be presented in this survey.

One case of a "production" debugger is one that is sold with computer systems, like the UDB debugger, developed at Kendall Square Research to run on the KSR1 supercomputer. In such a case, the extent to which the debugger is being used is determined by the popularity of the computer system it was developed for, as well as, by the availability of other debugging tools. Another type of debuggers that are placed in "production" category are the debuggers developed in academic environments which have been fully implemented, tested, and used by members of the home academic community. Some of the systems, like LPdbx developed at Brown University, have been also made available for distribution, so there is a chance that they will be used outside their home community.

From the twenty nine debuggers included in this survey, only nine could be placed in the "production" category. In this group three debuggers (IPD, LPdbx, Paragraph) are supporting message passing systems, the remaining ones are supporting shared memory architectures.

The debuggers classified to be in the "prototype" stage of implementation, are the ones that do not yet have all the features included in the design, but are complete enough to be used and tested. The prototype debuggers are important because they are often under continuous development, and give hope for more complete implementations in the future. The progress of the work being done on the debugger can be best measured by the dates of the first and the subsequent publications, as well as the coverage of the publications. One of such promising debuggers is Panorama, described in the previous sections. Because there were three publications in the last two years concerning this debugger one might assume that the work continues, and that the fully implemented version is soon to come.

The majority of debuggers presented in this survey fall into the "prototype" category when the stage of implementation work is considered. We were trying to include debuggers that are still begin developed, but in some cases we abandoned this rule and included debuggers that have been in the "prototype" stage for a long time, when they presented an approach to concurrent debugging that should be (in our opinion) promoted. Such is the case of the PPUTT toolkit which, although not recently heard of, introduces approach of integration of debugging tools to tackle many debugging problems.

The "partial" status of implementation means that only small part of the features described in the design have been implemented. The "proposed" status determines situations when the debuggers exist only in design papers. In both cases, the debuggers were included in the survey to acknowledge their contribution to concurrent debugging by introducing new, interesting approaches.

2.5.2 Functional Characteristics

This section presents information concerning the functional characteristics of the surveyed debuggers. The first subsection contains all the data in tabular form. Then the analysis of the data is presented.

Functional Characteristics Table

The debuggers in table 3 are ordered according to the type of execution replay facility that they provide. At the top of the table are placed debuggers that provide complete replay facility. These are followed by two debuggers that provide replay facility resticted to communication events only. The debuggers without any replay capabilities are placed at the bottom of the table. The debuggers belonging to these groups

are ordered according to the type of breakpoint provided. First are placed debuggers with global breakpoint, then the debuggers with local, single event, statement, and no breakpoints follow. Inside these groups the entries are ordered alphabetically. Other information provided by this table is the effect of the breakpoint on the program execution, and the type of events recognized by the debugger.
The notation used in the table is as follows:

- **Replay** - replay facility provided by the debugger
  - *complete* - entire program state is available
  - *commun* - communication state can be deduced
  - *none* - no replay facility available
- **Types of Breakpoints** supported by the debugger
  - *global* - state breakpoint can be set
  - *local* - state breakpoint can be set
  - *stmt* - breakpoint can be set at a source code statement
  - *single* - event occurrence breakpoints
  - *mult* - events breakpoints on occurrence of combination of events
- **Breakpoint Effect** on the program
  - *program* - the entire program is stopped
  - *process* - one process is halted
  - *either* - either one process or the entire program is halted
- **Event Type** - supported by the system
  - *ipc* - explicit interprocess communication
  - *shmem* - shared memory references
  - *stmt* - each statement execution

Debugger	*Replay*	*Type of Breakpoints*	*Breakpoint Effect*	*Event Type*
dbf Agora	complete	local, single	process	shmem
dbf Amoeba	complete	local, stmt, mult	either	ipc
CDB	complete	local	program	none
Instant Replay	complete	local, stmt	program	shmem
Recap	complete	local, stmt	process	ipc, shmom
RP3	complete	single	program	stmt
Bugnet	complete	none	none	ipc
Panorama	complete	none	n/a	none
ParaGraph	complete	none	n/a	stmt
PPUTT	complete	none	n/a	none
IGOR	complete	n/s	n/s	none
dbf ML	commun	stmt	program	none
VISIT	commun	stmt	program	stmt
IDD	none	global, mult	program	ipc
bdb	none	local	program	none
CXdb	none	local	program	none
ldb	none	local	program	none
LPdbx	none	local	process	none
MDB	none	local	n/s	none
MpD	none	local	n/s	n/s
UDB	none	local	program	none
IPD	none	stmt	process	none
NodePrism	none	stmt	program	none
Parasight	none	stmt	process	stmt
Ariadne	none	none	n/a	shmem
EBBA	none	none	n/a	stmt
HeNCE	none	none	n/a	stmt
MULTVISION	none	none	n/a	stmt
Voyeur	none	n/s	n/s	stmt

Table 3: Functional Characteristics

Analysis

The ordering of this table according to the replay facility reflects our belief that execution replay is a crucial feature that every concurrent debugger should possess. The replay facility allows for repeating the same execution many times, eliminating the problem of non-determinism inherent in concurrent programs. This makes possible use of other debugging techniques, like cyclic debugging, top-down analysis, breakpoint insertion, or single stepping, which otherwise would not be very effective in concurrent environments.

Nine of the debuggers presented in this table provide "complete" replay facility. The "complete" replay means that in the replayed execution all the information about the program state is available, as opposed to the "commun" (communication) replay, provided by two of the included debuggers, where only the history of the communication events is available for examination. The second type of replay is provided by the VISIT debugger, described in the previous sections, where the trace data about the high level events is gathered during the program run, and stored for later. This limited information is used for offline visualization of program execution. Although not all the program state is available, this type of visualization can be useful for constructing a high level view of the execution, which can be used granularity analysis and tuning.

The method used in providing the replay facility differs from one debugger to another. One of the popular methods is the Instant Replay, which is also used in the Panorama debugger. This method is based on saving the order of relevant events in the program execution, without saving the data associated with the events. The advantage of such approach is the low overhead introduced by the replay mechanism. The disadvantage can be found in the fact that even in the case when the user wants only to examine one of the processes, the entire program must be replayed. This problem is solved by the type of replay facility provided by Bugnet, where the IPC messages exchanged between processes are saved during the minitoring phase. In the replay phase the *send message* operations are not executed, and the *receive message* operations are replaced by reading the contents of the message from the log. In this way, during replay the user can run one process or many, depending on his debugging needs. The disadvantage of this method is the overhead in terms of time taken in saving the messages, and space used for storing the data included in the messages.

The discouraging observation that must be made when looking at this table, is the fact that only eleven from the twenty nine presented debuggers provide replay facility. The remaining eighteen debuggers ignore the problem of non-determinism. Although it does not mean that they are not good at all, these debuggers are missing a very relevant feature.

Another type of information presented in this table refers to breakpoint and their effects on the program execution. The most common type of breakpoint is the one denoted as "local", which is based on the state of the execution of a single process. Some of the debuggers provide only statement breakpoint, which allows only for breakpoint specification based on the location in the code. The global breakpoint, where the entire program state is considered for condition evaluation, is provided by only one of the debuggers included in this table (IDD). The effect that the breakpoint has on the program also varies from one debugger to another. The breakpoint can stop one process or the entire program. In some cases the user is able to choose the breakpoint effect according to his needs.

The last piece of information concerns types of events recognized by the debuggers. Some of the debuggers recognize interprocess communication events (marked by "ipc"), or shared memory events (marked by "shmem"). There are also debuggers that allow the user to define the events according to his needs. In such cases, execution of any statement can be defined as event. Many of the included debuggers do not support event based views of the debugged programs. This is the case of the debuggers that have been built by extending sequential debuggers.

2.5.3 User Interface

This section presents a compilation of information about features present in the user interfaces of the surveyed debuggers. The first subsection contains data in tabular form, the second subsection gives analysis of the data.

User Interface Table

The entries in table 4 are ordered according to the method they use to present the debugging information to the user. At the top of the table are the debuggers that provide multiple views of the executing program. Next are placed debuggers which use time process diagrams to provide an abstract view of program execution. The debuggers with time-proces diagrams are followed by a group of debuggers that use windows to convey useful information but do not create graphical views. The last group contains debuggers that do not use graphics and communicate with the user in textual form only. Alphabetical order is followed inside these groups.

- **Presentation** - in what way is the information presented to the user
  - *mult views* - of the execution are provided
  - *mult views +* - the views can be extended by the user
  - *windows* - multiple windows are used
  - *tp* - time process diagrams can be displayed
  - *command* - information is displayed in textual form
- **Examination & Modification** - capabilities for program state modification and examination
  - *global* - global state can be examined
  - *ipc* - communication state can be examined
  - *local* - local states can be examined
  - *+* - modification of state is possible
  -
- **Examination of Event History** - How user examines a recorded event history
  - *browser* - using an editor/browser
  - *language* - using queries in the indicated language
  - *replay* - only examination is to replay the history
  - *scroll tp* - scrollable time-process diagrams
  - *trace anim* - animation of the execution from the trace log

Debugger	*Presentation*	*Examination & Modification of State*	*Examination of Event History*
HeNCE	mult views	global	scroll tp
NodePrism	mult views	global, local+	n/a
Panorama	mult views +	global	n/a
ParaGraph	mult views +	global	replay
RP3	mult views	global	replay
VISIT	mult views	global, local+	n/a
Voyeur	mult views +	global	browser
IDD	tp	local, ipc	scroll tp
Instant Replay	tp	local	replay
MULTVISION	tp	global	trace anim
PPUTT	tp	global	n/a
dbf Agora	windows	local	replay
dbf Amoeba	windows	local+	replay
bdb	windows	global	none
CXdb	windows	global	n/a
LPdbx	windows	local	n/a
ldb	windows, audio	local	n/a
UDB	windows	local	n/a
Ariadne	command	global	query language
Bugnet	command	local, ipc	replay
CDB	command	local	n/a
EBBA	command	global	n/a
IGOR	command	local+	n/a
IPD	command	n/s	n/a
MDB	command	local	n/a
dbf ML	command	local	n/a
MpD	command	local	n/a
Parasight	command	local+	none
Recap	command	local	replay

Table 4: User Interface

Analysis

The table presented in this section compiles the most prominent features of the user interfaces provided by the surveyed debuggers. The first feature, the " presentation", shows what method is used in presenting the debugging information to the user. The second feature describes how the user can examine and modify the state of the debugged program. The last property of the user interfaces presented here is the method used in examination of the event history.

The first seven debuggers presented in the table use multiple views to illustrate the behavior of the debugged program. One of the popular views provided by most of the debuggers is the time-line view, where events occuring on each processor are displayed on one horizontal line. The displayed events are often related to interprocess communication, but can also include other relevant occurrences. Another often useful view, found in Panorama debugger, shows a global picture of message traffic by displaying the number of messages that are waiting to be received on each of the processors. The process tree view, provided by VISIT, shows the tree of processes spawned by a parallel program. The global view of the tree can be augmented with more detailed views of some parts of the tree. Another view, very useful in performance analysis, is the processor load balance view which shows the load on each of the processors participating in the execution. The views are often constructed in such a way that they give access to low details about the displayed events. This is the case of the time-line view in Panorama, where the user can click on any event with the mouse, to see additional information about the event. The multiple views are most advanced, and most desirable method of presentation, which allows for better understanding of often very complex behaviour of concurrent systems.

A factor that can make the multiple views even more appealing is the possibility of extending the default set of views provided by the debuggers according to the application needs. This feature, denoted by the plus sign, is provided by three of the presented debuggers.

One of the presented debuggers, the ldb, uses auditory data analysis. The user can listen to the audio representation of large volumes of data and try to identify anomalies, that can be later examined with traditional methods.

The next group of debuggers in the table use the time process diagrams to provide global view of program execution. This method can be considered to be a limited version of the multiple views method, with restriction to one view only.

The multiple windows are the next best thing, after use of graphics, in the user interfaces for concurrent debuggers. Many of the presented debuggers use this technique. These debuggers often rely on the technique of displaying information about each process participating in computation in a separate window. This can be useful, but a very important issue here is the way the different windows are controlled. It can be rather annoying when the user has to operate each window separately, so some global coordination is desirable. Also, this method can hardly be considered scalable when applied to a program consisting with hundreds of processes.

The largest group of debuggers fall into category denoted as "command", to reflect the fact that they provide command line interface only. They are simpler to build, since implementation of a good graphical user interface can take considerable amount of effort, but difficult to use what lowers their usability, and value as a software tool.

The feature placed in the middle column of the table shows how the user can examine and modify the program state. The debuggers with multiple views, and some of the event based debuggers allow for examination of global states of the execution, and sometimes for modification of local states. Others allow for access to local state of the processes only.

The examination of event history applies only to debuggers which take the event based approach to debugging. There are many methods used for this purpose. The event histories can be examined by using browsers, scrollable time process diagrams, trace animation, writing queries in some language, and replaying the recorded execution.

2.6 Summary

The survey presented here was preceded by another survey of similar scope, published in *ACM Computing Surveys* in 1989 by C.M. McDowell and D.P. Helmbold [33]. There are several differences between the two reports. The first difference is the time period covered. While the previous survey included documents published up to 1989, we have put the emphasis on publications that appeared in the last five years. In some cases however we have abandoned this rule and included earlier work. The sets of the debuggers covered in both surveys are therefore not completely disjoint, but this report contains more recent research material. Another important difference is the approach taken towards covered debuggers. The previous report takes a more theoretical approach, by concentrating on methods used in concurrent debugging. In this report our goal was to illustrate the functional aspects of the presented debuggers, and show how the debugging methods used can be useful in locating software bugs.

In spite of the different approaches, the 1989 survey can be used as a reference to assess the progress of the concurrent debugging research in the recent years.

The most visible change that can be observed in the debuggers recently developed is the use of more advanced presentation techniques than these employed previously. The use of multiple windows is almost a standard feature for today's debuggers. Also, the use of graphical representation of debugging data, interprocess communication or processor load is far more common than it was before. Some of the debugging tools presented here allow for viewing the program execution from different perspectives by creating multiple views of the program execution. In some cases it is also possible for the user to create views specific for the needs of the debugged application. There are also attempts to use non-conventional presentation techniques, like sound, for debugging purposes. Even in the cases when the information is displayed in textual form and multiple windows are not used, the upgrading of the user interface is always high on the list of future improvements. This is a big step forward, since a good user interface allows for easier access to the debugger features, and can make the debugging process less painful.

Another important issue in maximizing of usability of a debugger is the possibility of porting it to other systems. The benefits of using the same tool on different machines include eliminating the learning time required when switching to different architectures. The user will most likely choose a tool that s/he knows, over other, even more "sophisticated", but new. Some of the debuggers are difficult to port because they depend heavily on hardware or software details of underlying systems. Others cannot be ported because of the scalability problem - even if they perform well on one system, their performance on other systems is not acceptable. For these reasons, the appearance of debuggers developed with portability in mind, like the Panorama debugger, is very promising. By using the layered structure for the debugger architecture, the creators of the Panorama, have succeeded in developing a debugging tool, that can be used on any computer, as long as the lowest layer (a base debugger) can be provided. The base debuggers are in many cases provided by the computer vendors, and although they have limited capabilities when used alone, they can be very useful by providing low level details to a debugger with more advanced features.

Another promising fact is the appearance of debuggers that integrate debugging techniques used in sequential debugging with techniques specific to concurrent debugging. Examples of such debuggers, presented in this report are debugger for Amoeba, Panorama debugger, and debugger presented by the PPUTT group from Rochester. A debugger following this philosophy will benefit from the fact that many users are familiar with sequential debugging techniques and will feel comfortable seeing them integrated into a concurrent debugger. The problems specific to concurrent debugging are also addressed in these debuggers, so they can provide for all concurrent debugging needs.

A less optimistic observation that results from this survey is that not too many of the "sophisticated" debuggers are fully implemented. In many cases the debugging tools have been proposed and some partial or prototype only implementation has been achieved. This is often the case of the debuggers being developed in academic institutions. The commercial developers, on the other hand, often provide fully functional debuggers with the systems they deliver, but these debuggers are often very limited and simple in the facilities they provide. In effect the users are left with good debuggers existing mainly on the paper, and not too great debuggers being used in the computer labs. This situation is a little ironic since one would think that since concurrent programs are more difficult to write than sequential ones, the programmers should be supplied with good debugging tools.

Chapter 3

Deterministic replay

The non-determinism present in distributed systems makes debugging quite difficult. One approach to deal with this issue has come to be known as "deterministic replay". The deterministic replay alone is not adequate to locate errors, and it has to be integrated with other techniques of debugging.

3.1 Non-determinism in distributed systems

Non-determinism is an inherent property of distributed systems. Two consecutive executions of the same distributed program can possibly have different outcomes, even if the external input was the same in both executions.

The non-deterministic scenarios that will be presented in this section are based on the Mach operating system. The programming model used in Mach is built on the notions of *task* and *thread*. *Task* is a repository of resources. *Thread* is a light weight process that executes program instructions. A distributed program can consist of many tasks, working towards a common computational goal and communicating through message passing or shared memory. A task can contain one or many threads. Every thread belongs to a single task and may access all of that task's resources. The sources of non-deterministic program behavior in such a programming model can be viewed at two levels:

- **Interprocess communication level.** The communication patterns can differ from one execution to another. The order of arrival of messages at a port can be different. In the case of non-reliable communication channels some messages can get lost in one execution but reach the destination in another execution, which adds more complexity.

- **Thread interaction level.** In a multithreaded environment the additional source of non-determinism is the interaction between threads sharing the same resources, which include accessing the same memory locations or using the same interprocess communication primitives. In the case when two threads access the same program variable, the access patterns might vary from one execution to another, creating different program behavior.

3.1.1 Mach Basic Objects

This section illustrates different sources of non-deterministic behavior of distributed programs under Mach operating system, but before presenting the detailed list of non-deterministic situations in Mach programs, we will introduce the basic objects of the Mach operating system:

- **Task :** collection of resources including virtual memory and communication ports. Tasks are passive, they do not run on a processor.

- **Thread:** active execution environment. Each task may support one or many threads; all threads have equal access to task's resources. Each thread has private execution state that consists of set of registers, such as general purpose registers, stack pointer, program counter, and a frame pointer.

- **Port :** communication channel - a logical queue of messages protected by the kernel. Ports are location transparent, they can change owners. Access to ports is granted according to access rights (receive, send, ownership).

- **Message:** typed collection of data used for communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.

3.1.2 Message passing system calls

The message passing calls introduce many potential sources for non-repeatable program behavior at both interprocess and thread interaction level. The possible non-deterministic scenarios for each call are presented in table 5.

System call	Non-deterministic scenarios
msg_receive	If timeout is specified as a part of the receive operation, a failure can occur when the message M does not arrive at port P before the time specified by timeout argument elapses. Since the time of arrival of messages depends on the sending task as well as on the network delays, this call might fail due to timeout in one execution but succeed in another.
	If more than one thread in a task is receiving messages from a port, then in different executions the threads can receive the messages in different order. If thread T1 receives message M1 from port P and thread T2 receives message M2 from the same port P in one execution then in another execution, due the different scheduling of threads, thread T2 might receive message M1 and thread T1 might receive message M2, creating different execution history.
	If there are more than one task that have send rights to a specific port, then in different executions the order of arrival of messages to a port can be different. For example, if task K1 and task K2 both send messages to port P which belongs to task K3, then the message M1 from K1 can be enqued at the port P before the message M2 from K2 in one execution and after message M2 in another execution. Different order of M1 and M2 can affect execution of K3.
	In case of transfer of the receive rights, some of the messages sent to a port can be received by the old task in one execution and by the new task in another execution. If receive rights of port P are moved from task K1 to task K2 before task K3 sends message M then M will be received by K2. If the receive rights are transfered after M is send, task K1 receives message M.
	If there are no tasks with send rights to the port on which the receive operation is attempted, then the message receive call will fail. The relative timing of the receive call and the termination of sending tasks can cause success of the receive call in one execution and failure in another execution.
msg_rpc	Non-deterministic scenarios are the same as for *msg_send* and *msg_receive*.
msg_send	If the timeout is specified, the message send operation can fail when the destination port P is full, and the message M cannot be enqued within specified amount of time. The state of port P might depend on the sequence of message receive and message send calls in the program and can differ from one execution to another.
	If the destination port specifies a name P which has been deallocated, the message send call will fail. The timing of the send and deallocation calls can be different in each execution causing the send to fail some time and succeed some time.

Table 5: Message passing calls

3.1.3 Network message server calls

This section describes non-deterministic scenarios for calls to the network message server (*netmsgserver*). The *netmsgserver* is provided with Mach-based systems to facilitate passing of send rights between Mach tasks [4]. A task wishing to pass send rights registers an ASCII name and the port with the network message server (*netname_check_in*). Another task may obtain the send right to that port by querying the message server using the same ASCII name (*netname_look_up*). A port can be removed from the message server by an authorized task(*netname_check_out*). All non-deterministic scenarios for the *netmsgserver* calls are presented in table 6.

System call	Non-deterministic scenarios
netname_check_in	If many tasks are trying to check name N with the name server a task may succeed or fail depending on the order of their attempts because the same name can be checked in with name server only once.
netname_check_out	If several threads in a task are trying to check out the same name N, then only the first thread will be successful. All other threads will fail because a name can be checked out from the name server only once. The successful thread can be different in each execution.
netname_look_up	The success of the name look-up call depends on the relative order of look-up check-in and check-out calls. The look-up for name N might fail when it is executed before N was checked in with the name server or after N was checked out. The scheduling of these operations might be different in each execution, creating different execution histories.

Table 6: Netmessage server calls

3.1.4 Port related calls

Mach system calls presented in table 7 allow for performing different operations on ports. Along with basic allocation and deallocation calls, there are calls that allow to change default properties of ports like name (*port_rename*), transfer port rights among different tasks (*port_insert, port_extract*), or enquire about messages waiting at the port (*port_status*). The *port_names* call returns information about all ports in the task's port space. It returns names of the ports and their types.

Ports, just like any other task's resource, are shared between all threads supported by the task. Interaction between threads, as well as communication with other tasks in the program, make the port calls a potential cause of non-deterministic behavior.

System call	Non-deterministic scenarios
port_allocate	This call might fail when the kernel runs out of memory allocated for ports. The state of resources can differ from one execution to another so the outcome of the call might be different each time the program is executed.
port_deallocate	This call might fail in a multithreaded environment when several threads are trying to execute it on the same port. Only the thread that is scheduled first to execute *port_deallocate* will be successful, because a port can only be deallocated once.
port_rename	This call will fail when the new name already exists Race conditions can arise in a multithreaded environment. If thread T1 allocates port under name N before thread T2 executes a call to rename port P to name N, then thread T2 will fail because duplicate names are not allowed.
port_names	This call returns data about the task's port name space. This data can differ from one execution to another since the state of the name space depends on all threads in the task. In a multithreaded environment the timing between execution of the *port_names* call and different port manipulation calls can cause the *port_names* call to give different results each time the program is executed.
port_status	This call can return different values in each execution. Because of the non-determinism present in message receive and send calls, the state of port P can be differ from one run to another depending on how many messages have arrived at the port and how many have already been received.
port_extract **port_insert** **port_rename** **port_set_backlog** **port_set_backup** **port_status** **port_type**	These calls will fail if they are executed by thread T1 before port P is allocated. If the port allocation call is executed by thread T2, then the relative order of execution of T1 and T2 can be different in each run.
	These calls will succeed only when the task has receive or ownership rights to port P at the time when thread T1 is executing any of these calls. If port rights of P are to be transfered to other task by thread T2, then the success or failure of these calls will depend on the scheduling of threads T1 and T2.
	These calls will fail in the case when they are executed after port P was already deallocated. If port P is to be deallocated by thread T2, then in some executions thread T1 will succeed in these calls, and in others it will fail.

Table 7: Port calls

3.1.5 Port set related calls

A port set is a named collection of ports and can hold zero or more receive rights, allowing a thread to block waiting for a message sent to any of several ports. A port can belong to at most one port set at any time.

A port set can be allocated by *port_set_allocate* call. Messages are received from the port set with the *msg_receive* call, the receive operation completes when a message is available on any port within the named port set. Port set manipulation calls include *port_set_add*, *port_set_remove*, *port_set_status* and *port_set_deallocate*. Table 8 lists all port set related calls with the potential non-deterministic situations.

System call	Non-deterministic scenarios
port_set_add	If thread T1 executes a call to add port P to port set S1 and thread T2 executes a call to add P to set S2, then one of T1 and T2 will fail because a port can belong to one port set at a time. Since the relative order of execution of this call by T1 and T2 can be different in each run, one time T1 might be successful in adding P to set S1, and another time T2 might succeed and add P to S2.
port_set_allocate	This call might fail when the kernel runs out of memory allocated for port sets. The state of resources can differ from one execution to another so the outcome of the call might be different each time the program is executed.
port_set_deallocate	This call might fail in a multithreaded environment when several threads are trying to execute it on the same port set. Only the thread that executes *port_set_deallocate* as the first will be successful, because a port set can only be deallocated once.
port_set_remove	A port can be removed from a port set only once, so if threads T1 and T2 try to remove port P from port set S, then in one run T1 can succeed and in another T2.
port_set_status	This call can return different values in each execution. Because of the non-determinism present in *port_set_add* and *port_set_remove* calls,the state of port set S can be differ from one run to another depending on how many ports have been added to port S and how many have been removed.
	This call will fail if it is executed by thread T1 before port set S is allocated. If the port set allocation call is executed by thread T2, then the relative order of execution of T1 and T2 can be different in each run.
	This call will fail in the case when it is executed after port set S has been deallocated. If port set S is to be deallocated by thread T1, then in some runs thread T2 will succeed in execution of *port_set_status*, and in others others it will fail. The result will depend on the order of calls executed by threads T1 and T2.

Table 8: Port set calls

3.1.6 Task and thread related calls

System calls listed in table 9 allow for creation and control of Mach tasks and threads. Non-deterministic program behavior in execution of these calls might result from shortage of resources or interaction among multiple threads.

System call	Non-deterministic scenarios
task_create **thread_create**	These calls might fail due to shortage of resources.The state of resources can be different in each execution.
task_terminate **thread_terminate**	If several threads are competing to execute one of these calls, only the first one will be successful.
thread_resume **thread_resume**	These calls will fail in the case when the suspend count is already zero. If these calls were preceded by one suspend call and threads T1 and T2 were competing to execute resume call, then only one of T1 and T2 would succeed.

Table 9: Task and thread calls

3.1.7 C Threads calls

The C threads library provides programmers with an interface that allows for writing multithreaded programs in C without using the thread primitives. The provided calls facilitate creation and synchronization of multiple threads running within one task. Many of these functions can behave differently in consecutive executions.

C Threads call	Non-deterministic scenarios
condition_signal	This call can wake up only one thread from the set of threads waiting for a lock. For example, if threads T1 and T2 are waiting for the same lock L, then when thread T3 signals that L is free T1 or T2 will be randomly chosen by the kernel to receive the signal.
mutex_try_lock	This call returns TRUE when the lock is available or FALSE when the lock is held by some other thread. In multithreaded environment, the return values from this call might vary from one execution to another. If threads T1 and T2 compete for the same lock L and thread T1 issues *mutex_try_lock* call, then in some executions the call can take place after T2 have released L and return TRUE, and in other executions T2 can still hold L and the call will fail.

Table 10: C Threads function calls

3.1.8 Virtual memory calls

Mach gives the programmer complete control of a program's memory layout through the virtual memory system calls. The virtual memory primitives can operate on the calling task itself, another task on the same machine, or even on a program in a different machine. Mach system calls presented in this section allow for virtual memory operations. Non-deterministic scenarios related to virtual memory calls might result from shortage of resources, interaction among threads in the task, or interaction between tasks in the program. The listing in table 11 gives calls that must be monitored in a multithreaded environment in order to produce a deterministic execution of a program.

System call	Non-deterministic scenarios
vm_allocate	This call can fail when there is not enough memory space to satisfy the request.In a multithreaded environment it can happen that other threads have used all the available memory before the call was executed. The point of failure can be different in each execution.
vm_deallocate	This call can succeed only once, so it would be non deterministic in a multithreaded environment, when more than one thread would try to execute it.
vm_inherit **vm_protect**	These calls might fail when multiple threads operate on the same memory region and when the target memory region is deallocated by some other thread before the protection change takes place.
vm_read	The data returned by this call can be different in successive executions when several threads are accessing the same region of memory. For example, if thread T1 reads certain memory region R and thread T2 writes R, then the scheduling of threads T1 and T2 will decide on the data returned by the read call.
vm_region	This call returns data about a memory region. In a multithreaded environment the state memory can be different each time the call is executed due to different thread scheduling patterns.
vm_write	This call can fail when the memory region is no longer available for writing. For example if a thread in task K1 wants to write to memory of task K2, then it might happen that K2 executes memory protection call or deallocates the targeted memory region before K1 executes the write call. The timing of the actions taken by tasks K1 and K2 can vary from one execution to another.

Table 11: Virtual memory calls

3.2 Deterministic replay approach for debugging

To make use of the techniques of sequential debugging in the distributed world, deterministic replay is introduced. Many of the distributed debugging techniques can be improved by adding the capability of deterministic replay of a distributed program. In this context, we make the following observations:

- Multiple processes tend to generate a lot of output making observation for debugging difficult, and creating possibility of missing some important results.
- The programming environments for parallel architectures and parallel programs are not as mature as the environments for sequential machines, and often lack tools for collecting and analyzing the output data.
- Reproducible behavior is essential for cyclic debugging.

In the following, we will examine the various techniques used in sequential debugging in light of deterministic replay :

Cyclic debugging

Cyclic debugging is one of the more popular debugging techniques used in debugging sequential programs. This technique can be also used with distributed programs when it is augmented with deterministic replay mechanism. Successive trials of the cyclic method can be used to provide successively more information about the part of the program being debugged. One way to gather necessary information is to introduce some snapshot takers at suitable points in a program. This is successfully practiced with sequential programs, but can be used with distributed systems only when the determinism of re-execution is guaranteed. Each output statement in parallel programs can introduce delays and as a result change the relative timing of events occuring within the program. The cyclic method of debugging can be very valuable in debugging distributed programs, especially when other debugging tools are not available.

Breakpoint insertion

Breakpoint insertion is commonly used in sequential debugging. Breakpoints can be added at interesting points in the code. When the execution is suspended at the breakpoint, the programmer has a chance to examine the system state. This is straight forward for sequential systems since programs consist of a single process on a single processor. The breakpoint semantics for concurrent systems are not obvious, because breakpoint specification and detection must incorporate all processes in the program. This new type of breakpoint become known as "global breakpoint". Global breakpoint is based on the state of the entire program and stops all processes taking part in the computation. Local breakpoints, in contrast to the global breakpoints, are defined in context of one process and stop only the process into which the breakpoint has been inserted.

Both local and global breakpoints can benefit from availability of deterministic replay. Because detection of a global breakpoint is non-trivial, it often requires execution of many debugger instructions and introduces delays causing probe effect. For local breakpoints, the breakpoint specification is easy, but wh(n one process is stopped at the breakpoint, all other processes that interact with the stopped process are also affected. The sequence of events in the program execution can be changed by the delays introduced when the process is suspended at the breakpoint. By eliminating the probe effect introduced by breakpoints, deterministic replay makes it possible to cycle through breakpoints in many different processes during program replay, and examine the system state each time.

Single-stepping

The single-step technique allows the programmer to follow the execution of the program instruction by instruction. While stepping, it is very important to use the deterministic replay technique for distributed programs since the delays introduced by stepping will affect both the threads within the stepped process, and other processes taking part in the computation.

Top-down debugging

Top-down debugging methodology can be effectively applied to debugging distributed programs when combined with deterministic re-execution. Because the distributed programs tend to be long and produce large volumes of data, it is often helpful to be able to abstract details in order to understand the behavior of a program. This approach allows the programmer to start with the high-level view of a program behavior. Then, in successive runs, this viewpoint can be refined based on the available information to any level of detail desired. The capability of deterministic replay is desirable for top-down debugging since it eliminates possibility of the error being present in one execution and absent in another, what is fairly common in distributed programs. In such case the programmer could spend long time tracking an error that is visible at one level of abstraction but could not be located at another level.

Event based debugging

The event based view of program execution is created by collecting information about predefined events during execution of the program. The definition of an event can vary from one system to another. The event information can be used in several ways: browsing with some tools, replay based on recorded events, simulation of program execution. The use of event log technique in conjunction with deterministic replay can allow for repeatable executions of a subset of processes involved in the computation. By using the event log, the execution of the processes uninteresting from the debugging point of view can be simulated, using data extracted from the logs, allowing for re-execution of the relevant subset of processes only.

3.3 Functional requirements

Based on the information presented in the previous sections we define the functional characteristics of a distributed debugger, and the characteristics of a "minimal" distributed debugger. First we define the necessary conditions for debugging distributed programs. Next we explain how these conditions can be satisfied by a distributed debugger.

3.3.1 Conditions for distributed debugging

The following are the necessary conditions for successful and efficient distributed debugging:

1. Non-deterministic behavior must be eliminated.

2. Tools must exist for extracting information about run time details.

The two conditions are complementary to each other. The elimination of non-determinism alone will not help in debugging if no information about program execution will be provided to the programmer. On the other hand, the techniques that can provide the needed information cannot be used without the guarantee of deterministic-execution because some of them, like cyclic debugging would not be applicable, and others would not be very effective.

3.3.2 Functional requirements for a full scale debugger

The conditions described in the previous section can be best satisfied by means of the two-phase debugging scheme. The non-deterministic behavior can be eliminated by recording the necessary control information about the execution. Then different debugging techniques can be applied on the controlled replay of the program. The techniques for phase two can come from both the sequential and distributed world of debugging. All functional requirements for a full scale distributed debugger can be satisfied by following facilities :

1. **Deterministic replay** is necessary to implement the two-phase debugging methodology.

2. **Source code level information** about each process in the computation is needed to understand the fine details of program execution. Without this information the source of the error cannot be fully identified.

3. **Global breakpoints** are required to halt the execution of the program in a consistent state, to examine or modify its state, and to restart execution

4. **Checkpoint and rollback** facility is needed for restarting the program execution from any point in the program history. This is particularly important for very long programs.

5. **Data and execution visualization** tools are needed for interpretation of large volumes of data and understanding behavior of large programs. Graphical interpretation is often the best technique.

6. **Graphical User Interface** is necessary for providing the user with easy access to the above facilities, what is very important when debugging of several tasks at the same time is required.

3.3.3 Functional requirements for DDB

A full scale distributed debugger cannot be implemented in the scope of a Master's thesis project. The team based approach which was successfully applied in the CDB project would be a good solution. The second best alternative is to design and implement a smaller debugger that would provide basic set of facilities needed for distributed debugging. The debugger in its present "minimal" form will be powerful enough to be effectively used in debugging of distributed programs. The facilities that are absent in the present version of the debugger could be implemented in the future.
The list of components needed for the "minimal" debugger is as follows :

- Deterministic replay engine
- History files interpreter
- Source level debugger
- Graphical user interface

Deterministic replay engine
The deterministic replay will be present in both phases of the debugging process. During the first phase, called the *monitoring* phase, the distributed program will be executed and the information about program events will be collected and stored in the history files. The data stored will define the order of non-deterministic events. During the second phase, called the *replay* phase, the replay engine will use the information stored in the history files to control the execution of the program, so that it can follow the path of the original execution.

History files interpreter
The information stored in the history files for the purpose of controlling the execution can have alternative uses. It can be used to provide the user with high level view of the program behavior. This type of information can help the user to identify the faulty portion of the code, which can be further examined with other tools. The interpretation can be conducted in two ways. One way is to provide textual interpretation of the events and their outcome by interpreting information stored during record phase. Another way is to draw time-process diagrams. Time process diagrams can be very helpful in illustrating interaction between processes.

Source level debugger
A source level debugger can be used in the *replay* phase of debugging to provide all facilities that are conventionally used in sequential debugging. When a local debugger is attached to each process in a distributed program, the programmer has control over each task. He can set breakpoints, step the programs instruction by instruction, examine the source files, query the memory contents, or view the stack frames. These all would be possible because the execution is controlled so that the delays introduced by breakpoints and stepping will not change the order of events in the debugged program.

Graphical user interface
A graphical user interface is essential for any distributed debugger. It can make debugging activity easier by providing good access to large volume of information. It can also allow for handling output from multiple tasks and organizing communication with the base debugger. Graphics are also needed for displaying the time-process diagrams.

3.4 Other debuggers based on replay

The three debuggers presented in this section are based on the deterministic replay facility. Each of them represents a different approach to achieving deterministic replay. Bugnet [49] reproduces the process communication environment of a distributed program by saving all message events and periodically saving all variables in checkpoints. During the replay the execution can be restated from checkpoints and the message contents can be retrieved at appropriate times. The CDB debugger [40] saves all messages coming from the external environment, but saves only control information about messages within the debugged application. The Instant Replay [30] addresses the problem of reproducing concurrent access to shared memory systems, but can be extended to message-passing. Instead of recording values of variables, Instant Replay only records the order in which processes acquire and release locks.

3.4.1 Bugnet

Bugnet is a distributed debugger developed at the State University of New York at Stony Brook. It was designed to be used in debugging C programs distributed within a network of Unix systems. This debugger is interesting because it combines deterministic replay with checkpointing facility.

Bugnet monitors the execution of the program and gives the user all relevant information about the program behavior. This information includes interprocess communication, input and output events, and execution traces of all component processes. When an error is detected, Bugnet allows the programmer to roll the execution back and replay the events that caused the error. During the replay phase the user has a choice of replaying one process or many. The IPC messages that would be sent by the processes that are not being executed are provided by the debugger.

The capturing and time-stamping of all interprocess communications are the key features of the Bugnet debugger. The contents of messages are needed so that all external influences on a local process my be simulated. The time-stamping is necessary for operation of the debugger in the environment where no global clock is available.

Another important feature of the debugger is its ability to take periodic global checkpoints. In order to replay events at the same time intervals as they originally occurred, the Check-pointing algorithm periodically saves globally consistent states of the execution. The state of each application process is captured when the execution globally halts after some period of time (15 to 30 seconds). The decision to halt is made independently on each machine (to make it possible, the machines sometimes need to re-synchronize their clocks). Because all processes stop together and each process takes the same amount of time to save its state, the pauses are hidden from the execution unless an application reads the real clock. The only problem in this scheme are messages that are sent before all processes halt for checkpoint but arrive during the pause time. These messages have to be saved and resent locally to arrive at the correct time during the next run period.

The big advantage of the deterministic re-execution method in Bugnet is the possibility of replaying only the processes that are of interest to the user, which allows the programmer to focus on the process that contains the source of the error. The checkpointing facility is also very useful in case of long programs. The user does not need to replay the program from the beginning every time because the execution can be restarted at any point prior to the occurrence of the bug.

The disadvantage of this method is the storage overhead due to saving of the messages exchanged between processes.

3.4.2 CDB debugger

CDB is a debugger for distributed applications running on the top of the CHORUS distributed operating system developed at the Chorus Systems, France. The debugger operates on both shared memory and message passing systems. Besides the deterministic replay facility the debugger provides many of the common features found in traditional debuggers. The programmer can set breakpoints and query the state of the process, or display backtrace of stack frames which informs the user about function calls that have been executed before the program was stopped. It is also possible to examine or modify address space of a process.

The execution replay facility is based on the notion of *debug session*. A debug session can be *record* or *replay*. When an application executes in *record* session, its log is recorded in a *session log*. In the *replay* session, the behavior of the application is constrained to match the previously recorded session log.

There is one session log produced for each processor. The contents of the log can be analyzed with use of specific tools provided for this purpose. The tools range from the simple ones, which produce an ASCII human readable listing of a log's events, to tools that use the logs to draw space-time diagrams showing the causal relationships between events.

During the execution replay, the user can set breakpoints and query data. The speed of the execution during replay can be also adjusted: it can be replayed at normal speed, or context switch by context switch, or step by step. In step by step mode, the user chooses one thread and makes it execute one machine instruction. In the context switch by context switch mode, the user chooses one thread and makes it run to the point where the thread is preempted.

The approach that the debugger takes to recording relevant information differs according to the source of the information. For example, CDB takes the *data driven* approach in the case of messages received from the environment. The contents of these messages are saved, so they can be recreated at any time. The *control driven* approach is applied to the events internal to debugged program, in order to reduce the amount of logged information.

The proposed future improvements to the CDB debugger include check-pointing facility and implementation of source level debugging in the form of cooperation between the CBD and GDB which would act as a local debugger. The interesting point of the replay facility in the CDB is the two-fold approach to storing information. It is efficient since not too mach data needs to be stored, but at the same time allows for simulation of the external environment at replay time.

3.4.3 Instant Replay

The Instant Replay method was introduced by Thomas J. Leblanc and John M. Mellor-Crummey in [30]. The mechanism of Instant Replay was demonstrated in a debugger prototype implemented on the BBN Butterfly^{TM} Parallel Processor. The Butterfly Processor at the University of Rochester consists of 128 processing nodes connected by a switching network. Each switch node in the switching network is a 4-input, 4-output crossbar switch with a bandwidth of 32 Mbits/s. The memory of the system resides at individual nodes, but any processor can address any memory through the switch. The remote memory reference takes five times as long as the local memory reference. The operating system for the Butterfly is the Chrysalis.

The approach taken by the designers of the Instant Replay method treats all interactions between processes as operations on shared objects. Modifications to the objects are represented by totally ordered sequences of versions, where each version has a corresponding version number. This number is unique to each object. During the *monitoring* phase of the debugging, the partial order of the access to each object is recorded. This partial order is specified by a sequence of version numbers maintained for each object. To record the partial order, the debugging system maintains the current version of each shared object, and the number of readers of each version for each object. Each process executing as a part of the parallel program records the version number it accesses in the *monitoring* phase. In the *replay* phase this number is retrieved and used to ensure that the process sees the same value and in the same order it has seen in the original execution. The history information can be used as many times as necessary to repeat the original execution.

Instant Replay presents a general solution to the problem of non-determinism in concurrent debugging because it is applicable to both loosely coupled and tightly coupled environments. This method can be applied to any concurrent system since it does not depend on any particular form of interprocess communication, and it does not require synchronized clocks or globally consistent logical time. One of the important advantages of this method over other replay methods is the small overhead that the debugging system introduces. The overhead is small because only the order of relevant events is saved but not the data associated with the events. The time overhead is less than 1 percent, and the space overhead is also reasonable, making it possible to record events of a large scale production system.

The disadvantages of this method include the fact that the replay must include the whole cluster, not just the subset of processes. This can be rather impractical in the case of large systems. Another disadvantage of this method surfaces when the granularity of the communication is very small. In this case, it could become so that the space used for recording the version numbers associated with shared objects is larger than it would be if the contents of memory locations were recorded.

3.4.4 Evaluation

All of these debuggers described above, provide very good support for achieving deterministic re-execution of a concurrent program by using different mechanisms adjusted to specific environments, and debugging requirements. However, as we said before, deterministic replay alone is not useful in locating software errors. There must be tools cooperating with deterministic replay to allow the programmer obtain information about program behavior. All of the debuggers presented above are incomplete in this respect.

The Bugnet debugger provides information about interprocess communication, I/O events, and execution traces of the processes but source level debugging is still to be implemented by linking Bugnet with Unix *dbx*. There is no graphical representation of program execution.

The same problem exists in the CDB debugger. Although it provides variety of graphical and textual tools for analysis of the logs produced during debug sessions, the source level debugging is planned to be provided in the future by cooperation of the CDB and GDB.

The current implementation of the Instant Replay method consist only of the mechanism needed for deterministic replay support. Other tools, including source level single-process debuggers, tools to monitor execution with graphical displays, and specialized compilers are still to be developed.

The DDB debugger presented in this Thesis aims at providing an efficient and non-intrusive deterministic replay facility combined with a minimal but sufficient set of tools to support the debugging process. The debugging tools include source level debugger, time-process diagram viewer, and textual interpreter for log files. The program-view that the programmer obtains through the time-process diagrams and textual interpreter of log files can be further refined by using the cyclic debugging techniques practiced with the help of the source level debugger. The basic set of tools available in DDB can be extended in future to create a full scale commercial quality distributed debugger.

Chapter 4

Capturing non-determinism of Mach

In this chapter we describe the method that was used in development of the DDB debugger for capturing the non-deterministic behavior of distributed programs running under Mach operating system. The general approach to recording program execution can be described as *control driven* because the information gathered during the execution of the program relates to the order of events in the execution, not the data corresponding to the events (*data driven* approach). The information gathered in the *record* phase is used in the *replay* phase for achieving deterministic re-execution.

4.1 General approach to debugging

There are two distinct approaches to developing debugger applications described in the literature, based on the way in which the debugger executes control over user program. These are:

- **Implementation Based Approach**
  With this approach, the implementation of the run-time system, the operating system, and the compiler are modified to support debugging. The problem with this approach is that debuggers of this type are difficult to port.

- **Language Based Approach**
  This approach involves introducing modifications to the source code of the application program.

The approach taken in the development of the DDB debugger is a combination of the two methods. The debugger uses a language based approach when it records information about non-deterministic events in the monitoring phase and when it controls the execution of the application program in the replay phase. The mach system calls are "wrapped" by special debugger library routines in the preprocessing stage of the compilation. These library routines execute the intended call plus some additional statements for either collecting information (in the *monitoring* phase) or controlling execution (in the *replay* phase). The implementation based approach is used for operation of the local debuggers. The GDB requires that the program is compiled in such a way that a symbol table is produced which contains information needed for the local debugger to operate.

4.2 Assumptions

The assumptions made about the programming environment are necessary for designing an effective and efficient deterministic replay mechanism. The restrictions placed on the user programs debugged with DDB are introduced for two reasons. First reason is that the approach taken is sometimes insufficient to guarantee deterministic execution, as in the case of *port set* described below. Second reason is that the project would not be possible to complete within the Master Thesis work if the restrictions have not been placed. The following conditions are assumed to be true:

- **An equivalent virtual machine must be available.**
  To make deterministic replay possible, we make an assumption that the original execution of the program and subsequent replays occur in equivalent virtual machine environments. Two virtual machines A and B are said to be equivalent with respect to program P if program P can exhibit the same behavior whether executed on virtual machines A or B. This requirement is realistic only when the program P does not depend on physical details of its virtual machine. For example, if P's execution depended on real-time clock it would be very difficult to simulate the virtual machine during replay.

- The following is assumed to be true for user programs debugged with DDB:

1. **Migration of port rights is not allowed.** We assume that port rights are not transferred between tasks. The send rights to a port in user programs can be acquired only through the network message server. This restriction will simplify the keeping track of messages send between the tasks easier. Control of migrating ports is difficult and time consuming to implement, but it can be achieved in the future versions of the debugger.

2. **Port sets are not used.** Mach allows for grouping ports into a *port set.* The grouping allows the programmer to issue a single **msg_receive** system call that will receive the first available message from any of the ports that are members of the port set. We have decided to exclude *port sets* from the application programs because they introduce potential for non-deterministic behavior that is impossible to control without modifications to some of mach system calls.

3. **Memory is not shared between tasks.** The basic virtual memory functionality permits full sharing of read/write memory between two tasks only through inheritance. This results from the fact that the child of a task can only be created on the same machine, so the problem of supporting shared memory between tasks separated by a network is eliminated. Memory sharing between unrelated tasks can be achieved through use of memory managers that execute outside of the kernel, in the user mode. The decision to disallow memory sharing has been taken because control over non-determinism that could result from memory sharing would be difficult. Another reason is that the primary way of IPC communication in our distributed environment (network of PC) is message passing, so the restriction would affect very small percent of programs.

We believe that the restrictions placed on user programs will affect only a small part of possible Mach programs. This is because limitations that are introduced do not affect any critical aspect of program operation, so the programs can use the debugger by temporarily excluding the parts of the program that violate the restrictions. Some of these restrictions can be removed in the future versions of the debugger.

4.3 What information is recorded ?

Deterministic replay mechanism used in the DDB debugger is an adaptation of the method used in Instant Replay. In the *record* phase of the DDB, just like Instant Replay, we collect control information about the execution. No data values associated with the user program are recorded. The debugger stores only access patterns to shared variables within each task, and order of send and receive operations for each IPC port. Although the general approach is the same, there are differences between methods used by DDB and Instant Replay. Because of difference in the environments the two debuggers support, the sources of non-deterministic behavior are different, and in effect, the events that need to be controlled are different. The implementation of the Instant Replay described in [30] deals with shared memory model where non-determinism results from different access patterns to variables shared by multiple processes. In the case of the DDB debugger there are two potential sources of non-deterministic behavior. First, there is interaction among tasks in the program by message passing which might take different paths each time. Second, there is interaction among threads in each task that is also non-deterministic. For this reason, it is not sufficient to record only data concerning the interprocess communication, like it is done in the Instant Replay debugger. The interactions of threads inside each task must also be recorded and controlled. The remaining part of this section presents control information that is recorded in the history file of each task and explains why this information is necessary.

4.3.1 Task identification

The basic requirement for successful re-execution of a distributed program is to identify the tasks taking part in the computation. The debugger does not require that the task - host configuration be the same in re-executions of the same pro[illegible] [illegible]ks can be executed on different hosts as long as the equivalent virtual machine requirement is satisfied. The debugger also does not impose the "one task per machine" requirement, so there can be only one task executing on each host or all tasks executing on one host and the configuration can change from one re-execution to another.

To allow this flexibility, there must be a way of uniquely identifying each task in the user program and matching the appropriate history files with the tasks in the *replay* phase. Since there is no requirement that each task in the program has a distinct name of the executable file, another way of task identification is needed. This is done by attaching an identifier to the command line arguments of each task in both the *record* and *replay* phase. In the *record* phase the identifier is used in creation of history files. In the *replay* phase the identifier is used to open the corresponding history file for each task.

Task identification is also necessary for control over message IPC operations. This identification is achieved with the help of the Central Name Server (**CNS**). Each task, before starting execution of the user code, checks itself with **CNS**. In the *record* phase, **CNS** assigns a number to each task that will be later attached to all messages sent by this task. This number is stored in the task's history file. In the *replay* phase the number is simply retrieved from the history file. The user task is not aware of this naming scheme since all the related operations are done transparently.

4.3.2 Information related to C Thread calls

The information related to C Thread calls is presented in table 12. The information collected in the *record* phase must enable the debugger to eliminate the potential of non-deterministic behavior resulting from existence of multiple threads within each task in the *replay* phase. The following conditions must be satisfied for C Thread calls in each re-execution of a distributed program:

- **Threads are assigned the same virtual identification number each time the program is executed.** The virtual ID numbers are needed to uniquely identify each thread in the execution. The virtual scheme of identification guarantees that the names assigned to threads will not change from one re-execution to another. This is required since the name assigned to a thread by the kernel at creation time might be different each time the program is executed.

- **Mutex locks are assigned the same virtual identification numbers in each execution.** Since access to mutex locks is controlled by debugger in the *replay* phase, the mutex locks must by uniquely identified by virtual id numbers that will not change from one execution to another.

- **Mutex locks are obtained by the threads in the same order in each re-execution.** The access to shared variables is assumed to be done through

Cthread call	Data recorded	Explanation
mutex_allocate	- mutex number	The ID number is needed for future references.
mutex_clear *mutex_free*	- mutex number - mutex version	The version number is needed to ensure that the mutex is not cleared/freed too soon.
mutex_lock	- mutex number - mutex version	The version number is needed to ensure the same order of obtaining the lock in next runs.
mutex try_lock	- mutex number - result of the call - version number	The result of the call must be the same in all executions. In case of success the order of access must be also maintained.
cthread_fork	- virtual thread ID	The thread ID is needed for thread identification.

Table 12: C Thread calls

the use of exclusion locks. By controlling the access to the mutex locks, the debugger, in effect, controls access to shared variables. The control is executed by use of version numbers, which are associated with each *mutex_lock* operation. The version numbers, along with mutex identification numbers are stored in the the history file of each task.

- **Mutex variables are not deallocated before all operations (mutex_lock, mutex_unlock) have taken place.** Due to different paces of execution, the thread that deallocates a mutex variable might reach the mutex deallocation instruction before other threads execute all operations on this mutex which were executed in the original execution. For this reason, a mutex version number is stored for *mutex_free*, and *mutex_clear* calls, so the their execution in the replay phase can be controlled.

4.3.3 Network message server calls

Table 13 provides information about data recorded for network message server related calls. For the first two calls (netname_check_in, netname_check_out) we only need to record the outcome of the call (success/failure). Both calls can only be executed once for the same port name, so the task successful in the original execution should also succeed in all replayed executions. The third call (netname_look_up) requires also the information about the identity of the owner of the target port, as well as the virtual ID number within the owners task space. This information is needed for control over the order of messages that will be later send to the port that is being looked up.

Netname call	Data recorded	Explanation
netname_check_in	- value returned by netmsgserver	In the replay phase the outcome of this call must be the same.
netname_check_out	- value returned by netmsgserver	In the replay phase the outcome of this call must be the same.
netname_look_up	- value returned by netmsgserver - ID of the owner task - virtual ID of looked up port	The same value must be returned, the sending task must know the port owner ID and port virtual ID.

Table 13: Network message server calls

The information about owner task ID and port ID recored for the *netname_look_up* cannot be directly obtained from the network message server. To make the information available, DDB must use a "shadow" server of the **netmsgserver** that would store and distribute required information for task and port identification. This is achieved with help of the **CNS** in the following way :

- Each task that checks in a port sends an additional message to the **CNS**. The message contains the ASCII name of the port to be checked in, the task's ID, and the port's virtual ID within the owner task port space. The **CNS** receives and stores the information for later use. Figure 1 illustrates communication between the user task and **CNS**.

- Each task that looks up a port, in addition to making a call to the **netmsgserver**, sends an RPC message to the **CNS** providing the ASCII name of the foreign port to be looked up. In the return message, the CNS sends the virtual ID of the task to which the port belongs and the virtual ID of the port in the task's port space. The communication between the user task and **CNS** is illustrated on figure 2.

The use of the Central Name Server for port identification is necessary only in the *record* phase. During the replayed execution, this information is simply retrieved from the history files where it has been previously stored.

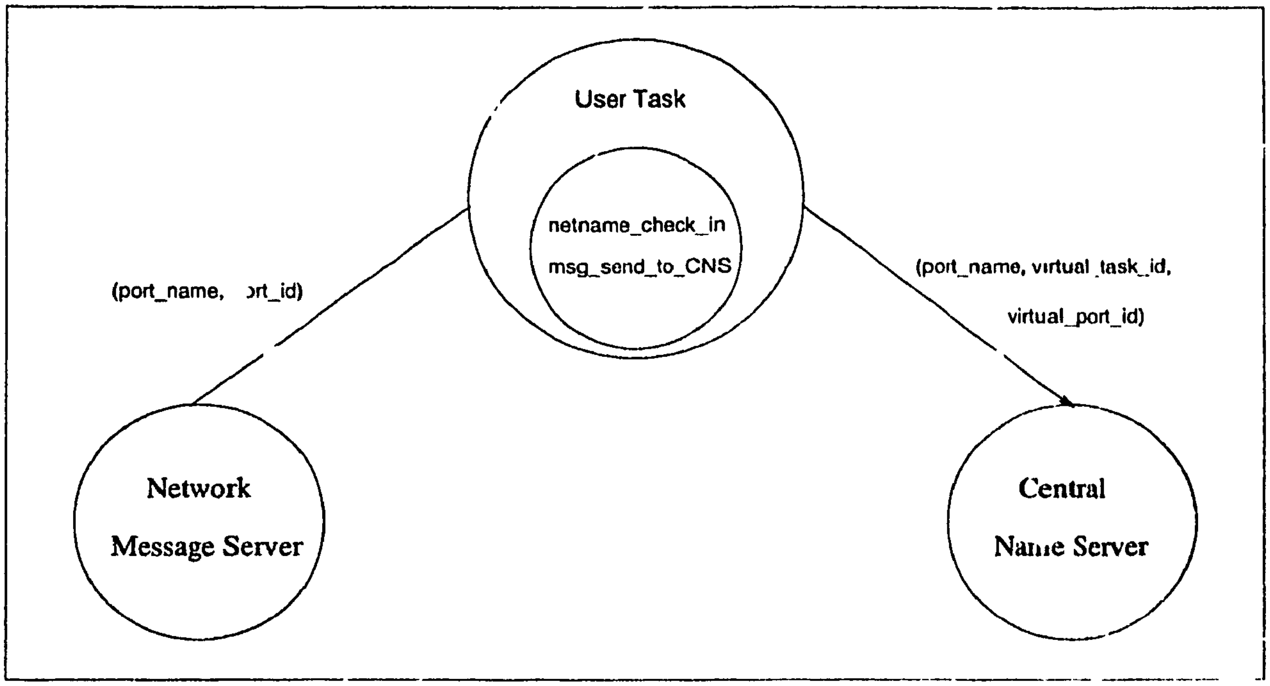


Figure 1: Communication of user task with CNS during check in operation

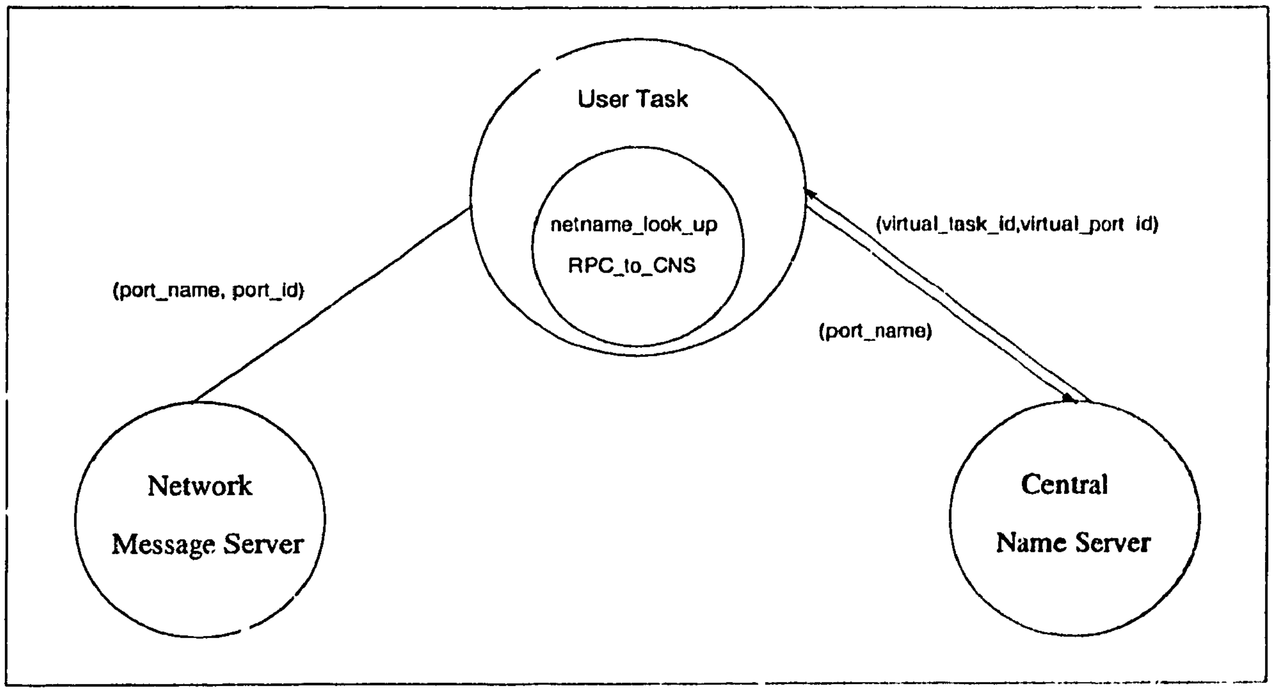


Figure 2: Communi:ation of user task with CNS during port look up operation

4.3.4 Message and port related information

The message passing calls are responsible for the non-deterministic behavior that results from the interprocess communication. The data recorded for message and port related calls must be sufficient to ensure that in the replay phase the following will be true:

- **Ports within each task space will receive the same identification number each time the program is re-executed.** This is necessary because the port names returned by the kernel from the *port_allocate* system calls can vary from one execution to another. The debugger requires that ports be identified by some unique number, which can be guaranteed to be the same in each re-execution. The virtual port names are introduced for this purpose. The virtual port names are stored in the history files when the ports are created in the *record* phase and retrieved whenever the program is re-executed.

- **No port is deallocated before all operations on this port (msg_receive, port_status, etc.) that took place in the original execution are executed.** To make sure that a port will not be deallocated until all operations of the original execution took place, the version number for each port is recorded during the *port_deallocate* call.

- **Each message receive operation will return the same message in each successive re-execution.** Because of the non-determinism of the interprocess communication calls, for each received message the sender's identification number is stored in the *record* phase. This information, along with port version number, is used in the *replay* phase to ensure that messages are received in the correct order.

- **The order of messages sent by the task to a remote port will be identical each time the program is re-executed.** The version number of the port that receives the message indicates how many messages have already been sent from the sending task to this port. This information will help to enforce the same order of *msg_send* operations in the replayed executions.

- **The values returned by the kernel for each listed system call are always the same.** Since the state of the port space is not guaranteed to be

System call	Data recorded	Explanation
msg_receive	- value returned by kernel - virtual ID of receiving port - version of receiving port - virtual ID of the sender	Receiving port must have virtual ID for reference in the replay phase. The version number is needed to ensure that the same threads receive the same messages in replay phase. The sender ID is used by msg_server ensure proper order of messages arriving at the port.
msg_send	- value returned by the kernel - receiving task ID - virtual ID of receiving port - version of receiving port	This information is needed to ensure that messages to the same port are send in the same order each time the program is executed.
port_allocate	- value returned by the kernel - virtual ID of allocated port	The ID number is needed to ensure right order of port creation in next executions.
port_deallocate	- value returned by the kernel - virtual ID of deallocated port - version number of the port	The version number of the deallocated port is needed to ensure that this port is not deallocated until all msg are received.
port_status *port_names*	- value returned by the kernel - all information returned	All original information must be saved to be returned in the subsequent executions.
port_rename	- value returned by the kernel	The same value must be returned in all runs.

Table 14: Message related calls

the same each time the calls are executed, the debugger simply returns the values stored in the *record* phase without executing the calls.

The detailed list of the relevant message related information is presented in table 14.

4.4 How the recorded information is used ?

Previous section presented the program control information recorded and stored in this history files by the DDB debugger. In this section we show how this information is used in achieving deterministic re-execution of distributed programs under Mach.

To ensure deterministic re-execution, there must exist, in the first place, a way of identifying all relevant objects existing within each task. These are threads, ports and mutex locks. The unique identification is necessary so the reference to the same entities can be made in each re-execution. Although we present in detail only the method used for identification of threads, the naming of ports and mutex constructs is done in a very similar way.

4.4.1 Naming Scheme for Threads

The only available thread identifier is the thread ID assigned to each thread by the kernel at the creation time. This identification number changes from one execution to another, so it cannot be used for naming threads in repeated executions. A virtual identification number which would remain the same in each execution is needed.

The virtual naming scheme designed for this purpose can be described as follows:

- In each task space there exists a *thread counter* that is always equal to the number of threads existing within the task.
- Each newly created thread is assigned a virtual thread number equal to the *thread counter.*
- The *thread counter* is incremented after each thread creation operation.
- In the *record phase* the virtual ID of the created thread is recorded in the history file of the creating thread.
- In the *replay phase*, when a new thread is created, the virtual thread from the original execution is obtained from the history file. The data record created for the new thread will contain (among other pieces of information) the virtual ID of the thread and the thread ID returned by the kernel.

```
begin
 if (debugging_phase == RECORD)
       begin
             mutex_lock(thread_lock)
             p = new(thread_data)
             p->kernel_id = thread_fork()
             p->virtual_id = thread_counter
             send_data_to_log(thread_counter)
             INC(thread_counter)
             add_p_to_thread_data_list(p,thread_head)
             mutex_unlock(thread_lock)
         end

       else if (debugging_phase == REPLAY)
          begin
             mutex_lock(thread_lock)
             thread_id = get_data_from_log()
             p = new(thread_data)
             p->kernel_id = thread_fork()
             p->virtual_id = thread_id
             add_p_to_thread_data_list(p,thread_head)
             mutex_unlock(thread_lock)
         end
        end
```

Figure 3: Algorithm for creating threads

- Each time when the virtual identification of a thread is needed, the data record is accessed based on the kernel assigned ID (which can be obtained at any time from the system) and the virtual ID is retrieved.

The algorithm for thread creation is given in figure 3. The thread creation and assignment of the virtual ID number are executed under exclusion locks to ensure binding the correct thread counter value with the created thread. The algorithms for creation of ports and mutex locks are very similar to the one presented in figure 3. They follow the same principle when using different counters, mutex and condition variables, and data structures.

4.4.2 Mutual Exclusion Locks

One of the potential sources of non-determinism in execution of multithreaded programs running under Mach operating system lies in the access to variables shared by threads within each task. Due to different paces of execution, the progress of each thread can be different in subsequent runs and, in effect, the pattern of access to shared objects can be different each time.

The solution to the problem of non-determinism in Mach programs resulting from access to shared data is presented below. We assume that the shared variables in a multithreaded user program are always protected by the mutex constructs. The use of mutual exclusion locks is necessary for correctness of the program since data corruption can result from unsynchronized access to the shared data object. The mutex locks provide the ability to serialize access to shared data.

The main issue here is to ensure that all operations on shared data take place in the same order each time the program is executed. This can be achieved by using a following scheme :

- Each mutex lock variable is assigned a unique virtual number that remains the same across all re-executions of the program.
- Each mutex lock is assigned a version counter.
- The version counter is incremented each time the mutex lock is obtained by a thread in both, the *record* and *replay* phase.
- The virtual ID and the version of the acquired lock will be recorded in the thread's history file during the *record* phase.
- In the *replay* phase, the thread will not attempt to acquire the lock until the version number of the mutex variable will be equal to the version number retrieved from the history file. The conditional variable associated with each mutex variable will be used to block the thread until the right time comes to perform the *mutex_lock* operation.

The algorithm that is used to implement the outlined above method is presented in figure 4. In the *record* phase the record for the mutex is accessed, the lock version number is retrieved and sent to the history file. Then the version number is incremented. All of these operations are enclosed within a mutex lock for protection in case that more than one thread would attempt to access the same record. The mutex variable is specific to each individual record so the granularity of locking is as small as possible in this case. In the *replay* phase the the mutex version is retrieved from the history file. In the case where the version received from the history file and the current version recorded in the mutex data recored are not equal, the thread will block waiting for this condition to become true. After the mutex is obtained, the version lock is incremented and the condition broadcast call is issued for all threads that might be waiting on the mutex condition.

```
begin
  if (debugging_phase == RECORD)
     begin
         p = find_the_node(mutex_var)
         mutex_lock(p->lock)
         mutex_lock(mutex_var)
         send_data_to_log(p->version_num)
         INC(p->version_num)
         mutex_unlock(p->lock)
    end
  else if (debugging_phase == REPLAY)
    begin
         p = find_the_node(mutex_var)
         m_version =  get_data_from_log()
         mutex_lock(p->lock)
         while (m_version != p->version_num)
                condition(p->cond, p->lock)
         INC(p->version_num)
         mutex_lock(mutex_var)
         mutex_unlock(p->lock)
         condition_broadcast(p->cond)
   end
end
```

Figure 4: Mutex locking algorithm

4.4.3 Port Allocation Call

Control over the order of the arrival of messages at each task's port cannot be achieved by the debugger library alone. There must be an external agent that reorders the messages sent to each port and resends them in the order of the original execution. This is done by the Message Server (MS). In order for the MS to be able to intercept messages sent to a user port, the receive rights of the port must belong to the server. The receive rights are exchanged in following way (see figure 6) :

1. Port P is allocated within the user task space.
2. The receive rights of port P are sent to Message Server via IPC message.
3. Port P' is allocated within MS task space.
4. The receive rights of port P' are sent in IPC message to user task.

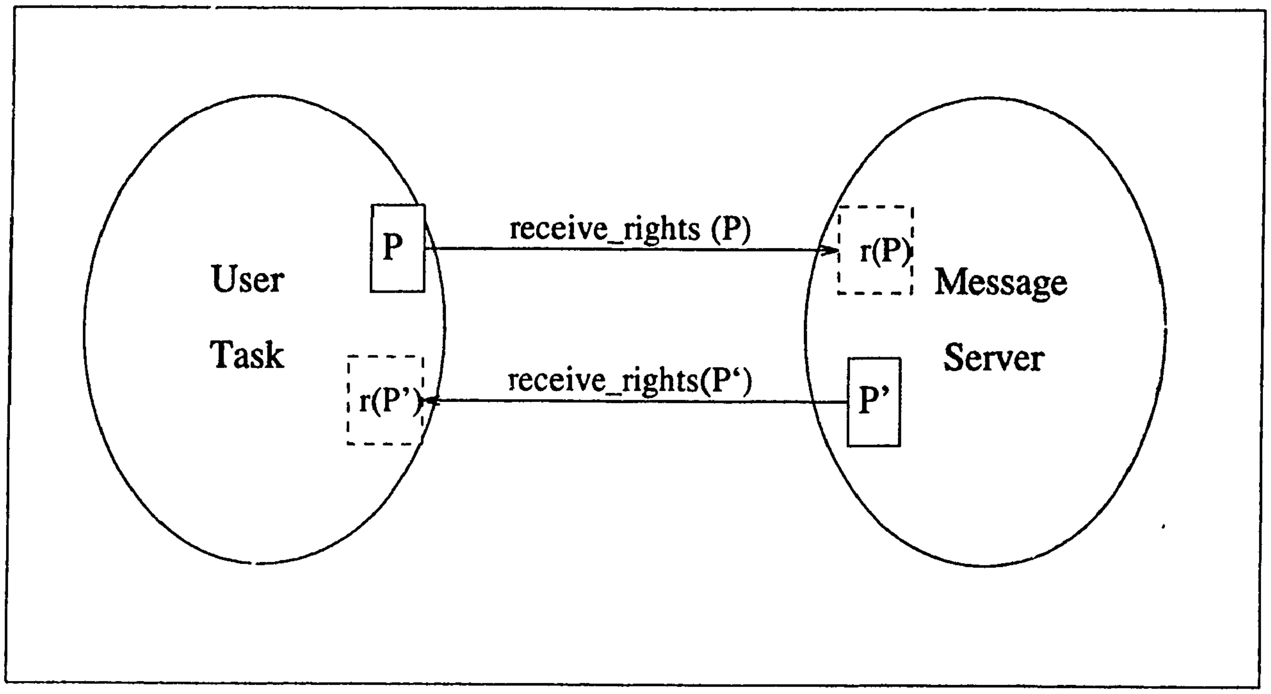


Figure 5: Exchange of port receive rights during port_allocation call

In effect, in the above operation the user task will receive messages on the port P' allocated by the Message Server. The Message Server will be able to intercept the messages sent to the user port P. All of the above is done transparently, so the user task is not aware of the fact that it does no longer poses the receive rights to port P.

4.4.4 Message Receive Calls

To preserve the order of the arrival of messages during replayed executions the virtual ID of the sender and the version number of the receiving port are recorded in the history file for each *message_receive* system call. In the *replay* phase, the information is extracted and the message receive operation executed as follows:

1. If the port version extracted from the history file is not equal to the current version number for this port, it means that other threads are still to execute message receive operation on this port. Block until the right time to perform the receive operation comes.

2. Send a control message to the Message Server asking for message from sender indicated by data obtained from the history file to be received on port P.

3. Receive the next message available on the port P'.

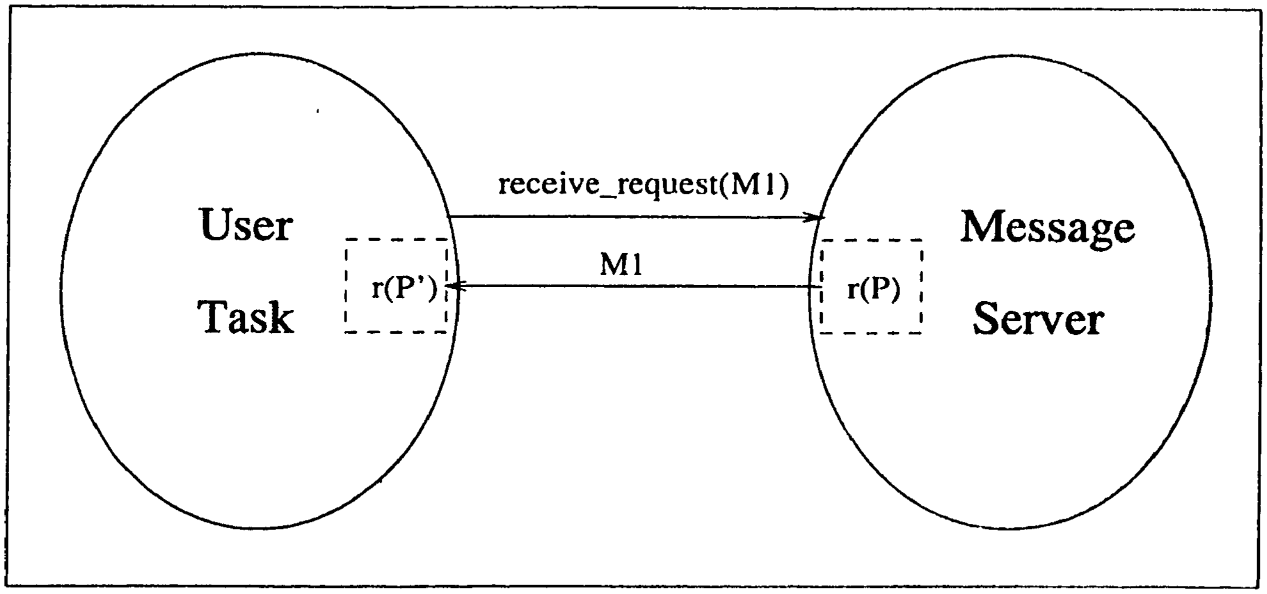


Figure 6: Message receive system call

All of the above operations are executed transparently. The user task executes *message_receive* call on port P, but the port is exchanged by the library routine that was wrapped around the original message receive operation.

4.5 Evaluation of the method

The major concern in deterministic replay systems is the probe effect they introduce in collecting of trace information. There is no way to avoid the probe effect entirely. Any tool that needs to monitor execution of a program will affect this execution in some way and alter it if the potential of non-deterministic behavior exists. The question is then not how to eliminate the probe effect, but rather how to minimize it.

We believe that the mechanism developed for deterministic replay in DDB introduces minimal probe effect for the environment which the debugger supports. The recording of version numbers for shared variables introduces much less probe effect than recording the contents of the variables since the volume of the data to be stored is smaller. The interprocess communication is handled in similar way to the access to shared variables; the information that is recorded for each message is the identity of the sender task and the version of the receiving port. Therefore, there is no overhead from storing contents of messages.

Chapter 5

Design of DDB

This chapter presents design of the distributed debugger named DDB based on deterministic replay. The first section presents the programming environment by describing the Mach operating system, the C-Threads package, the local debugger (GDB), and the Mach Interface Generator (MIG). This is followed by a description of the architectural design and modules of the DDB debugger. The implementation notes close the chapter.

5.1 Programming environment

The distributed environment described in this thesis consists of a set of four Intel *i486* hosts connected into a local area network by the Ethernet and running Mach.

The user program can execute on any number of available hosts. There can be one or more tasks running on the same computer. The only restriction on the network is that all hosts on which the computation is executed must belong to the same network partition. This is necessary because the message server provided by Mach operating system (netmsgserver) does not operate over network bridges, so message passing between two tasks executing in different network partitions would not be possible.

The distributed nature of the debugger is illustrated in figure 7. It contains a centralized part and several distributed parts. The programmer controls DDB from the host running user interface of the debugger and this forms the central part. The remaining parts of the debugger execute on different hosts along with the user tasks and communicate between the user task and the user interface part.

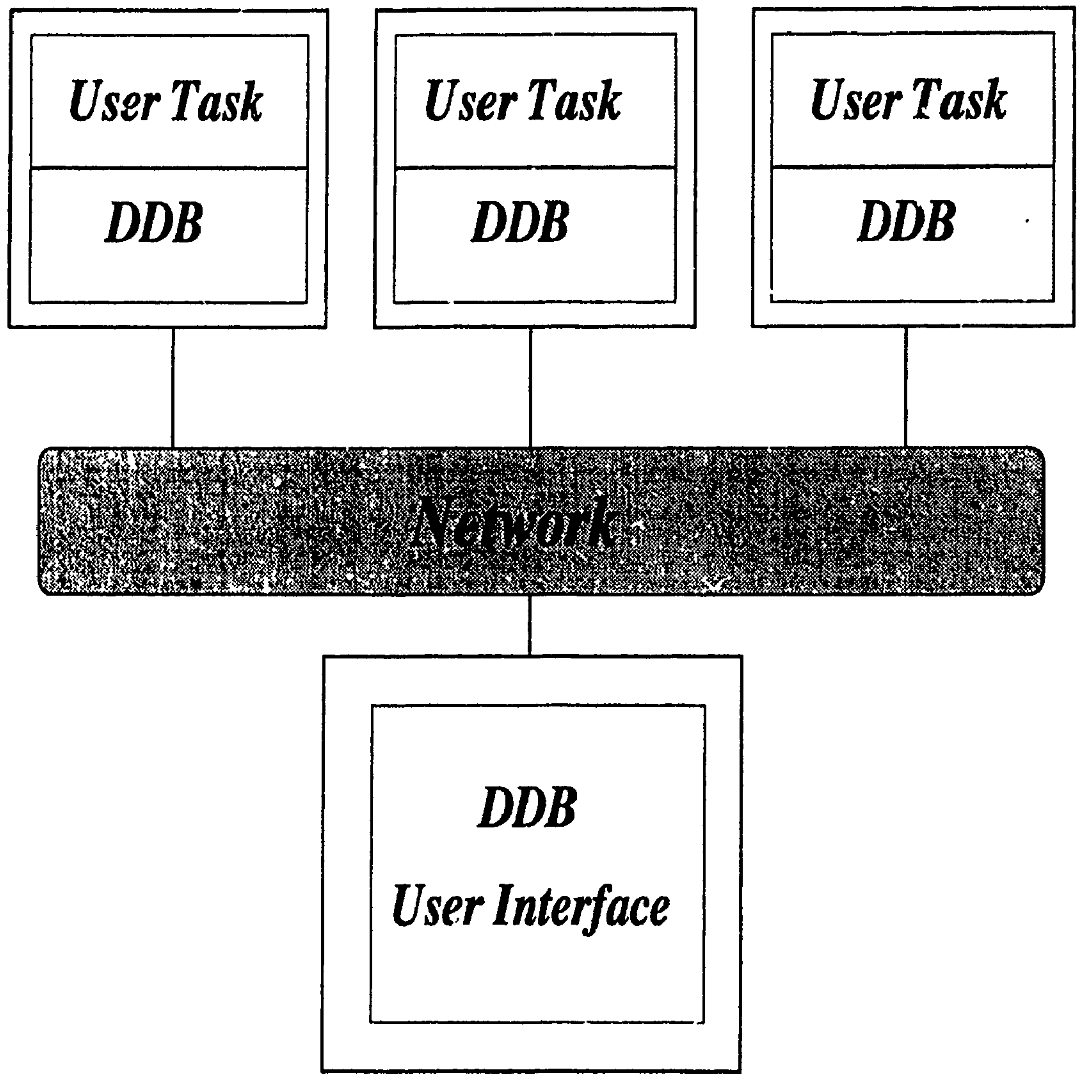


Figure 7: Distributed debugging model

5.1.1 Mach Operating system

The Mach operating system, developed at Carnegie Mellon University (CMU), incorporates many innovations from the operating system research area. The key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware. The main design principles of Mach can be described as follows [42]:

- Support for diverse architectures, multiprocessors and uniprocessors, with message passing or shared memory communications.
- Ability to function with different inter-computer network speeds (wide area networks, local area networks, multiprocessors).
- Simplified kernel structure, with small number of abstractions which are general enough to be used as a base in the development of other operating systems.
- Distributed operation and network transparency.
- Heterogeneous system support that will make Mach available and interoperable among computer systems from multiple vendors.

A very important factor which added to the success of Mach is its compatibility with the Unix operating system. Programs written for 4.3BSD can run on Mach with no changes, although they do not take advantage of the features unique to Mach.

The small Mach kernel, called *microkernel*, has been used as a base for developing emulations of other popular operating systems. The high level view of Mach architecture is presented on figure 8. The *microkernel* contains only basic functionalities like task and thread operations, IPC (Inter Process Communication) virtual memory primitives, and scheduling. The top layer of the figure shows other operating systems that run on the Mach *microkernel*. These systems operate in user mode.

5.1.2 C Threads Package

C Threads is a C language interface library provided by Mach designers to facilitate creating multithreaded applications. This interface allows the programmer to create and use multiple threads without need to handle low level details. Use of C Threads library also enhances portability of the multithreaded program. When the basic kernel threads primitives are used, the programmer has to handle the machine dependent

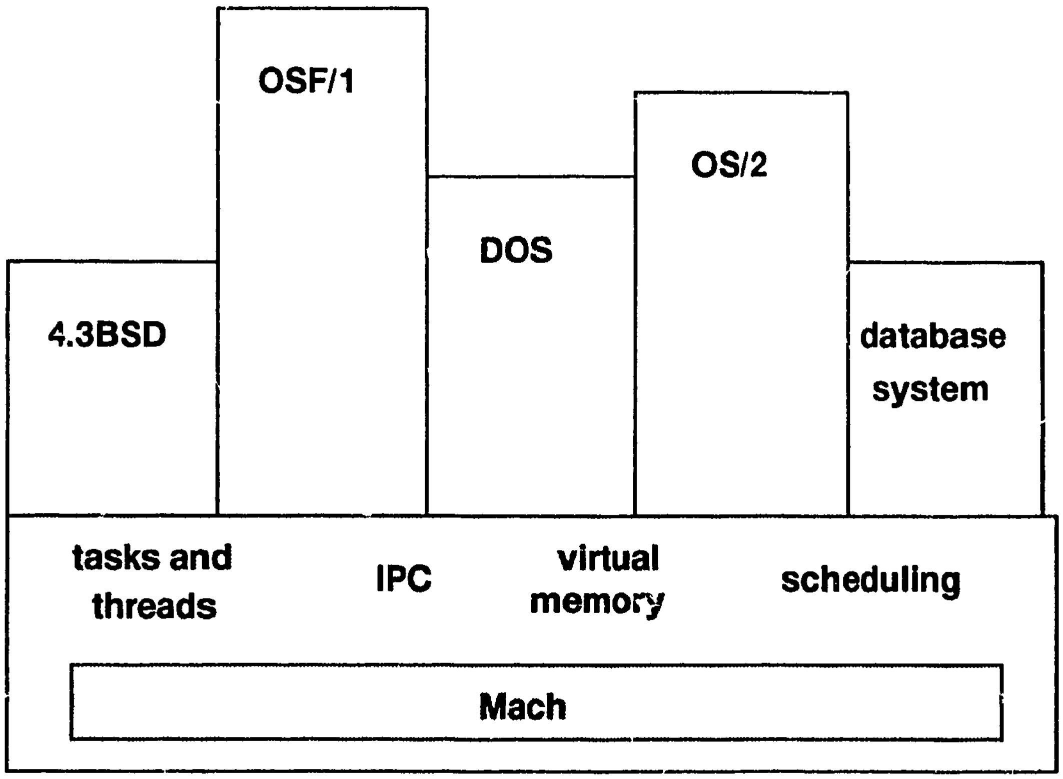


Figure 8: Mach structure

details like machine state of the thread depicted by its registers. The C Threads library provides machine independent function calls, so any multithreaded application can be easily ported to different hardware platforms. Another benefit of using this library package is the availability of the synchronization primitives for controlling access to shared data by multiple threads.

The C Threads library has been extensively used during implementation of the DDB debugger since all servers of the debugger are multi-threaded. We also make the assumption that the user programs handle thread related operations through the C Thread package. This assumption is particularly important in the case of synchronization variables, since DDB monitors the access to shared data through access to the C Threads by synchronization primitives.

5.1.3 Local debugger

The idea of introducing a local debugger into the structure of a distributed debugger has been demonstrated in the past in the design and implementation of the CDB debugger [52] [28]. The local debugger used in CDB as well as in DDB is an extension of the GDB debugger from Free Software Foundation.

GDB debugger

The original GDB debugger has been designed to debug single threaded applications running under Unix operating system. It provides good set of facilities for conventional debugging:

- **Breakpoints.** They can be used to make the program stop whenever a certain point in the program is reached. Breakpoints can be set at a line number, offset from the current line number, function name, or address in the program. Conditional breakpoints take effect when the expression in the condition statement evaluates to true. For each breakpoint, there can be different conditions specified in order to control the behaviour of the breakpoints. For example, a breakpoint can be disabled after it stops the program.

- **Watchpoints.** This facility can be used to stop the program execution whenever the value of an expression specified changes, without having to predict the particular place where this may happen.

- **Stepping.** It allows for executing just one "step" of the program, where "step" can be defined as one procedure, a few lines of source code, one line of source code, or one machine instruction.

- **Stack Examination.** Stack examination facility provides information on the stack frames. This information includes address of the frame, addresses of the called and the caller frames, arguments of the frame, and values of the local variables.

- **Source File Examination.** Parts of the program source code can be displayed by the debugger. Also there are commands to map source lines to program addresses and to display a range of addresses as machine instructions.

- **Data Examination.** Examination of data involves displaying values of program variables, displaying data in memory at a specified address, evaluating "debugging expressions". Values once displayed are saved in the GDB value history so that they can be referred to in other expressions.

- **Symbol Table Examination.** Commands which belong to this group allow the programmer to inquire about the symbols (names of variables, functions and types) defined in the program.

- **Altering Execution.** The execution of the program can be altered by changing the values of variables, memory location, or instruction pointer when the execution stops on a breakpoint.

Modified GDB

The single threaded version of the Unix GDB debugger has been modified by Caswell and Black [8] to support multiple threads of Mach. The resulting enhanced GDB debugger (version 3.4) allows user to select any thread and analyse its execution during a debugging session. The designers have adopted the concept of the *current thread* to handle the selection of threads to be examined. Any thread can be selected as the *current thread* and the selection can be modified during execution. All the GDB commands that are used for examination and modification of a Unix process can be applied to the *current thread.* Some modifications to these commands include change of the *single step* command to step only the current thread while preventing all the other application threads from running.

This modified version of GDB was adapted for the special needs of distributed debugging by C. Yep. [52]. The interface of the debugger was changed so that it uses IPC messages for communication instead of standard input and standard output. The advantage of this adaptation is that the debugger can now receive commands from a process executing on a remote host.

Although both CDB and DDB debuggers use this modified GDB debugger, the role that the local debugger plays is different in each case. In the CDB the local debuggers were used to control the execution of the user program and support the development of facilities such as distributed breakpoints, checkpoint and rollback, and vector clocks. The local debuggers are used in DDB to provide source level debugging for each task participating in the computation during the *replay* phase of debugging. The GDB is available for the programmer to extract any information about the task that s/he needs to understand the behaviour of the program. The place of the local debuggers in the DDB structure is illustrated in figure 9. There is one local debugger attached to each task in the distributed program. The programmer communicates with the local debuggers through graphical user interface at the central debugging site.

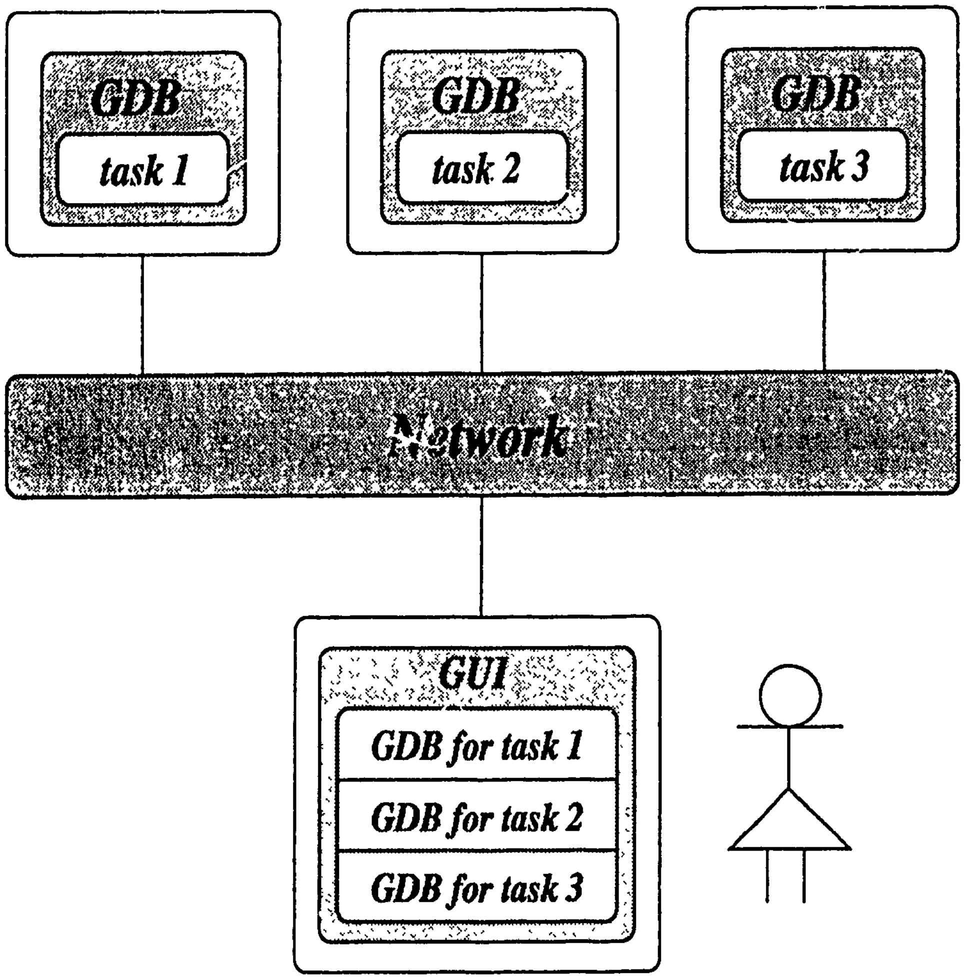


Figure 9: Local debuggers in the structure of DDB

5.1.4 Mach Interface Generator

MIG (Mach Interface Generator) is a facility which provides code for communication between programs that follow the client/server programming paradigm. In Mach, servers and clients execute as separate tasks and communicate by sending Mach interprocess communication (IPC) messages. Because the IPC interface is rather complex, the MIG facility was designed to make programming easier. MIG automatically generates procedures in C which pack and send, or receive and unpack the IPC messages used for interprocess communication based on the specification file provided by the programmer.

The MIG program has been used in the implementation of the debugger to provide interfaces for communication between different debugger servers (history server, message server) and the user task. The DDB programming project has benefited from the use of MIG in several ways. First, time needed to write the IPC code has been decreased. Second, there was no need for debugging, since the generated code was error free. Third, the modifications to the interfaces have been easy to make, since it involved only introducing changes to the MIG specification file and recompiling the code.

5.2 System Architecture

This section presents the system architecture of the DDB debugger. The brief description of the architectural design is followed by presentation of the layered structure of the system. All of the modules of DDB are described in detail in section 5.3.

5.2.1 Architectural Design

The complete architectural design of the DDB debugger is depicted on figure 10. Almost all of the modules of the debugger operate as separate multi-threaded tasks and communicate with each other via IPC messages. The exception is the DDB Library which is compiled with user task code.

- **User Interface** - provides the user with easy access to all services offered by the debugger. It displays the time process diagrams and textual interpretation of contents of history files.

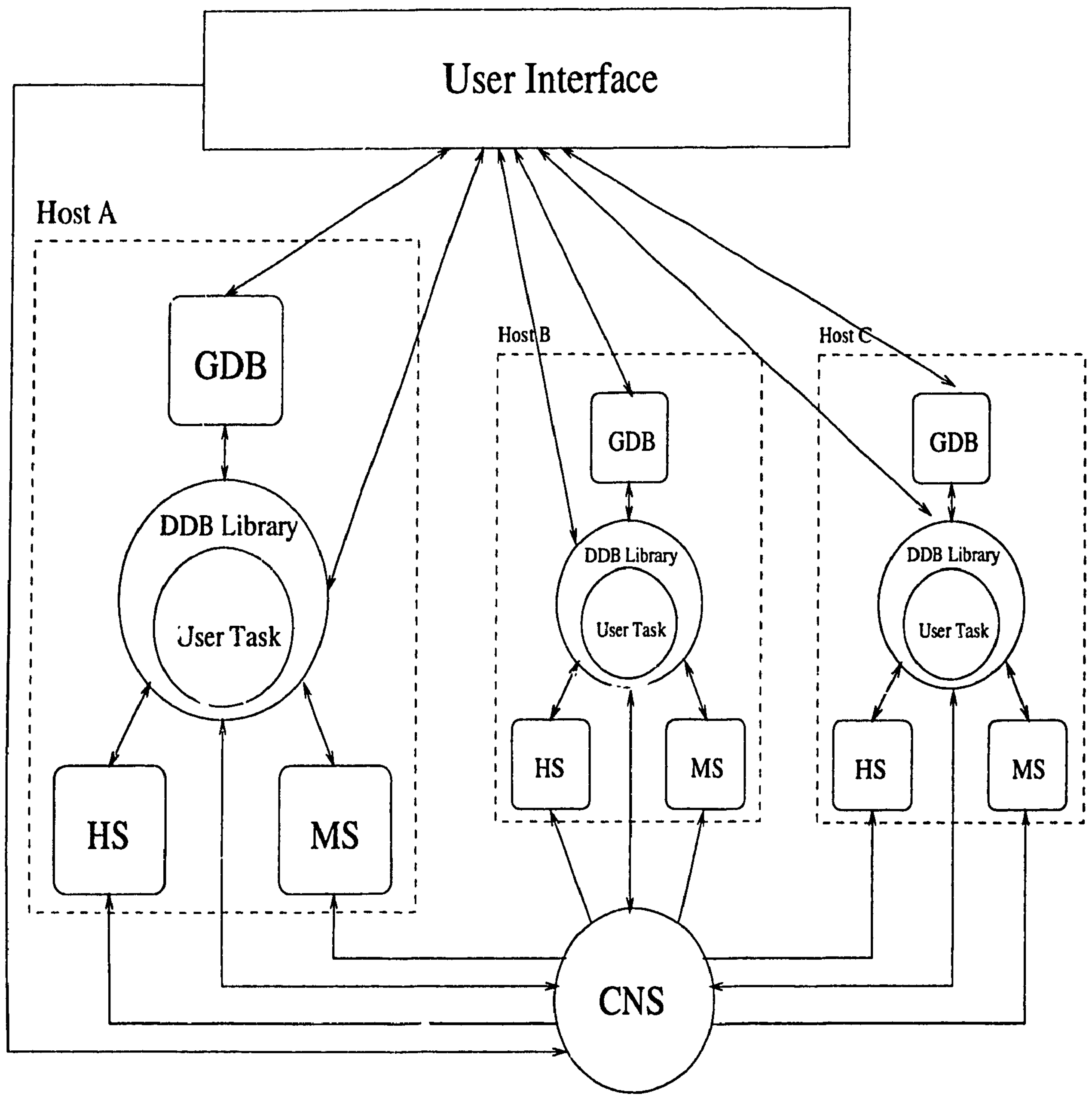


Figure 10: DDB system architecture

- **GDB** - the Modified GDB debugger is used for source level debugging during the *replay* phase. GDB is not invoked in the *record* phase.
- **DDB Library** - provides control over execution of Mach system calls and communicates with other parts of the debugger.
- **HS** - History Server maintains the control information related to execution of the user task. It stores the data received in *record* phase. In the *replay* phase it provides the data when it is requested.
- **MS** - Message Server is invoked only in *replay* phase. It intercepts messages sent to the user task ports and resends them in the order of the original execution.
- **CNS** - Central Name Server distributes identification numbers to the tasks participating in the computation in the *record* phase, and starts the server processes (HS, MS) for each user task.

The distributed architecture of the debugger follows the structure of the user program. Except for the User Interface and the Central Name Server, all the debugger modules are invoked for each user task in the application program. The History Server, Message Server, and GDB are always invoked on the same host as the user task to make communication as efficient as possible. In ther case where there is more then one task executing on a host, the debugger servers will be duplicated, since each server can provide services to only one task. There is no communication among servers belonging to different tasks nor among different servers of the same tasks.

5.2.2 Layered Structure

The layered system structure is shown in figure 11. A horizontal line separates a client from a server, the dotted areas indicate that there is no connection between subsystems.

The bottom layer is provided by the Mach kernel, on top of which there are the debugger servers reside which give services required by the User Interface and the DDB Library. The debugger library uses CNS, MS, and HS to monitor and control the application program. The User Interface is based on the popular X Window system and uses services provided by the modified GDB, and the DDB Library to access debugger facilities.

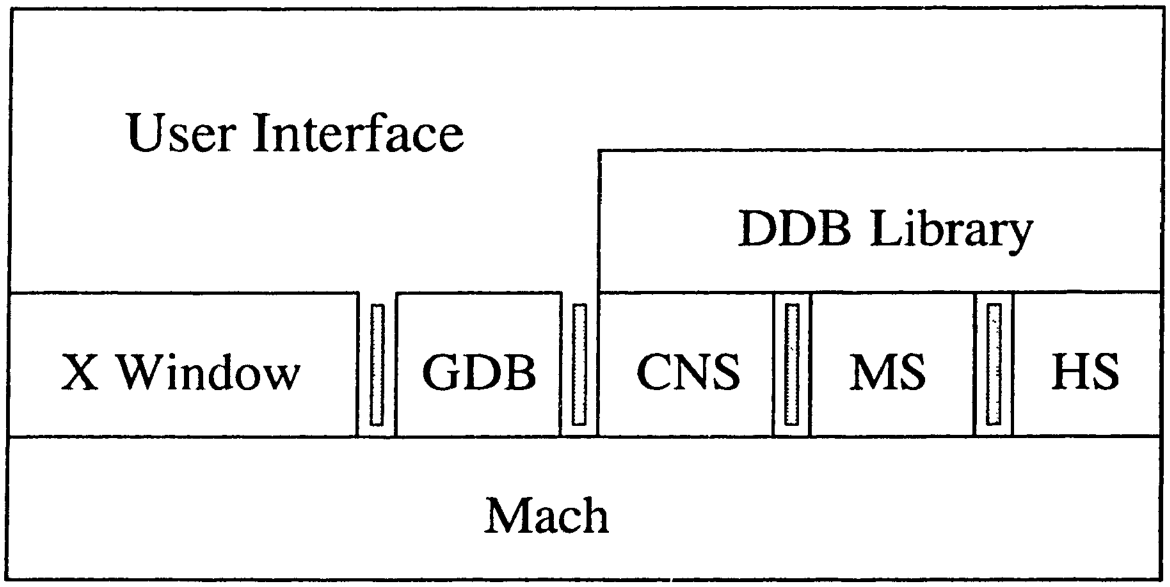


Figure 11: Layered structure

5.3 Modules of DDB

This section describes modules of the DDB debugger. According to the distributed model of computation, all modules discussed below operate as independent tasks and communicate with each other by exchanging Mach messages.

5.3.1 Central Name Server

Since the modules of the DDB debugger are distributed and can execute on different hosts along with tasks of the application program, there must be an agent that coordinates all entities of the debugger. This role is played by the Central Name Server (CNS). The services of CNS are required in both Record and Replay phases:

- CNS assigns unique identification numbers (names) to all tasks participating in the computation. The names are used in recreating the same interprocess communication patterns in successive re-executions of the user program. For example, the task identification number is attached to each message sent by the task.

- Starting and terminating the debugger servers: At the beginning of the computation the initialization routine provided by the DDB Library sends a message to CNS to inform it that the user task has been created. In response to this message, the CNS starts the servers that are required to support the deterministic replay (History Server, Message Server). At the end of the computation each user task sends a message via the library exit routine which requests that CNS terminate the servers.

- CNS provides information about ports that are checked in and looked up with the netmessage server. During netname look up and check in operations the DDB Library delivers/requests information about IPC ports registered with the network message server.

The design of CNS takes advantage of multi-threaded capabilities of Mach and uses multiple threads to achieve short response time for client requests. There is no blocking required for any service, and the communication between the CNS and other modules is infrequent so the overhead introduced by this central server is not substantial.

5.3.2 DDB Library

The DDB library consists of set of routines that are linked with the executable file of each user task. The library routines are used for gathering execution time trace information and controlling the application program. Functions performed by DDB library are as follows:

- Initialization, which include sending server creation request to the Central Name Server (SCR message) and setting the initial values for the library data structures. These tasks are performed by function RR_INIT, which must be inserted into user code before any executable statement.

- Sending the control information to be stored in the History Server in the *record* phase. The information is sent after the execution of a Mach system call.

- Retrieving the control information from the History Server in the *replay* phase. The control information is requested before the execution of a Mach system call.

- Sending control messages to the Message Server with requests for two different actions:

  - Port receive rights exchange (RRX message) request is sent after a port allocation in the user task space.
  - User message receive (UMR message) request is sent when a thread in the user task is ready to receive an IPC message.

- Enforcing correct order of execution of *mutex_lock* calls and *msg_receive* calls in the *replay* phase. The required order is indicated by data received from the History Server.
- Sending server termination request (STR message) to the Central Name Server at the end of the computation to inform the CNS that the servers belonging to the task can be terminated.

The DDB Library routines that replace Mach system calls in the user code always consist of two parts. One part is executed in the *record* phase and another in the *replay* phase so that there is no need to create separate executable files for each phase.

5.3.3 Modified GDB

The GDB debugger and the services it can provide are described in section 5.1.3. Within the DDB debugger the GDB plays the role of the local debugger. The local debugger is used in *replay* phase of the debugging for extracting the source level information, which can be used to refine the high level view of the execution provided by the time-process diagrams.

5.3.4 History Server

History Server (HS) manages control information used by DDB in achieving deterministic replay. There is one HS for each user task, the server is always started on the same host as the user task to make the communication overhead as small as possible. The operation of the server is different in two debugging phases:

- In the *record* phase, the history server receives messages send by the DDB Library. The contents of the messages are stored in server data structures as they are received. At the end of the computation the control information is copied to persistent storage (text file).

- At the beginning of the *replay* phase the contents of the file containing control information are copied into memory of the History Server. When the control information is available in memory, the server listens on the control ports waiting for messages requesting the information. When the message arrives, the History server retrieves the next control record and sends it in the return message.

The communication between the user task and the History Server is very frequent since control information is transferred for most Mach system calls. Therefore, it is important to minimize the delays. This is done by the multi-threaded design of the server. For each thread created in the user task there is a corresponding thread created in the History Server that will respond to requests from the newly created thread. In effect, the structure of the History Server task dynamically changes following the changes in the structure of the user task.

5.3.5 Message Server

One of the basic conditions for ensuring deterministic re-execution is that each task receives the IPC messages in an order identical to that of the original execution. The deterministic replay system must contain an agent that makes it possible. In the DDB debugger role of this agent is played by Message Server. The main function of the Message Server is to control messages received by user task by intercepting them and re-sending to the receiving port when the "right" time comes. As figure 12 illustrates, there is one copy of the Message Server for each user task, residing on the same host. Each message sent by task T1 to task T2 is intercepted by Message Server MS_2, and conversely, each message sent by task T2 to task T1 passes through server MS_1.

The services of the Message Server are required only during the replay phase of debugging. In the record phase, the order of the arrival of messages must only be recorded and this is achieved by sending relevant information to the History Server. The information record sent to the History Server for the receive operation (see table 14) contains, among other data, the identification number of the sending task. Because Mach kernel does not provide any facility for identification of the sender of a message, the debugger transparently adds the sender's identification number to each message. On the receiver's side, the debugger extracts the sender information and forwards it to the History Server. This information is used in the replay phase to ensure that messages arrive to the receiving port in the original order. When a thread is ready to execute a message receive operation the debugger checks in the history record for the sender's ID and sends request to the Message Server (UMR message) to forward the next message from this sender to the port.

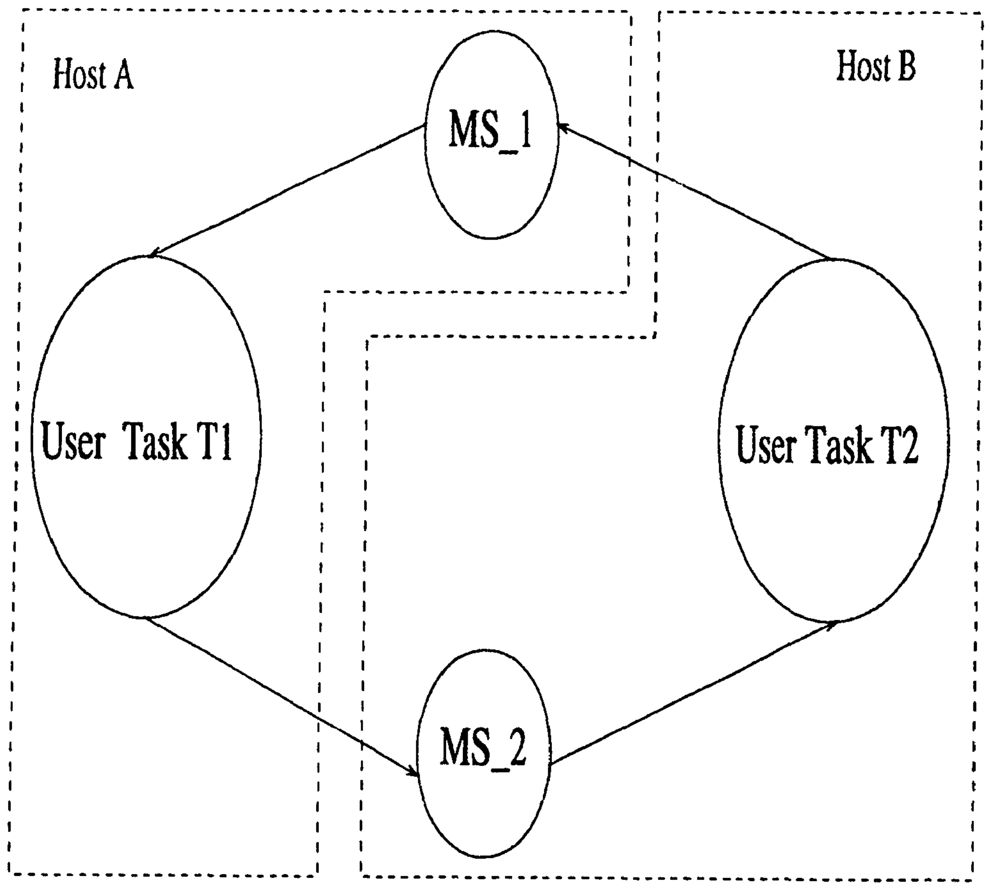


Figure 12: User Task - Message Server configuration

How the Message Server operates

The Message Server operates by responding to IPC messages. There are two types of messages that can be received :

- **Application Messages** - these are the messages that are meant to reach the user task that the Message Server is controlling. The sending user task "believes" that the message is sent directly to the destination task. After receiving such a message the server checks the sender's identification and forwards the message to a temporary port.

- **Control Messages** - these are the messages that are sent by the DDB Library routines of the controlled task. There are two types of control messages :

  - **Receive Rights Exchange Request (RRX)** - this is a RPC message which is sent when a new port has been allocated by the user task. The message contains receive rights of the newly created port. The MS allocates a new port and sends the receive rights of the new port to user task. The receive rights of the user task and server port are exchanged and interception of user messages can proceed.

  - **User Message Request (UMR)** - this message indicates that the user task is ready to perform *message_receive* operation. The message contains the port on which the receive operation is to take place and the identification number of the sender. In response to this message, the MS forwards the first message from the temporary port to the indicated port.

Multiple threads are used to make operation of the Message Server as fast as possible. There is one thread for each port allocated by the user task, and one thread receiving requests for messages.

5.3.6 User Interface

The graphical user interface for DDB was developed with the widgets of the OSF/Motif toolkit. The user interface provides easy access to all features of the debugger. The main window of the interface is presented in figure 13. The menu bar visible at the top of the window contains the following items :

- **Set Program** - allows the user to interactively set the names of the tasks participating in the computation and the names of the hosts that the tasks are to be run on. This eliminates the need for configuration files.

- **Run** - allows the user to start the program in either *record* or *replay* phase. The user starts each task of the program separately so that the appropriate order and delays can be applied.

- **View** - provides high level view (time-process diagram or textual interpretation of the history files) of the program execution based on the control information gathered during *record* phase. The user can choose to view the time-process diagram where each task is represented by events placed in the horizontal line according to the elapsed time. Example can be seen on figure 14. The textual description of system calls executed in the program can be accessed from the "View History Files" sub-menu. It gives more detailed information than the time-process diagram.

- **Quit** - terminates the DDB debugger.

- **Help** - gives access to help facility.

The area below the menu bar is used to display output of the user tasks during the *record* phase. In the *replay* phase it allows for communication with the base debugger controlling each task. During program execution, each task has a dedicated scrollable window. The windows are re-sizable to allow more space for tasks of interest.

The bottom part of the window is used for displaying error messages. This layout of the main window complies with the guidelines of the *Motif Style Guide*.

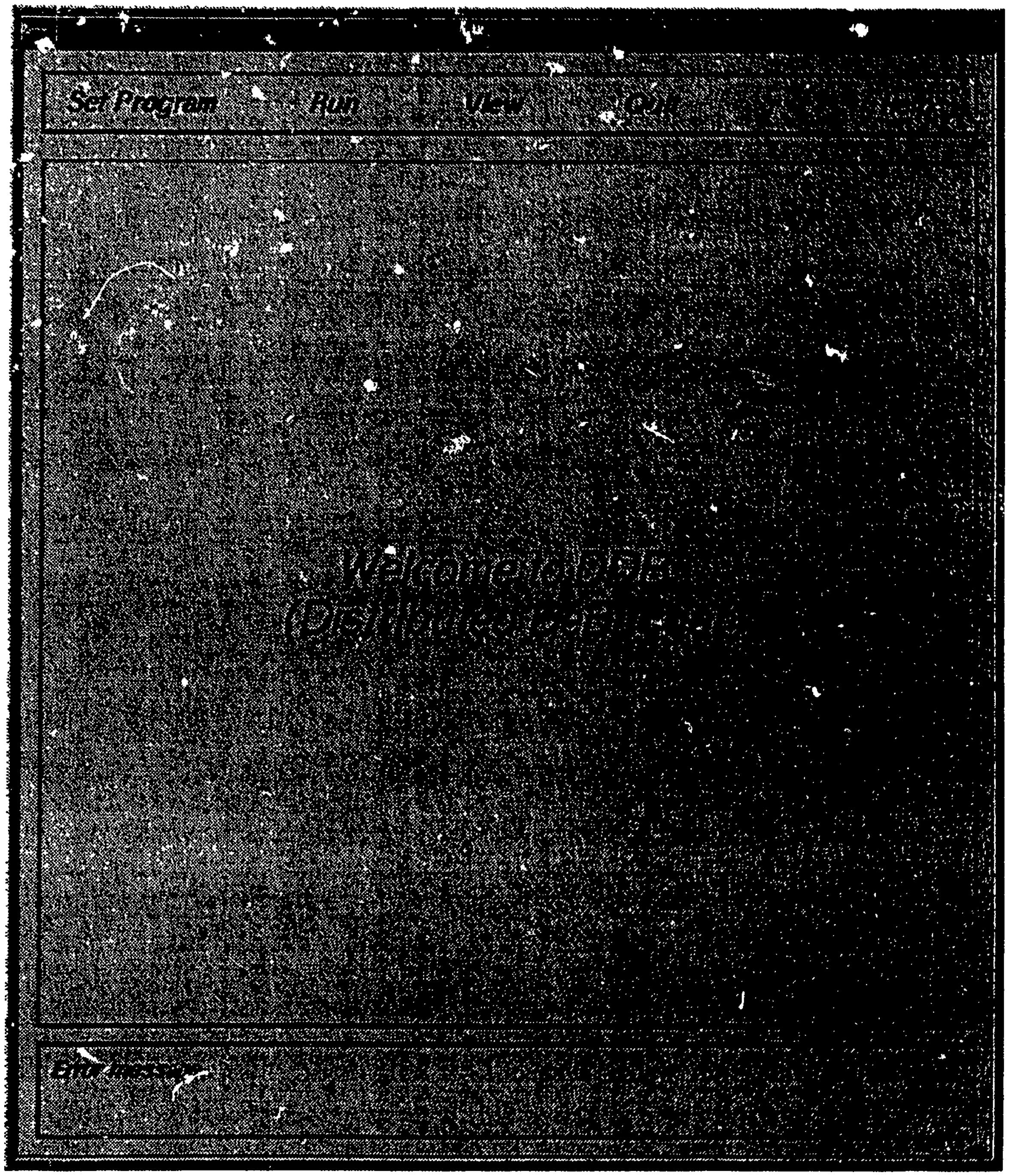


Figure 13: Main window of the DDB

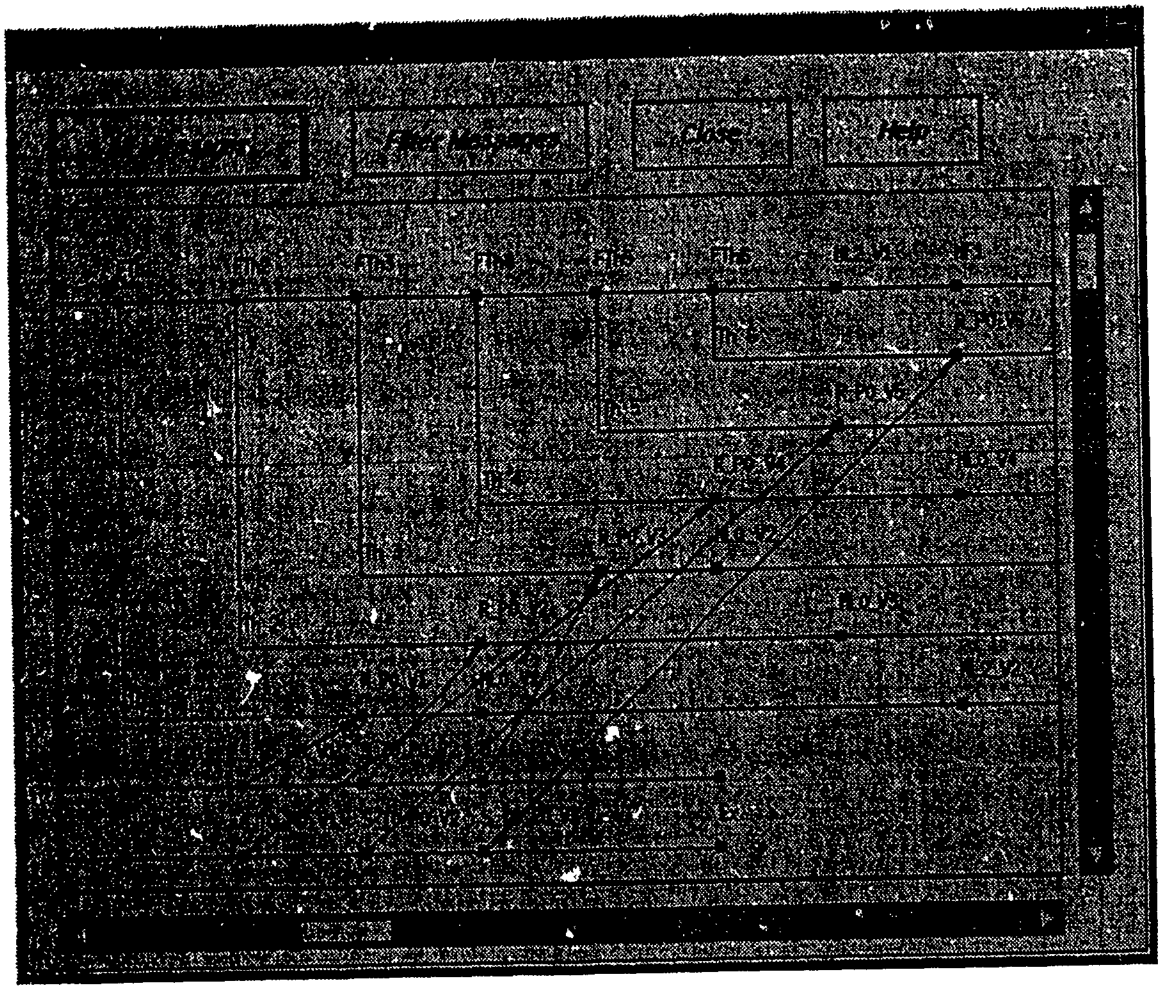


Figure 14: Time-process diagram

The time-process diagram and the information in the text window are based on the control information gathered in the *record* phase. The main purpose of this information is to enable the debugger to produce repeatable executions of the program. The secondary application of this data is in building the high level view of the program. The algorithms that are used in creating the time-process diagrams and the textual descriptions from the history files have been developed by Alexandre Oumanski. Alex has developed both algorithms and wrote some test programs for the debugger during his COMP 492 course during summer 1995.

5.4 Implementation Notes

A prototype implementation of the design described in this chapter has been achieved. The C Threads library was extensively used to handle multiple threads in the debugger servers. The MIG (Mach Interface Generator) was used to implement procedures for communication between DDB Library, History and Message Servers.

The User Interface for the DDB was partly implemented using UIM/X (User Interface Manager for X). Since the UIM/X release for Mach operating system is not available, the design of the User Interface was carried out on the Unix platform, and the generated code was ported to Mach. The difficulty of this approach is that UIM/X library could not be used and often Motif library functions had to be used to replace calls to the UIM/X library.

One of the major difficulties in the implementation of the DDB was integration of the GUI with the computational part of the debugger. The difficulty came from the fact that Motif does not provide any standard way of communication between the "interface process" and the "computational process" when they execute on different hosts. To overcome this problem, the standard Motif communication mechanism based on pipes was augmented with Mach message passing. The communication between the UI and the remote processes was divided into two phases: the transfer of information between remote hosts was via Mach messages, then when the information was available on a local host, it was transferred via pipes.

Chapter 6

Debugging with DDB

This chapter presents how the DDB debugger can be used. This is achieved by description of a sample debugging session. The description is augmented with illustrations of different debugger windows.

6.1 How to prepare user program for debugging

There are several conditions that the user program must satisfy to make debugging with the DDB debugger possible:

1. The `rdb.h` file must be included in each source file of the program. This file contains C preprocessor directives for replacing the Mach system calls with DDB Library calls.

2. Each Mach message structure must contain `DEBUG_MSG` field. The debugger will use this field to transparently add the task ID to each outgoing user message. On the receiver's side, this ID will be stored in the history files (in *record* phase) or used to control order of arrival of messages (in *replay* phase).

3. The line `RR_INIT(argv, argc)` must be inserted before any other executable statement in the code. This function calls the initialization routines that set all the debugger variables and connect to the Central Name Server.

4. The user program must be compiled with the -g flag to let the compiler know that the symbol table is to be created. This is only needed if the GDB debugger is to be used in *replay* phase.

5. The object files of the program must be linked with the DDB Library contained in librr.a. This library contains all routines replacing Mach system calls 'n user code.

6.2 Steps in debugging

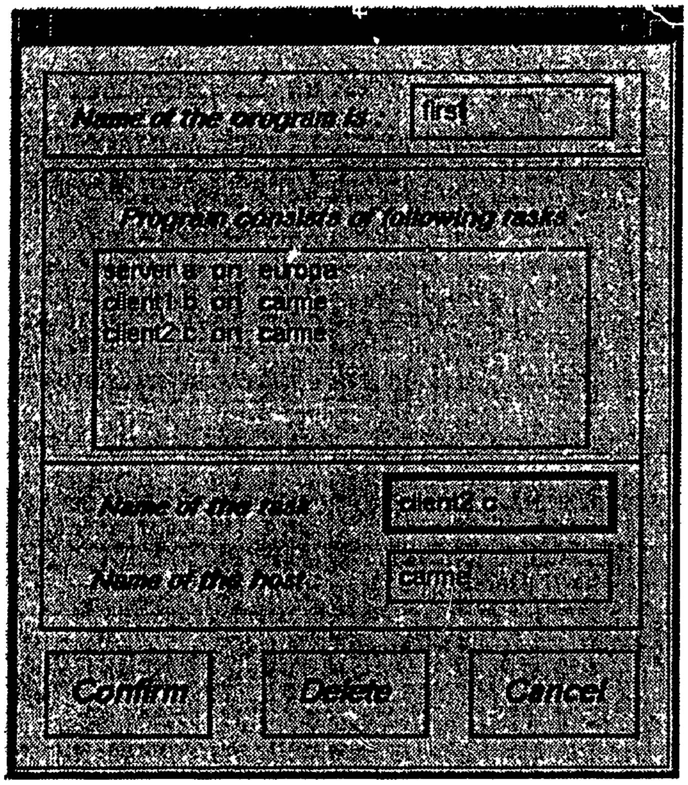
This section gives step by step walk through a debugging session. The user program selected for this demonstration is quite simple. It follows the client-server model with two clients (client1, client2) and server (server). Each client sends three messages to the server. The messages of client2 have larger size than messages of client1. The server creates one thread to receive a client message and service the request. The source code for this program can be found in appendix A.

6.2.1 How to set up the program

To set up the program, select **Set Program/Set New Program** option from the menu bar. A window appears where the program specifications can be entered. Figure 15 shows the window with the complete list of tasks in the program and hosts on which the tasks are to be executed. In our case the program will be executed on two hosts: europa and carme.

The program name appears at the top of the window. This name is required for identification of all history files and debugger servers belonging to one program execution. If two instances of the debugger were to be executed at the same time, the program name would allow for identification of the DDB servers belonging to each program. The program name is also used when the program is to be replayed. If there were several executions of some set of tesks recorded, the program name could be used to identify the appropriate history files. This is important when the user wants to run the program several times in the *record* phase, then examine the information about the executions with the **View** tools, and select one of the recorded executions to be replayed.

The program set up can be confirmed by clicking the mouse pointer on the **Confirm** button. The **Delete** button can be used to delete an entry from the list of tasks in case of an error. The **Cancel** button can be used to close the set up window and abandon the setup operation.



Name of the program is : first

Program consists of following tasks :

```
server a on europe
client1 b on carme
client2 b on carme
```

Name of the task : client2 c

Name of the host : carme

Figure 15: Set Program window

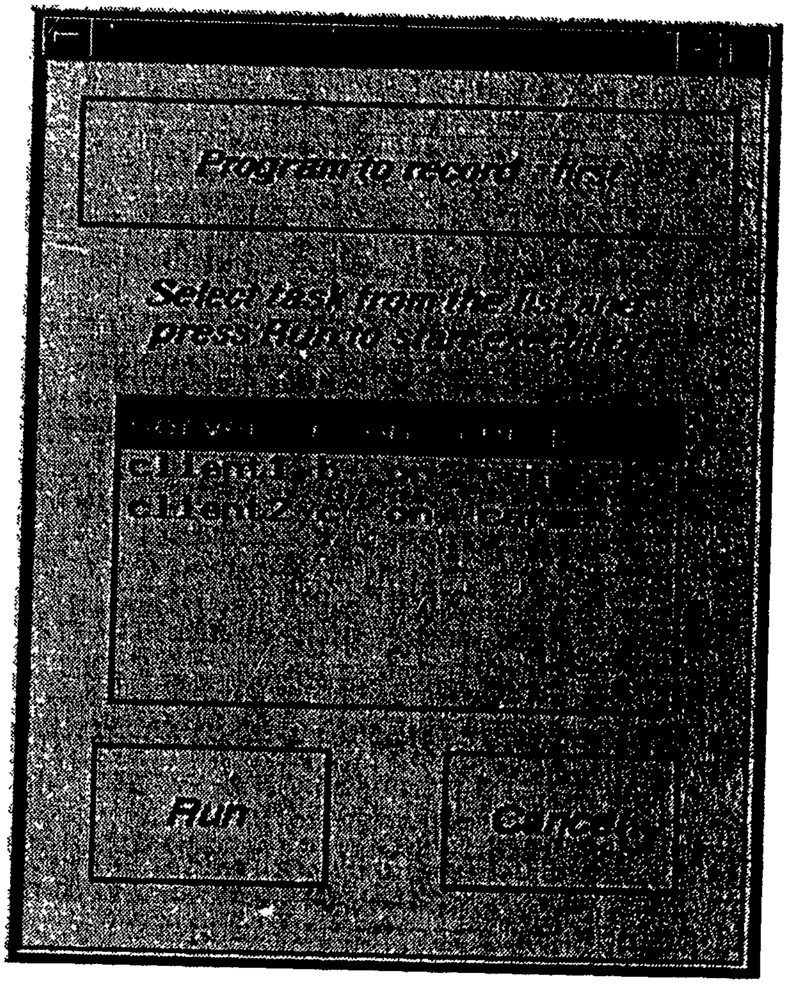
6.2.2 Start the program

To start execution of the user program we have to select the **Run/Record** option from the menu bar. The window depicted on figure 16 will appear on the screen.

The program is started incrementally by initiating execution of each task belonging to the program.This can be done by selecting a task from the list and clicking the mouse on the **Run** button. The task will disappear from the list. There will be a scrollable window created in the main window of the debugger for the newly started task. The output from this task will appear in the scrollable window 17.

After the execution of all tasks is initiated the list of tasks becomes empty and the window disappears.

In case of a mistake in the program setup, the **Cancel** button can be used at any time to abandon the operation and close the window.



Select task from the list and
press Run to start [illegible]

[illegible]
client1 [illegible]
client2 [illegible]

Figure 16: Run Program window

6.2.3 Observe the output

During execution of the program, the output of each task will appear in a task-dedicated scrolled window. The windows are re-sizable to allow for increasing the size of the most interesting window. The output from program *first* can be seen on figure 17. The output indicates that the execution was successful, and that each of the clients has sent three messages to the **server** task. Each of the threads of the **server** task has successfully received one message from a client.

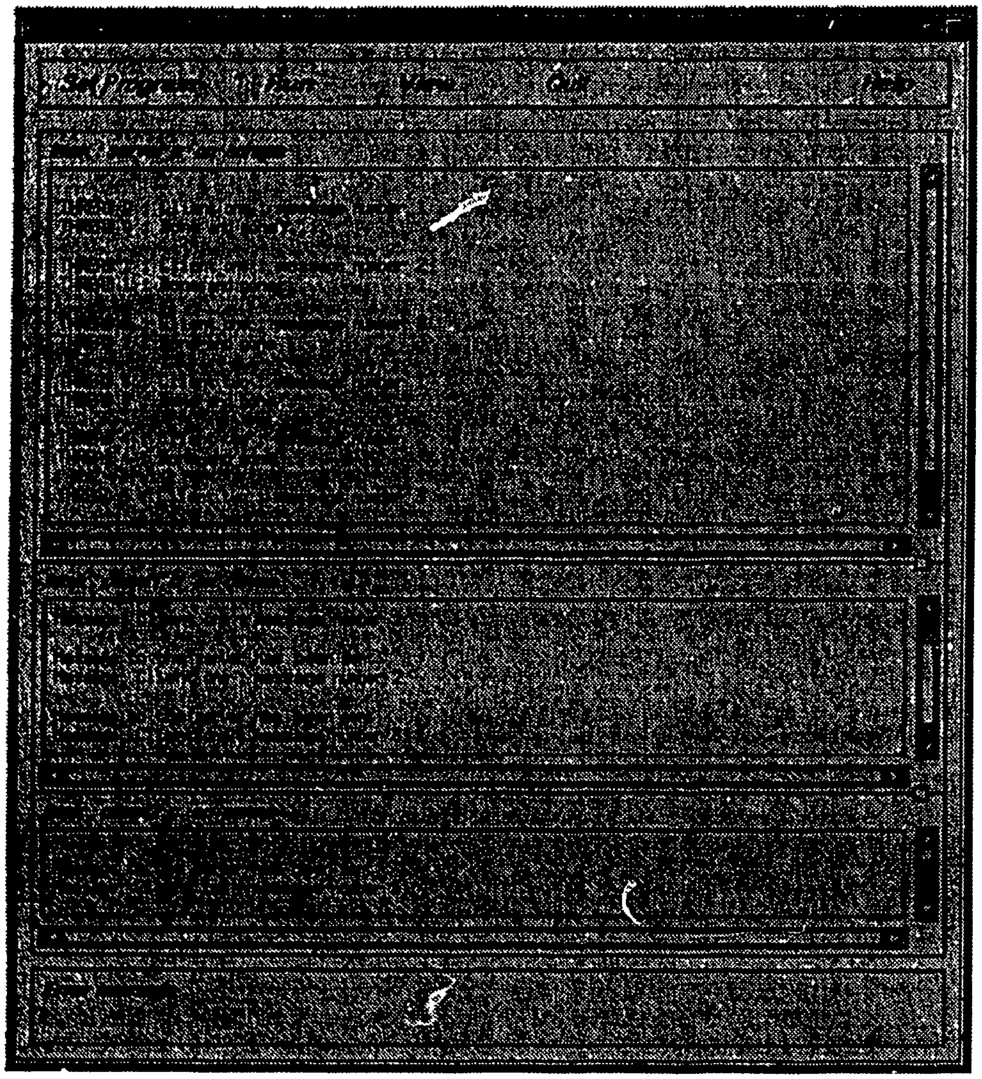
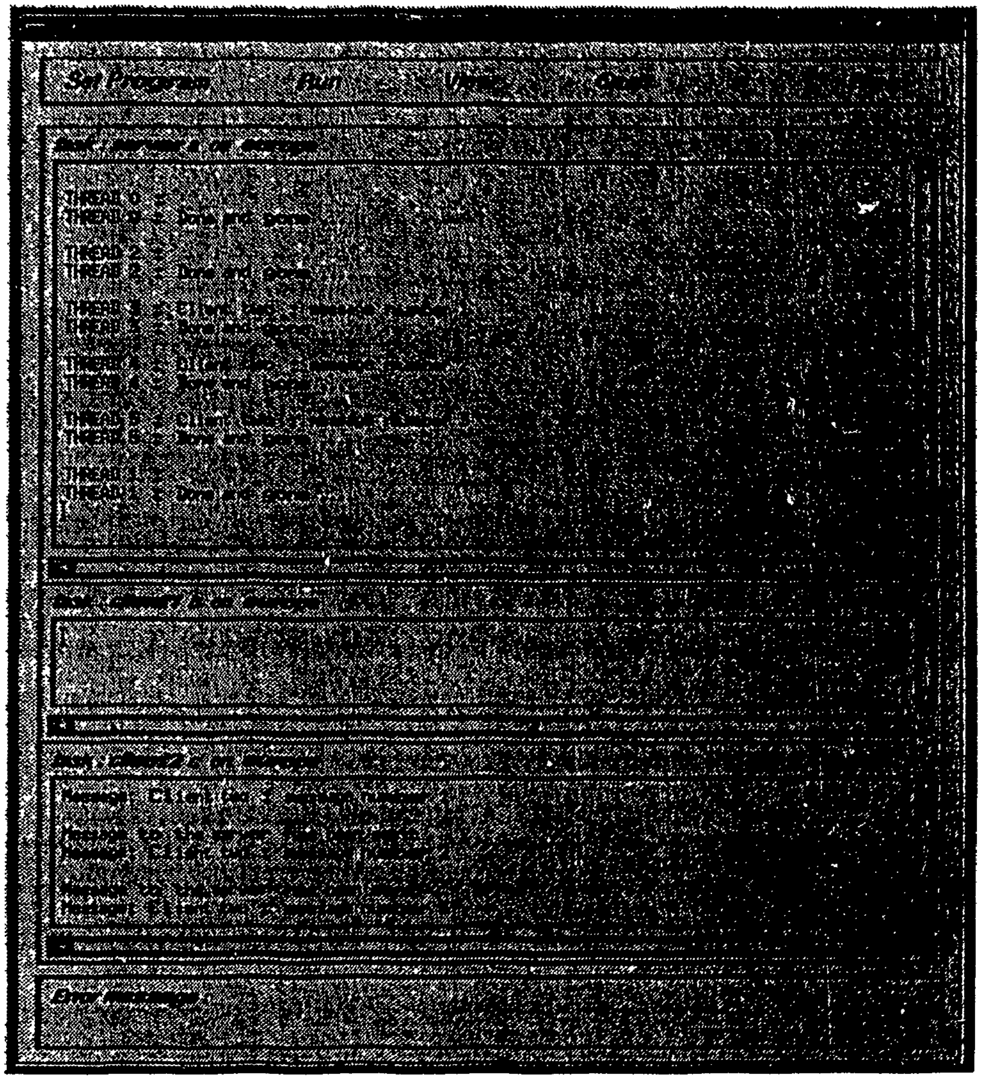


Figure 17: Output from the first run of the program

To verify the expected behaviour of our program we executed it again. We run the program by choosing **Run/Record**, there is no need to set up the program this time since the last program configuration is remembered by the debugger. The output from each task can be seen on figure 18. This time it is not exactly what we have expected to see. It appears that only task client2 managed to send messages to the server. And only three threads of the server reported "successful message receive". What happened this time ? (There is a bug).



```
THREAD 0 - [illegible]
THREAD 0 - Done and Gone
THREAD 2 - [illegible]
THREAD 2 - Done and Gone
THREAD 3 - [illegible]
THREAD 3 - Done and Gone
THREAD 4 - [illegible]
THREAD 4 - Done and Gone
THREAD 5 - [illegible]
THREAD 5 - Done and Gone
THREAD 1 - [illegible]
THREAD 1 - Done and Gone
```

```
[illegible]
```

Figure 18: Output from the second run of the program

6.2.4 View time-process diagram

Since a picture is worth a thousand words we will now take a look at the time process diagram of the second execution. This can be done by selecting **View/View Graph** from the menu bar. A window appears on the screen showing the time process diagram of the current execution as shown in figure 19.

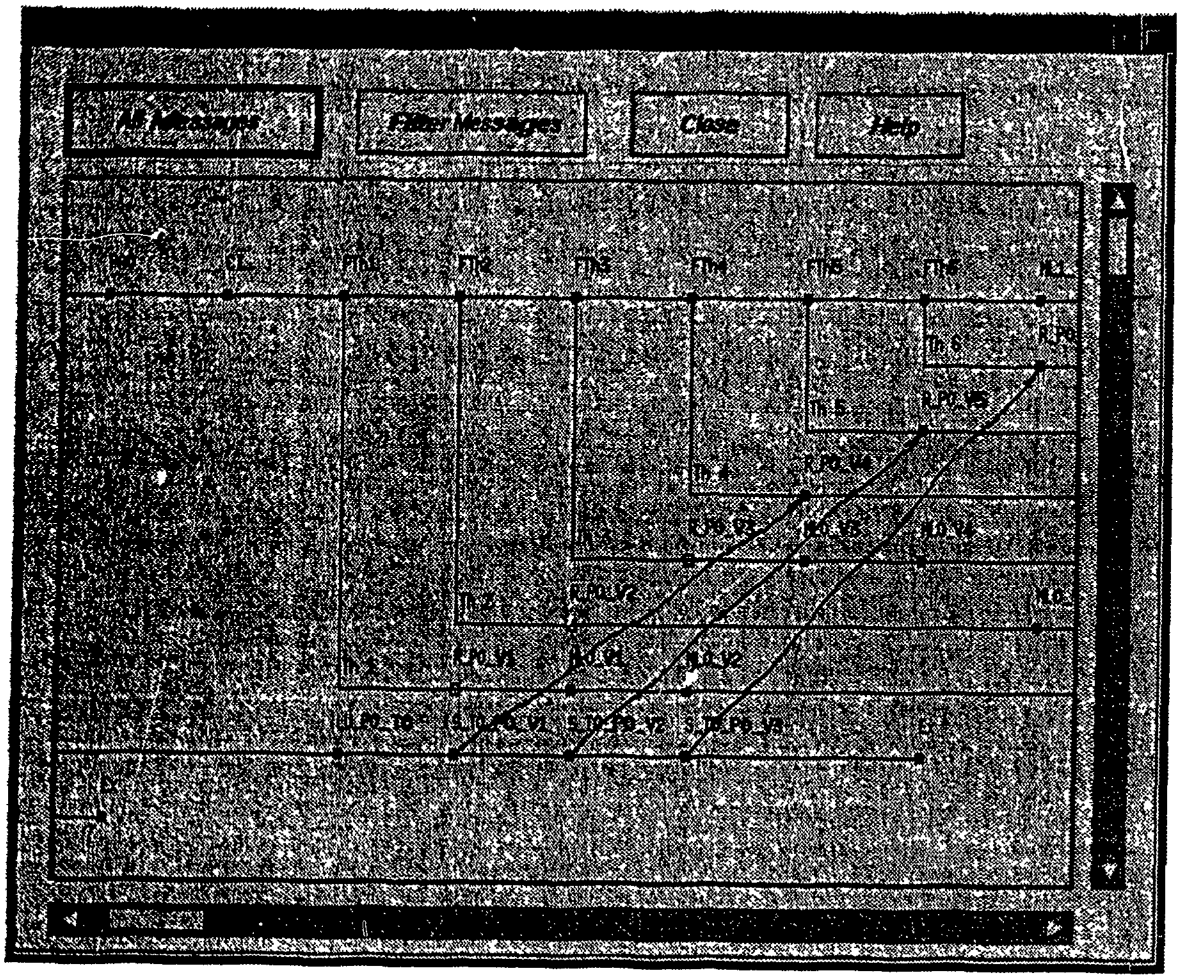
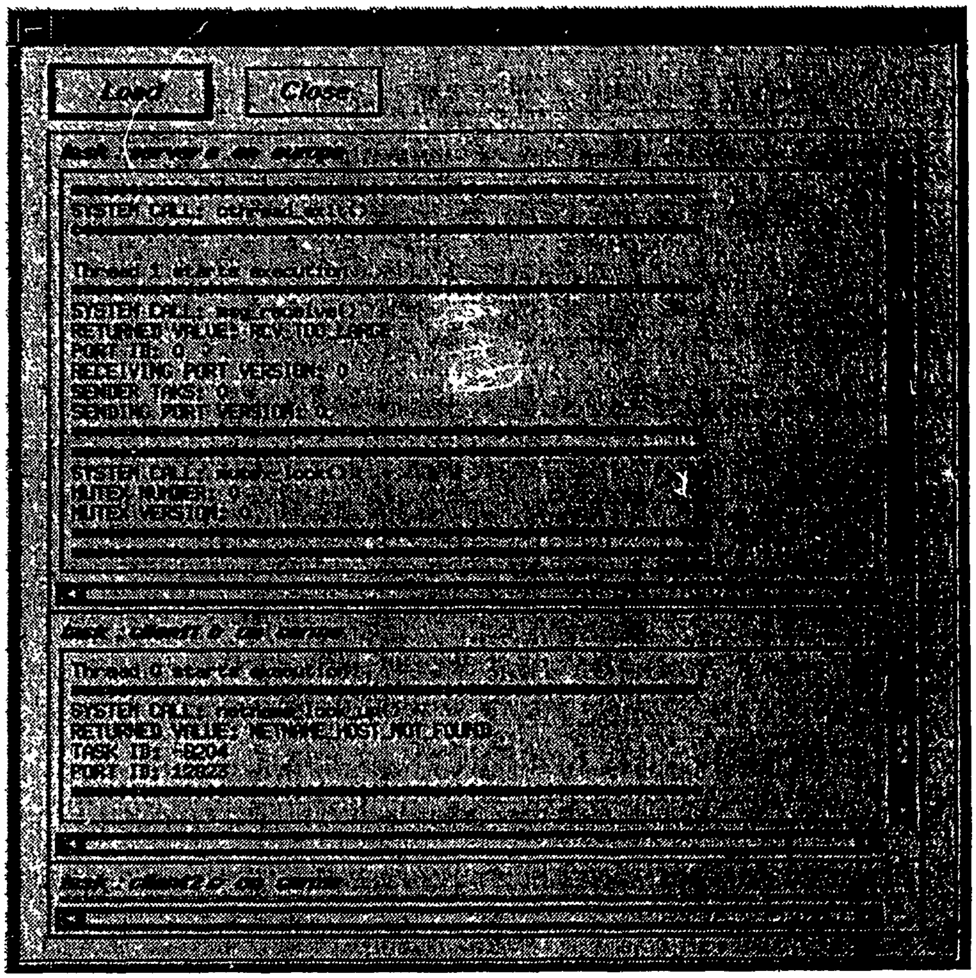


Figure 19: View

The time-process diagram appearing in the View window confirms our suspicions about a bug. Only three of the threads have received messages. For others the receive operation was unsuccessful. Also the client1 task has terminated execution before it could send any messages to the server task. The only recorded system call for this task (not shown on the picture) is the *netname_look_up* operation.

6.2.5 Look at the history files

The time-process diagram does not show the details of system calls to preserve the clarity of the presentation. These details can be obtained by selecting **View/History Files** from the menu bar. A window that appears on the screen (figure 20) contains two push buttons: **Load** and **Close**. The history files can be brought to the window by clicking the mouse on the **Load** button.



```
[illegible]
SYSTEM CALL: [illegible]

Thread 1 started execution
SYSTEM CALL: msg_receive()
RETURNED VALUE: RCV_TOO_LARGE
PORT ID: 0
RECEIVING PORT VERSION: 0
SENDER TASK: 0
SENDING PORT VERSION: 0

SYSTEM CALL: mutex_lock()
MUTEX NUMBER: 0
MUTEX VERSION: 0

[illegible]
Thread 0 started execution
SYSTEM CALL: netname_look_up()
RETURNED VALUE: NETNAME_HOST_NOT_FOUND
TASK ID: -8204
PORT ID: 12823

[illegible]
```

Figure 20: History files

After examination of the history file information we learn that the *msg_receive* operations of the server threads have failed with the RCV_TOO_LARGE return value. The client1 *netname_look_up* operation have failed with return code NETNAME_HOST_NOT_FOUND. So we know what happened, now we need to understand why this has happened and how to fix the bug. This can be looked at in *replay* phase of debugging.

6.2.6 Play it again ...

To replay the program we have to select **Run/Replay** option from the menu bar and start all tasks one by one as we have done in *record* phase. For each task in the computation there is a scrollable window created on the main pane of the debugger window. Each window displays **GDB** prompt. To start execution of a task, the GDB command **run** has to be entered.

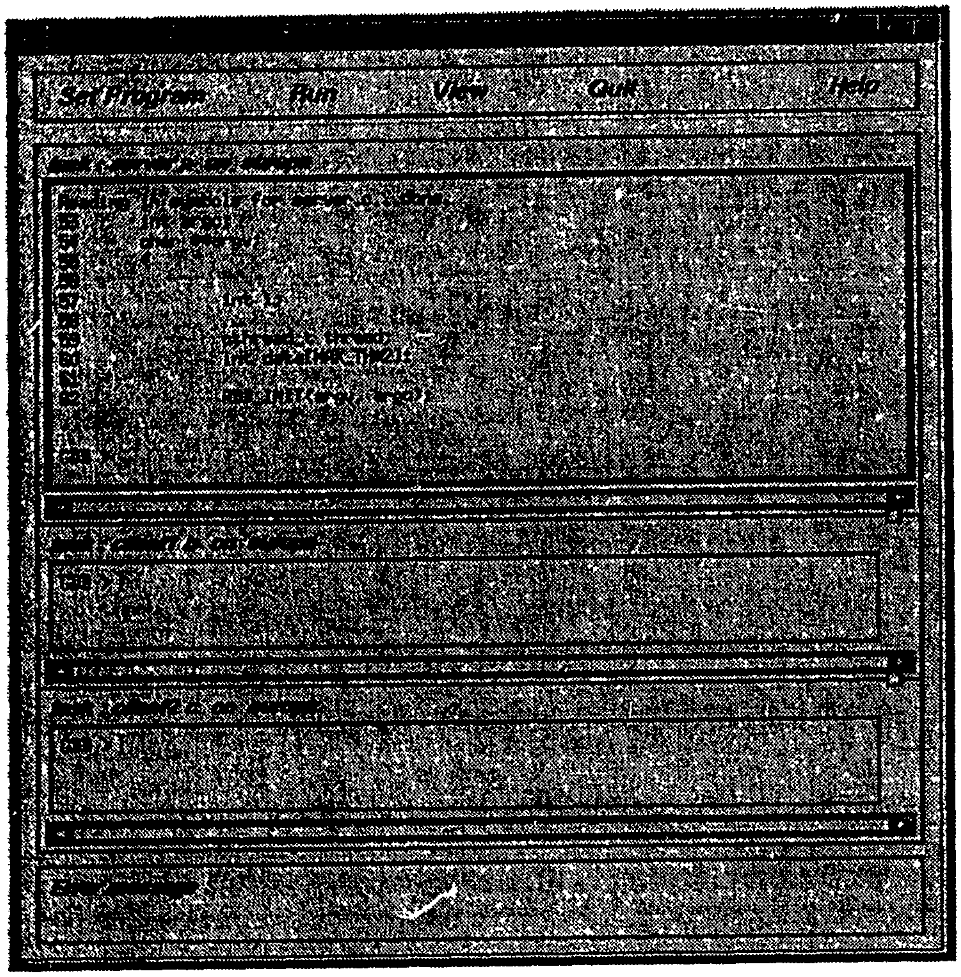


Figure 21: Replayed execution

There are several things that can be done now. We can, for example, insert a breakpoint in the client1 task right before the *netname_look_up* call. When the execution stops at this point we can advance the execution statement by statement (step) to discover that after unsuccessful *netname_look_up* call the *exit* call takes place and the task does not have a chance to send any messages to the server.

We could also examine the source code of the server task to discover that there are two types of threads: the threads 0-2 expecting small messages from client1 , and the threads 3-5 expecting larger messages from client2. All threads receive messages on the same port.

What went wrong ?

Based on this information, the details from the time-process diagram, and the information from the history files, we can now identify the source of the problem, and explain why the first execution has been successful and the second has failed.

In the first execution the small messages from client1 arrived before the large messages from client2. Since the threads 0-2 were created first they got to execute the *msg_receive* operation first and successfully received the three small messages from client1. The next three messages from client2 were received by threads 3-5.

In the second execution task client1 was delayed. The task client2 was the first to send messages to the server port. Now threads 0-2 all fail in the *msg_receive* operation. Threads 3-5 receive the three messages and the server task terminates execution. When the client1 gets to call the *netname_look_up*, the server has already terminated and this is why the NETNAME_HOST_NOT_FOUND value is returned by the netmsgserver.

How to fix this bug ?

There can be several ways of eliminating this non-deterministic behaviour. One is to introduce two ports in the server task space instead of one. Then the small messages could arrive at one port and be received by threads 0-2, and the large messages would arrive at the second port and be received by threads 3-5. This way the server would not terminate before the messages from one of the clients are received.

Chapter 7

Summary

This chapter summarises the work presented in this thesis. The summary includes description of the main contributions of this thesis work, examination of the possible benefits of this work for other designers of concurrent debugging tools, and possible future extensions to the DDB debugger.

7.1 Contributions of the thesis

The main contribution of this thesis is the design and development of the DDB debugger. We have also made a survey of 29 distributed and parallel debuggers, and classified the sources for non-deterministic behaviour in Mach programs.

7.1.1 DDB - simple but usable debugger

In this thesis we presented DDB, a distributed debugger based on deterministic replay. The main virtues of the DDB debugger are its completeness, usability, and simplicity.

The DDB is self-contained since it provides all necessary means for distributed debugging under Mach. The set of features provided by the debugger addresses the major issues of concurrent debugging by eliminating the non-deterministic behaviour in the *replay* phase of debugging and providing tools for examination of program execution and interactive debugging. The deterministic replay facility addresses the problem of non-determinism inherent in distributed programs. The time-process diagrams, and history files provide a global view of program execution, which can be helpful in understanding the interaction among multiple tasks. The source level debugging provided by GDB can be used to further investigate the program behaviour in repeated deterministic re-executions until the error is located.

The DDB debugger provides a graphical user interface that is very simple, but at the same time gives easy access to all the debugger features. The programmer is not required to create configuration files since the program specification can be entered through the user interface. The time-process diagram and the history files information can be easily accessed from the main window of the debugger. The output from each task of the program is displayed in a separate scrollable window. The source level debugger accepts commands through the graphical uses interface. Each task has a dedicated window for interaction with the GDB where the programmer can enter GDB commands. The "minimal" approach used in the design of the DDB, unlike the previous project CDB, allowed for creation of a debugger that could be designed and implemented as a master's thesis. The DDB debugger does not provide any sophisticated features, like checkpoint and rollback, or global_breakpoints. However, even in the present form, it can be successfully used for debugging of many distributed programs, and the basic set of features provided by DDB creates a good base for future developments. By extending the set of available features, the DDB can be developed into a full scale distributed debugger.

DDB has been designed with reusability in mind. During the development of the debugger we have aimed at reusing the distributed debugging research work carried in the past in the Computer Science Department at Concordia University. The reuse has been at conceptual as well as practical level. The example of practical benefit for the DDB is the use of the GDB source level debugger modified for the needs of distributed debugging during the CDB project.

The Mach operating system presents several challenges to software developers by introducing multi-threaded environment, which along with its many benefits brings the potential for errors resulting from interaction among multiple threads. An important contribution of the DDB is that it addresses the issue of debugging of multi-threaded applications, central to Mach software development. By capturing the non-deterministic behaviour at the thread level, DDB is able to repeat the patterns of interactions between multiple threads in a task.

7.1.2 Survey of concurrent debuggers

Another contribution of this thesis work is the survey of distributed and parallel debuggers presented in chapter 2. The survey is based on extensive research of the concurrent debugging field. The debuggers presented in the survey have been, in majority, developed during the last five years. The debuggers are evaluated according to their usability in concurrent debugging by placing the emphasis on the functional aspects of the debugging rather than on the methodological issues.

The information gathered by the survey is presented by descriptions of sample debuggers representing each of the identified groups of debuggers, and by presentation of functional characteristic of a larger population of debuggers (29) in tabular form. The conclusion of the survey identifies the main directions in the development of concurrent debugging tools during the recent years.

7.1.3 Classification of non-determinism in Mach programs

The design of the deterministic replay engine for the DDB has been based on examination of the potential sources of non-deterministic behaviour in Mach programs. The analysis considers two levels of communication within the distributed and multithreaded program as sources of non-deterministic behaviour: the interprocess communication level, and the thread interaction level within each task. The various points for non-determinism in Mach computational model were classified and explained with different scenarios in chapter 3. Based on this analysis we have created a mechanism for capturing the non-deterministic behaviour in multithreaded Mach programs. This mechanism is described in chapter 4.

7.2 Benefits to designers of debuggers

The results of the research conducted during this thesis project can be useful in the development of other distributed debugging tools in several ways:

Direct reuse of implemented modules

The direct reuse could have the current DDB implementation as a base for development of other debugging tools for Mach operating system. This type of reuse would benefit most of debugging systems since the deterministic replay facility provided by

DDB is the necessary component for all distributed debugging systems.

Reuse of architectural design

The architecture of the DDB is composed of entities which can be used to implement any other debugging systems. The architecture of the DDB can be viewed as a framework giving good base for any debugging tool. From all of the debugger modules only the DDB library has to be modified since it is based on Mach system calls. The History Server is easily reusable since it simply receives, stores and retrieves information without regard to the content of the information. In the same way, the Message Server and the Name Server could be reused in a message passing environment.

Trade-offs in debugger design

A designer faced with the task of developing a distributed debugger must make a choice concerning the approach to selecting the set of features to be provided by the debugger. In this respect, the approaches demonstrated by the CDB debugger and the DDB debugger are on the two opposite ends of the scale.

The CDB debugger aimed at providing full set of features for distributed debugging, including distributed breakpoint, checkpoint and rollback. On the other hand, the DDB takes "minimal" approach, and provides only essential features shifting more responsibility for debugging to the user. We conclude that our approach has more potential for success, since it allows for incremental growth.

7.3 Future work

DDB provides only minimal set of functionalities needed for distributed debugging. This minimal set can be extended to create a full scale debugger for distributed and multithreaded programs. The following extensions can be introduced:

Global Breakpoints

Global breakpoints, unlike local breakpoints, are defined in terms of events taking place in all processes in a distributed program. Definition and detection mechanism for global breakpoints has been introduced by Christy Yep in his master thesis work for the CDB debugger [52]. This mechanism could be integrated with DDB after introducing support for multiple threads, which is not available in the present version of the Breakpoint Module developed for CDB.

Checkpoint and Rollback

The checkpoint and rollback mechanism is very useful for long programs. During program execution, the state of each process is periodically saved, creating different checkpoints. When an error is detected, the program can be rolled back to a checkpoint that is considered error free, and restarted, giving the programmer the opportunity for further examination. The Checkpoint and Rollback Module developed by Alain Sarraf for the CDB debugger [41] could be integrated in the future with the DDB if modifications were made for support of multiple threads.

Visualisations of Program Execution

At present the only graphical interpretation of program execution provided by DDB is in the form of time-process diagrams. Another possible views of program execution are visualisation of the access to shared variables, messages waiting on each program port, or the computational load on each processor. These views can be dynamically built in the *replay* phase of debugging when the probe effect is not relevant.

Better integration of local debuggers with User Interface

The user of DDB interacts with the local debuggers by typing the debugger commands in the appropriate text windows. This could be changed by providing a better interface where the user could interact with the GDBs through graphical representation of the program. For example, a time-process diagram would display the execution and the user would specify local breakpoints by clicking the mouse at some point on the line representing the process.

The basic set of tools provided by DDB can be extended to transform it into a sophisticated debugging system, and reach the goals stated by the designers of the CDB debugger.

Appendix A

Distributed program example

```
/* ****************************************************************************
 * server.c - creates six threads to receive messages from clients.
 * **************************************************************************** */
#include <stdio.h>
#include <cthreads.h>
#include <mach.h>
#include <servers/netname.h>
#include <mach/message.h>
#include <mach/port.h>
#include <rdb.h>

#define MAX_TH  3
#define MAX_STRING 100

port_name_t     receiver_port;

struct user_msg_small{     /* structure for small messages */
     msg_header_t hdr;
     DEBUG_MSG
     msg_type_t type;
     char string[MAX_STRING]; } ;

struct user_msg_big {     /* structure for large messages */
     msg_header_t hdr;
     DEBUG_MSG
     msg_type_t type;
     char string[MAX_STRING*2]; } ;
```

```
int  th_count = MAX_TH *2;        /* number of threads */
mutex_t        out_lock;          /* lock for output */
mutex_t        count_lock;        /* lock for thread count */
condition_t    thh_cond;          /* condition for exiting threads */

void thread_small();
void thread_big();

/* ------------------------------------------------------------------ */

main(argc, argv)
int argc;
char **argv;
{
int i;
        cthread_t thread;
        int data[MAX_TH*2];

        RDB_INIT(argv, argc);
        out_lock = mutex_alloc();
        count_lock = mutex_alloc();
        thh_cond = condition_alloc();
        init();

       /*
        * Fork off all the child threads.
        */
        for (i = 0; i < MAX_TH; i++) {
                data[i] = i;
                thread = (cthread_t)cthread_fork(thread_small, &data[i]);
                cthread_detach(thread);
        }

        for (i = 3; i < MAX_TH*2; i++) {
                data[i] = i;
                thread = (cthread_t)cthread_fork(thread_big, &data[i]);
                cthread_detach(thread);
        }
        /*
         * Wait for all child threads to finish.
         */
        mutex_lock(count_lock);
        while (th_count != 0)
                condition_wait(thh_cond, count_lock);
        mutex_unlock(count_lock);

        /*
         * Free up all the stuff we allocated.
         */
        mutex_free(out_lock);
        mutex_free(count_lock);
        condition_free(thh_cond);
        cthread_exit(0);

} /* of main */



/* ------------------------------------------------------------------
 * Program initialization:
 *           - Allocate a port to receive message
 *           - Check in the allocated port
 * ------------------------------------------------------------------ */

init()
{
        kern_return_t   ret;
        netname_name_t  name;

        ret = port_allocate(task_self(), &receiver_port);
        if (ret != KERN_SUCCESS) {
             mach_error("in port allocate : ", ret);
             exit(1);
        }
        strcpy(name, "tserver_port");

        ret = netname_check_in(name_server_port, name,
                               task_self(), receiver_port);
        if (ret != NETNAME_SUCCESS) {
                mach_error("netname check in:", ret);
                 exit(1);}

} /* end of init */


 /* ------------------------------------------------------------------ */


void thread_small(data_item)
     int * data_item;
{

        struct user_msg_small msg ;
        msg_return_t       ret;
        int i;

       /*
        * Fill in the message header.
        */
        msg.hdr.msg_size = sizeof(struct user_msg_small);
        msg.hdr.msg_local_port = receiver_port;

        ret = msg_receive(&msg.hdr, MSG_OPTION_NONE,0);

        mutex_lock(out_lock);
        printf("\nTHREAD %d : %s\n", *data_item, msg.string);
        printf("THREAD %d  : %s\n", *data_item, "Done and gone ....");
        mutex_unlock(out_lock);

        mutex_lock(count_lock);
        th_count--;
        mutex_unlock(count_lock);
        condition_signal(thh_cond);

cthread_exit(0);

} /* of thread_small */
```

```
/* ----------------------------------------------------------------- */


void thread_big(data_item)
     int * data_item;
{

        struct user_msg_big msg;
        msg_return_t       ret;
        int i;

       /*
        * Fill in the message header.
        */
        msg.hdr.msg_size = sizeof(struct user_msg_big);
        msg.hdr.msg_local_port = receiver_port;

        ret = msg_receive(&msg.hdr, MSG_OPTION_NONE,0);

        mutex_lock(out_lock);
        printf("\nTHREAD %d : %s\n", *data_item, msg.string);
        printf("THREAD %d : %s\n", *data_item, "Done and gone ....");
        mutex_unlock(out_lock);

        mutex_lock(count_lock);
        th_count--;
        mutex_unlock(count_lock);
        condition_signal(thh_cond);

        cthread_exit(0);

} /* of thread_big */
```

```
/* ************************************************************************
 * client1.c : sends a messages to the server, does not expect reply
 * ************************************************************************ */

#include <stdio.h>
#include <cthreads.h>
#include <mach.h>
#include <servers/netname.h>
#include <mach/message.h>
#include <mach/port.h>
#include <rdb.h>

#define MAX_STRING 100

port_name_t     receiver_port;

struct user_msg{
     msg_header_t hdr;
     DEBUG_MSG
     msg_type_t type;
     char string[MAX_STRING]; } ;

/* -------------------------------------------------------------------- */

void main(argc, argv)
int argc;
char **argv;
{

        RDB_INIT(argv, argc);
        init();
        process();

        exit(0);

}

/* --------------------------------------------------------------------
 * Program initialization:
 *           - Look up the receiver port
 * ------------------------------------------------------------------ */
init()
{
        kern_return_t   ret;
        netname_name_t  name;

        strcpy(name, "tserver_port");

        ret = netname_look_up(name_server_port, "*",
                              name, &receiver_port);
        if (ret != NETNAME_SUCCESS) {
                mach_error("netname look up:", ret);
                 exit(1);
        }
} /* end of init */

/* ----------------- send the messages -------------------- */

process()
{
        struct user_msg cmsg ;
        msg_return_t       ret;
        int i;

       /*
        * Fill in the message header.
        */
        cmsg.hdr.msg_simple = TRUE;
        cmsg.hdr.msg_size = sizeof(struct user_msg);
        cmsg.hdr.msg_type = MSG_TYPE_NORMAL;
        cmsg.hdr.msg_local_port = NULL;
        cmsg.hdr.msg_remote_port = receiver_port;
        cmsg.hdr.msg_id = 123;

       /*
        * Fill in the type structure for a string of characters.
        */
        cmsg.type.msg_type_name = MSG_TYPE_CHAR;
        cmsg.type.msg_type_size = sizeof(char) * 8;
        cmsg.type.msg_type_number = MAX_STRING;
        cmsg.type.msg_type_inline = TRUE;
        cmsg.type.msg_type_longform = FALSE;
        cmsg.type.msg_type_deallocate = FALSE;

       /*
        * Fill in the data structure.
        */
        for (i = 0 ; i < 3 ; i++)
        {
                sprintf(cmsg.string, "Client one : message number %d", i+1);

               /*
                * Send the msg to the receiver.
                */
                ret = msg_send(&cmsg.hdr, MSG_OPTION_NONE,0);
                printf("\nMessage to the server has been sent\n");
                printf("Message: %s\n", cmsg.string);
}
} /* end of process */
```

```
/* ***********************************************************************
 * client2.c : sends a messages to the server does not expect reply
 * *********************************************************************** */

#include <stdio.h>
#include <cthreads.h>
#include <mach.h>
#include <servers/netname.h>
#include <mach/message.h>
#include <mach/port.h>
#include <rdb.h>

#define MAX_STRING 200

port_name_t     receiver_port;

struct user_msg{
     msg_header_t hdr;
     DEBUG_MSG
     msg_type_t type;
     char string[MAX_STRING]; } ;

/* ------------------------------------------------------------------- */

void main(argc, argv)
int argc;
char **argv;
{

        RDB_INIT(argv, argc);
        init();
        process();

        exit(0);

}

/* -------------------------------------------------------------------
 * Program initialization:
 *          - Look up the receiver port
 * -------------------------------------------------------------- */
init()
{
        kern_return_t   ret;
        netname_name_t  name;

        strcpy(name, "tserver_port");

        ret = netname_look_up(name_server_port, "*",
                              name, &receiver_port);
        if (ret != NETNAME_SUCCESS) {
               mach_error("netname look up:", ret);
                exit(1);
        }
} /* end of init */

/* ---------------- send the messages -------------------  */
process()
{
        struct user_msg cmsg ;
        msg_return_t       ret;
        int i;

       /*
        * Fill in the message header.
        */
        cmsg.hdr.msg_simple = TRUE;
        cmsg.hdr.msg_size = sizeof(struct user_msg);
        cmsg.hdr.msg_type = MSG_TYPE_NORMAL;
        cmsg.hdr.msg_local_port = NULL;
        cmsg.hdr.msg_remote_port = receiver_port;
        cmsg.hdr.msg_id = 123;

       /*
        * Fill in the type structure for a string of characters.
        */
        cmsg.type.msg_type_name = MSG_TYPE_CHAR;
        cmsg.type.msg_type_size = sizeof(char) * 8;
        cmsg.type.msg_type_number = MAX_STRING;
        cmsg.type.msg_type_inline = TRUE;
```

```
      cmsg.type.msg_type_longform = FALSE;
      cmsg.type.msg_type_deallocate = FALSE;

     /*
      * Fill in the data structure.
      */
      for (i = 0 ; i < 3 ; i++)
       {
            sprintf(cmsg.string, "Client two : message number %d", i+1);

           /*
            * Send the msg to the receiver.
            */
            ret = msg_send(&cmsg.hdr, MSG_OPTION_NONE,0);

           printf("\nMessage to the server has been sent\n");
            printf("Message: %s\n", cmsg.string);
       }
} /* end of process */
```

Bibliography

[1] Peter Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):11-23, January 1989.

[2] Peter C. Bates, and Jack C. Wileden, "High level debugging of distributed systems: The behavioral abstraction approach", *Journal of Systems and Software*, 3(4):255-264, 1983.

[3] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Keith Moore, Vaidy Sunderam, " HeNCE: Graphical Development Tools for Network-Based Concurrent Computing", *Proceedings of Scalable High Performance Computing Conference 92*, pp. 129-136, Williamsburg, Virginia, April 1992.

[4] J. Boykin, D. Kirschen, A. Langerman, S. LoVerso, "Programming under Mach", *published by Addison-Wesley*, 1993.

[5] Don Breazeal, Personal communication, *Intel*, December 1994.

[6] Don Breazeal, Karla Callaghan, Wayne D. Smith, "IPD: A Debugger for Parallel Heterogeneous System", *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 216-218, Santa Cruz, California, May 1991.

[7] Jeffrey S. Brown, "The Los Alamos Debugger ldb", *Proceedings of Supercomputer Debugging Workshop '91*, Albuquerque, New Mexico, November 1991.

[8] Deborah Caswell, David Black, "Implementing a Mach Debugger for Multithreaded Applications", *Conference Proceedings of Winter 1990 USENIX Technical Conference and Exhibition*, Washington, DC, January 1990.

[9] Chu-Chung Lin, Richard J. LeBlanc, "Event-based Debugging of Object/Action Programs", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):23-34, January 1989.

[10] Janice Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin,Lawrence Snyder, David Stemple, "The Ariadne Debugger: Scalable Application of Event-Based Abstraction", *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 85-95, San Diego, California, May 1993.

[11] I.J.P. Elshoff, "A Distributed Debugger for Amoeba", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):1-10, January 1989.

[12] Perry Emrath, Bret Marsolf, "MDB - A Parallel Debugger for Cedar", *Proceedings of Supercomputer Debugging Workshop '91*, pp. 1-3,Albuquerque, New Mexico, November 1991.

[13] Perry Emrath, Bret Marsolf, "MDB - Xylem Parallel User's Guide", *Technical Report CSRD Report 1180*, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, December 1991.

[14] Stuart I. Feldman, Channing B. Brown, "IGOR: A System for Program Debugging via Reversible Execution", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):112-123, January 1989.

[15] Alessandro Forin, "Debugging of Heterogeneous Parallel Systems", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):130-140, January 1989.

[16] Robert J. Fowler, Thomas J. LeBlanc, John M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):163-173, January 1989.

[17] Joan M. Francioni, Larry Albright, Jay Alan Jackson, "Debugging Parallel Programs Using Sound", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp.60-67, Santa Cruz, California, May 1991.

[18] Ilya Gertner, Greg Schaffer, T.T Ramgopal, "Parasight - Non-Intrusive Real-Time System Monitor", *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 231-236, Santa Cruz, California, May 1991.

[19] Robert H. Halstead, Jr., David A. Kranz, Partick G. Sobalvarro, "MULTVISION: A Tool for Visualizing Parallel Program Executions", *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 237-239, Santa Cruz, California, May 1991.

[20] Paul K. Harter, Jr., Dennis M. Heimbigner, Roger King, "IDD: An Interactive Distributed Debugger", *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pp. 498-506, Denver, Colorado, May 1985.

[21] Stefan U. Heanssgen, Personal communication, *University of Karlsruhe, Germany*, April 1994.

[22] Michael T. Heath, Jennifer A. Etheridge, "Visualizing the Performance of Parallel Programs", *IEEE Software*, pp.29-39, September 1991.

[23] Hermann Ilmberger, Sabine Thurmell, Claus-Peter Wiedemann, "VISIT: A Visualization and Control Environment for Parallel Program Debugging", *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 199-201, San Diego, California, May 1993.

[24] Hermann Ilmberger, Personal communication, *Siemens, Germany*, May 1994.

[25] Hermann Ilmberger, Sabine Thurmell, "A Toolkit for Debugging Parallel Lisp Programs", *Proceedings of PARLE'91*, pp. 406-423, June 1991.

[26] Gail Kaiser, Personal communication, *Columbia University*, December 1994.

[27] D.N. Kimelman, T.A. Ngo, "The RP3 program visualization environment", *IBM Journal of Research and Development*, 35(5/6):635-651, September 1991.

[28] V. Krawczuk, "Distributed Debugging Based on Deterministic Reexecution-Methodology and Design of a Working Prototype", M.Sc Thesis, Concordia University, Montreal, September 1992.

[29] Thomas J. LeBlanc, Personal communication, *University of Rochester*, May 1994.

[30] Thomas J. LeBlanc, John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, Vol. C-36, No.4, pp. 471-481, April 1987.

[31] Thomas J. LeBlanc, Barton P. Miller, "Workshop Summary", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):iv-xxi, January 1989.

[32] Charlie McDowell, "My list of a dream debugger features", *Proceedings of Supercomputer Debugging Workshop '91*, pp. 1-6, Albuquerque, New Mexico, November 1991.

[33] Charles E. McDowell, David P. Helmbold, "Debugging Concurrent Programs", *ACM Computing Surveys*, 21(4):593-622, December 1989.

[34] John May, Personal communication, *University of California*, San Diego.

[35] John May, Francine Berman, "Panorama: A Portable, Extensible Parallel Debugger", *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 96-106, San Diego, California, May 1993.

[36] Douglas Z. Pan, Mark A. Linton, " Supporting Reverse Execution of Parallel Programs", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):124-129, January 1989.

[37] Barton P. Miller, Charles McDowell, "Summary of ACM/ONR Workshop on Parallel and Distributed Debugging", *Proceedings of the ACN/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(2):18-31, December 1991.

on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices, 26(2):18-31, December 1991.

[38] M.Krish Ponamgi, Wnewey Hseush, Gail E. Kaiser, "Debugging Multithreaded Programs with MPD", *IEEE Software*, pp. 37-43, May 1991.

[39] Krish Ponamgi, "MpD: A Multiprocessor C Debugger", *Technical Report CUCS-021-91*, Columbia University, September 1991.

[40] Frederic Ruget, "Time-Travel for CHORUS, *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 226-228, San Diego, California, May 1993.

[41] Alain Sarraf, "Checkpoint and Rollback Recovery in a Non-FIFO Multi-Channel Distributed Environment", M.Comp.Sc. Thesis, Concordia University, Montreal, June 1993.

[42] A. Silberschatz, J. Peterson, P. Galvin, "Operating System Concepts, 3rd ed.", *published by Addison-Wesley*, Third Edition, pp 597-626, 1992.

[43] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, Rich Title, "A Scalable Debugger for Massively Parallel Message-Passing Programs", *IEEE Parallel and Distributed Technology*, pp. 50-56. Summer 1994.

[44] David Socha, Mary L. Bailey, David Notkin, "Voyer: Graphical Views of Parallel Programs", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):206-215, January 1989.

[45] Pierre E. Sorel, Mariano G. Fernandez, Sumit Ghosh, "A Dynamic Debugger for Asynchronous Distributed Algorithms", *IEEE Software*, pp. 69-76, January 1994.

[46] Larry V. Streepy, Jr., "CXdb: The Road to Remote Debugging", *Proceedings of Supercomputer Debugging Workshop '92*, pp. 305-334, Dallas, Texas, October 1992.

[47] Andrew P. Tolmach, Andrew W. Appel, "Debuggable Concurrency Extensions for Standard ML", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 115-127, Santa Cruz, California, May 1991.

[48] Larry D. Wittie, Personal communication, May 1994.

[49] Larry D. Wittie, "Debugging Distributed C Programs by Real Time Replay ", *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributing Debugging, published in ACM SIGPLAN NOTICES*, 24(1):57-67, January 1989.

[50] Steven A. Zimmerman, Personal communication, *Kendall Square Research, Waltham, Massachusetts*, June 1994.

[51] Steven A. Zimmerman, "UDB: A Parallel Debugger for the KSR1", *Proceedings of Supercomputer Debugging Workshop '92*, pp.93-102, Dallas, Texas, October 1992.

[52] C.Yep, "A Debugging Support Based on Breakpoints for Distributed Programs Running Under Mach", M.Comp.Sc. Thesis, Concordia University, Montreal, November 1992.

[53] Benjamin Young, "bdb: A library approach to writing a new debugger", *Proceedings of Supercomputer Debugging Workshop '91*, pp. 1-6, Albuquerque, New Mexico, November 1991.

[54] Benjamin Young, Personal communication, *Cray Computer Corporation.*