# Distributed Debugging Based on Deterministic Reexecution – Methodology and Design of a Working Prototype

Victor Krawczuk

A Thesis
in
The Department
of
Computer Science

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

September 1992

ISBN 0-315-81030-0

## Abstract

Distributed Debugging Based on Deterministic Reexecution –
Methodology and Design of a Working Prototype

Victor Krawczuk

Effective methods and associated tools for debugging distributed programs have evaded programmers since distributed computing's inception. The inherent problems that parallel and distributed programs introduce over sequential programs include plural loci of control (both sharing an address space (threads) and not sharing an address space (distributed systems)) and additional sources of non-determinism such as the message enqueuing ordering at given ports, the order in which threads dequeue messages from given ports, and the order of entry of threads into mutual exclusion constructs.

A methodology is presented to debug distributed programs on the asynchronous message-passing process-model (one thread per address space) based on deterministic re-execution (replay). The various ways in which replay can facilitate traditional debugging services such as breakpointing, stepping, monitoring, etc..., in a distributed setting is discussed.

The software design of a working prototype encompassing the above-mentioned features is presented. The design is founded mostly on a *language-based approach* in which the program is augmented for the collection of non-deterministic decisions and for its replay. The design also is founded partially on a *micro-kernel based approach* in which message interception is made transparent to the user of the debugger using low-level micro-kernel primitives.

Furthermore, a mechanism of replay in the context of a micro-kernel environment and a typical threads run-time environment running within the distributed system are proposed. This mechanism encompasses the process-model paradigm mentioned above, as well as support for threads, synchronous messages, migration of port-capabilities and other various basic services of a modern micro-kernel that have non-deterministic possibilities. The replay mechanism was partially implemented.

Dedicated to my parents, Michael and Halina.

## Acknowledgments

I would like to thank my supervisor, Dr. H. F. Li, for his guidance. I would also like to thank him for his financial support. Thanks also goes to Dr. T. Radhakrishnan.

I would also like to thank my friends and colleagues in the Department for all the advice (especially programming assistance) they gave me during the implementation phase of this project, especially Rick Clark. I also would like to thank Paul Gill for setting up and maintaining the Mach system.

I also would like to extend special thanks to some of the developers of Mach at Carnegie Mellon University (specifically Rich Draves, Dan Julin and Mary Thompson) for answering the many questions I had asked them time and time again.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

*Maurice Wilkes discovers debugging, 1949*

One area of ongoing research in software engineering concerns developing methodologies to translate a software requirements specification (SRS) into source code that satisfies the requirements specification and is bug-free. Unfortunately, today's state of software engineering still does not produce errorless programs either at the source-code level or in design documents that evolve out of the SRS. The release of flawless software of significant size in practice is considered a hopelessly optimistic task now, and into the foreseeable future. Proving a program is correct via mathematical rigor is only considered practical for trivial programs.

Debugging can be described as the process of determining why a program has violated its SRS either by performing some unauthorized action such as:

- a legal transition has resulted in an illegal state
- an illegal transition has resulted in an illegal state

- an illegal transition has resulted in a legal state

Finding the cause of errors (the bug) can be a labour intensive activity. Therefore, debugging methodology and tools to support the developed methodologies must be devised in order to improve programming efficiency.

Inevitably, when one discusses debugging, one finds it difficult to separate it from program testing. While the two concepts are very much related, testing is more a methodology to determine whether illegal states/transitions **can** occur or legal states/transitions **cannot** occur, using the SRS's definitions of what constitutes a legal state/transition. Debugging is more concerned with finding the root cause of an illegal state or an illegal transition (at the level of the SRS), which often finds its roots in faulty design documents, algorithms, etc..., and ultimately, faulty source code. Furthermore, the underlying cause of a program violating its SRS often occurs far before the explicit manifestation of an error relative to the SRS, and in the case of a distributed program, the source of the error (bug) may have originated on a different process/task[1] from which the violation of the SRS was detected!

## 1.1 Distributed Debugging is Different from Sequential Debugging

### 1.1.1 Re-execution and Non-Determinism

Distributed debugging techniques are more complex than conventional debugging techniques. Conventional debugging techniques are designed for a single process (with a single locus of control, single-threaded) running on one processor. Conventional debugging techniques also assume that given the same sequential stream of external inputs to the program and the absence of random number generators, the program can be rerun countless times and it will always follow the same sequence of states and the sequence of transitions will not change. Repeatability is essential in debugging, since one execution is necessary to detect an error, and often a program must be re-run several times as the cause of the error (the bug) is tracked down.

[1]a process is considered a single-threaded task in this thesis

Debugging a distributed system is complicated by the potential for non-determinism in one's program. Thus the corner-stone of conventional software debugging, which is the guaranteed reproduction of program errors, often cannot be applied to distributed programs. For example, since multi-threaded tasks communicate via messages sent to ports, if two such tasks have send rights to a given port $p$, the order of reception from the two tasks at port $p$ cannot be guaranteed for each re-execution. If a particular thread in a multi-threaded task is capable of receiving from a port-set $P$ and will randomly pick a port to receive from if more than one port is non-empty, then this is another source of non-determinism, which cannot be guaranteed to be reproducible during subsequent re-executions. Furthermore, the timing of message arrival to any port in a port-set also adds to non-determinism since the receiving thread of a port-set is obliged to dequeue a message in a port in a port-set if all the other ports in the port-set are empty.

Multi-threaded processes (tasks) are inherently non-deterministic, since CPU scheduling often determines which thread gets exclusive access rights to a critical region shared by all the tasks in the thread and protected by a mutual exclusion construct (such as a semaphore or monitor), assuming there are no errors in the mutual exclusion construct itself! In a message-based distributed system, multi-threaded tasks significantly add to overall non-determinism since several threads can potentially receive from a given port in a given sequence, which cannot be guaranteed to happen on subsequent re-executions. Furthermore, since various threads from various threads can acquire send rights to a common port, the enqueuing order at a port cannot be guaranteed to be reproducible on repeated executions.

### 1.1.2 Breakpoints

Conventional programs allow the user to specify an unambiguous breakpoint in the code at which point the user can examine the state of the process (*e.g.*, variables). The breakpoint is unambiguous and it is guaranteed to be reached if it indeed can be reached[2], since there is only one flow of control (thread) in a sequential program.

[2]if random number generators (RNG) and external data cooperate, and there is no bug in the code to prevent the program from ever reaching the breakpoint, irrespective of RNG's and external

Once the program is restarted from a breakpoint, the program behaves as if it had never been artificially suspended. Furthermore, given the deterministic behavior of sequential programs, the user can use the technique of *stepping*, in which the process is automatically suspended after every $x$ number of instructions is executed. Usually, $x$ is equivalent to all the instructions inherent in one line of source code.

Breakpoints in a sequential program have an implied reference to time, typically when a user-composed assertion becomes true. A distributed system is typically a loosely coupled multiprocessor system, which implies that the various processors do not share a common clock and that there are several threads of control spread across several tasks. The fact that the tasks do not share a common clock implies that there is no way to stop all the processors at precisely the same time. This implies that the concept of simultaneous events must be redefined to one that is suitable for distributed systems. Another complication that must be considered is that if one places a breakpoint in a given single threaded task and it is reached, inevitably, this will cause other processes that receive messages from the suspended process to block, while other processes may progress to completion, the result being that the concept of a breakpoint in a distributed program must be redefined (discussed later in section 3.1).

### 1.1.3 Transparency of the Debugger

Ideally, the distributed program should behave in exactly the same manner with the debugger as it would if the debugger were not present. In reality, this does not seem to be possible with distributed systems. Message sending and receiving can be delayed by the debugger itself by it "stealing" CPU cycles from the application program being monitored/debugged, affecting CPU scheduling, and possibly delaying the application program access to system resources such as the bus, I/O peripherals (disks, Ethernet, etc...). This disruptive effect of the debugger, the **probe** effect, can mask errors that would otherwise be observed during an execution of the program *without* the debugger present.

data

### 1.1.4 Tracing

*Tracing* is another sequential debugging technique often used. The user typically specifies which events he wants written to standard output when they occur (*e.g.*, variable changed, variable referenced, label passed, etc...). Typically, only a few variables and/or labels are traced at any given time, to avoid generating a flood of data that the user must sift through.

In a distributed system, traces must be collected from various sites and then "combined" at a central site either physically or logically. In a sequential program, outputting trace statements will not generate a probe effect, whereas in a distributed setting it would, due to the extra instructions necessary to generate traces, and due to the burdening of the communication system linking the nodes if monitoring data must be sent/coordinated between various nodes.

### 1.1.5 Programming Environment Supported

In this thesis, three aspects of distributed debugging are discussed: methodology, debugging tools to support the methodology, and a mechanism to monitor and replay non-deterministic distributed programs.

Starting first with the mechanism for monitoring/replay, the programming environment supported consists of:

- a microkernel (specifically Mach [2]). All the non-deterministic events that such a kernel presents (except for the external memory manager [2]) are recorded and the non-deterministic events are regenerated during replay. Facilities supported include asynchronous and synchronous message passing, send and receive timeouts, non-deterministic system calls, a multi-threaded environment (see next item for qualifier), name-servers, and the ability of passing port capabilities in messages.

- the C-Threads package [6] that uses the low level thread primitives of a microkernel and provides thread synchronization features.

- the C programming language.

The **cyclic debugging** methodology presented in chapter 2 is designed to take advantage of the monitoring/replay facility described above.

Some of the described debugging tools designed to support the **cyclic debugging** methodology are for use only in a process-based (one thread per task) asynchronous message-passing environment. The breakpoint and stepping tools (see sections 3.1 and 3.2) fal into this category. The database, and the manual checkpoint, rollback and recovery facilities (see sections 3.3 and 3.4) can function in the full environment, as described above. Time constraints prevented the development of breakpoint and stepping tools that would function in the full environment described above.

### 1.1.6 Previous Work

#### Mechanism of Monitoring/Replay

While previous papers hinted at the utility (and mechanism) of deterministic re-execution for debugging parallel and/or distributed systems, the first serious attempt at discussing the issue in any great depth, thus expounding the necessity of deterministic re-execution in debugging, was put forward by LeBlanc and Mellor-Crummey [21]. In this paper, the general idea of only recording the relative order of significant non-deterministic events as they occur, as opposed to all events and the data associated with the events, during the *recording* phase was proposed. Since they implemented a prototype parallel debugger based on a shared-memory multiprocessor, their recording and replay technique was directed at the shared memory paradigm. They only proposed a recording and replay algorithm for the shared memory paradigm because they believed that all parallel and distributed debugging is derived from the shared-memory paradigm.

The prototype described in this thesis differs from from the one developed by Leblanc and Mellor-Crummey [21]. They proposed a very general solution to deterministically replaying parallel programs in that the mechanism is based on the shared memory paradigm, arguing that this paradigm's basic principles can be extended to other paradigms, such as loosely coupled systems. While this premise (and their algorithms for replay on such a paradigm) is arguably correct, the amount of detail that rests between their algorithms and a working replay mechanism on a loosely coupled system is large. Issues like efficiency, the probe-effect, etc...in a loosely coupled system are glossed over in the algorithm proposed in [21]. Issues such as how the replay system can be effectively hidden from the user and how to name operating system resources, which are given different ID's during each reexecution, thus making the job of mapping execution histories to the correct resources difficult, is not dealt with in [21]. Leblanc and Mellor-Crummey also do not describe how timeouts, name servers, and non-deterministic system calls should be handled during replay. They also do not consider the various race conditions that can occur in a loosely coupled system of multi-threaded servers.

Fidge [11] proposed a monitoring/replay mechanism for purely message-based programs that are based on CSP [16] (*i.e.*, rendez-vous synchronization). He also proposed an algorithm for checkpointing a computation during the *replay* phase in case the recorded execution history became unmanageably long and had to be truncated. The approach taken in this thesis differs in that the proposed monitoring/replay mechanism supports a platform encompassing more than just the CSP paradigm. Furthermore, in [11], the checkpointing algorithm is based on an algorithm developed by Chandy and Lamport [5].

In [3], a mechanism for the replay of a program based on shared variables was put forward. The mechanism does not deal at all with message-passing programs.

In [31], a monitoring/replay mechanism was proposed for a real-time distributed system. They claim to avoid any probe effect during the *recording* phase by using hardware probes to collect the execution history. The work in this thesis uses software probes instead during the *recording* phase, and is not intended for a "hard" real-time environment. Furthermore in [31], since they are targeting a "hard" real-time system, they monitor and replay I/O and hardware clock interrupts, aspects that are not critical to replaying most non-real-time distributed systems.

# Chapter 2

# Debugging Methodology

The apparent necessity of having a replay facility in a distributed debugger was elegantly put forward by [21]. In addition to the fact that replay guarantees reproduction of an error/bug on every re-execution (if external data is the same on each execution), **cyclic debugging** methodology, which is the staple of sequential (conventional) debugging, can be, in principle, employed in the context of a distributed program. This is typically achieved by focusing one's attention on a smaller, more specific "region" of the program with greater attention to detail on each iteration of the execution until the bug that caused the error is found. Humans generally think sequentially rather than in parallel. Therefore, incorporating methodologies from the sequential debugging paradigm (where possible) is important in order to streamline the debugging process and provide a more intuitive environment for the user.

## 2.1 Recording and Replay Phases

The methodology assumed for debugging non-deterministic distributed programs consists of two phases—the *recording* phase and the *replay* phase.

### 2.1.1 Recording Phase

The *recording* phase can be considered the **first phase** of the debugging methodology. During this phase, all non-deterministic choices are monitored and recorded as they occur for use during the *replay* phase. The *recording* phase is not concerned with the reasons why a given non-deterministic choice was made–it is simply concerned with recording the choice.

It is assumed that no debugging activity will take place in the *recording* phase other than recording the minimum amount of data related to non-deterministic choices (done automatically by the debugger) in order to deterministically re-execute a program during the *replay* phase. The *recording* phase will thus be typically used when a program is being tested to determine if the program satisfies its specified requirements. If an error is found, the user would revert to the *replay* phase in order to use the cyclic debugging methodology to reexecute the program, using various debugging techniques, to find the cause of the error.

Since non-deterministic choices are being logged onto secondary storage with limited capacity, a lengthy execution may produce more monitored data than can fit onto the secondary storage medium. In [11], when a pre-determined number of choices is recorded for any process, a checkpoint is initiated at that process by sending marker messages to all its neighbor process(es) (similar to [5]). When space runs out on secondary storage, a portion of the earliest recorded history still present on the secondary storage device must be deleted so that monitoring can continue. Since reexecuting from the initial state during the *replay* phase is not feasible if the earliest recorded choices are not available to guide the re-execution with determinism, the user must rely on a checkpoint that was taken automatically during the recording phase as a starting point.

Even if the minimum amount of data is logged during the *recording* phase, it still may introduce a small probe effect that may mask errors/bugs that would have surfaced if the debugger was not being used. A software solution to the probe effect currently does not exist. There are few techniques that can be used to compensate and/or minimize for any probe effect that may be present in the *recording* phase:

**Exhaustive Testing** If the program is exhaustively tested, then the probe effect is irrelevant, since all lurking bugs should have been unmasked. The *recording* phase is used to pinpoint certain execution runs that exhibited faulty behavior under specific controlled conditions, so that cyclic debugging techniques can be used to find the cause of the error. Exhaustive testing is often impractical in practice due to the large number of states and permutations of their execution sequences that a typical distributed program can find itself in. Furthermore, developing test suites (and techniques) that will test (and coerce) **all** possible states of a program is a research field by itself.

**Random Artificial Probe Effect** Adding artificial delays to message delivery, putting arbitrary threads and/or tasks to sleep for arbitrary amounts of time, etc..., is a technique that can be used to unmask bugs that may be masked by the debugger's presence during testing or the *recording* phase. The artificial probes can effectively cancel out the probe effect due the debugger.

**Airline "Black-Box" Technique** This technique for compensating for the probe effect involves permanently incorporating the *recording* phase of the debugging environment and logging every execution's non-deterministic activity, much like the flight-recorder in an airplane, which always monitors its operation on every flight–if something goes wrong, the data is used to reconstruct the events that lead to the malfunction. If the debugger masks any bugs in the execution, it is of no concern since the program doesn't execute the masked faulty portion of the program anyway. When the program eventually does execute a faulty portion of the program (due to sheer luck) that the debugger generally masks, that execution was monitored. Thus, the faulty run can be re-executed in the *replay* phase, applying cyclic debugging methodology to debug the program. Typically, some type of automatic error-detecting device (like an on-line specification checker described in [7]) could be employed in order to flag an error condition if it ever occurs.

### 2.1.2 Replay Phase

During this phase, the debugger ensures that all non-deterministic choices the program makes (or is coerced into making by its environment) matches the choices it made during the *recording* phase. The program is thus rendered deterministic and its **behavior** is no longer affected by the debugger's **probe** effect, creating the proper environment for **cyclic** debugging methodology. Since the probe effect is not an issue, one can attach and use as many debugging tools to a re-execution as necessary without affecting the program in any manner. An assortment of debugging tools designed and/or derived from the conventional debugging world is described in chapter 3. The debugging methodology in this phase becomes heuristic. The overall methodology of this phase is to re-execute the program as many times as it takes, each time gathering more detailed information about fewer processes that the user suspects may contain the bug that caused the error (top-down methodology) until the bug is found. The order and selection of using the various debugging tools depends on the type of program being debugged and its related algorithms and the user being aware of the tools at his disposal.

### 2.1.3 Debugging Based on Replay

While there are some drawbacks to basing a distributed debugger on replay, such as the substantial complexity of developing a **replay** subsystem for a distributed system, the non-trivial probe effect present in the *recording* phase, and large amount of space such a debugger requires on secondary storage (for the execution logs and the checkpoints), there appears to be no other more efficient methodology for debugging a non-deterministic program. The cost of secondary storage is falling steadily, and software platforms are becoming more and more standardized, allowing debugging tools to become more portable. Furthermore, fewer upgrades of the debugger become necessary as software standards become less of a moving target, thus the core of the complex debugger itself need not be changed as often.

# Chapter 3

# Cyclic Debugging Tools

As mentioned in chapter 2, all debugging activity is done in the context of the *replay* phase and the debugging activity uses facilities that take advantage of and/or rely on deterministic re-execution to support **cyclic debugging**. Most of the facilities have been taken from the realm of traditional debugging and have been adapted for use in distributed programs.

## 3.1 Breakpoint Related Tools

The main problems of setting breakpoints in non-deterministic programs is that the breakpoint itself can introduce a probe effect (resources must be used to detect when to precipitate a distributed breakpoint, as well as coordinating a distributed breakpoint when the precipitation location has been detected) and a given distributed breakpoint may not be reached on repeated executions due to random delays. These two problems do not exist in sequential debugging. Thus, breakpoints are only set during the *replay* phase where the program is executed deterministically.

The different key paradigms used in distributed programs would place different requirements on debugging. To meet these requirements, two classes of breakpoints are proposed: Optimistic Consistent Breakpoint (OCB) and Pessimistic Causal Breakpoints (PCB) [19]. Both OCB and PCB can be specified as distributed breakpoints over a **subset** of the communicating processes. Their definitions guarantee that the **global state** reached by the distributed program is unique in that it can be reproduced at will on subsequent re-executions.

When the user specifies the OCB and/or PCB, the set of all processes in the distributed program is partitioned into two sets $P$ and $Q$ such that for each process in $P$ the user has specified a breakpoint. In the debugger, facilities are provided for the user to specify such breakpoints either in the **code space** (program source code) or in a **synchronization specification space** (messages exchanged at the process boundaries).

The PCB and OCB together identify a state region in which certain restricted conditions hold (such that some selected processes have reached certain local states and have ceased to progress further). This region of global state is useful in checking safety properties restricted to a subset of the processes that are allowed to progress in the absence of the rest Figure 3.1 describes how the PCB and OCB breakpoints differ given the same breakpoint set in a given process. It should be noted that the breakpoint schemes just described (PCB and OCB) are designed for distributed programs made up of *single-threaded* application tasks (tasks that consist of only one application thread, *i.e.*, a traditional process).

### 3.1.1 Optimistic Consistent Breakpoint

In the case of OCB, each process is stopped at either its breakpoint or at a point where it is blocked waiting to receive a message from another process that is also blocked or has reached its breakpoint. The fact that the processes in set $Q$ are allowed to advance forward **maximally** until they are blocked allows the user to investigate possible violations of mutual exclusion (*e.g.*, data contamination, liveness violations, etc...).

### 3.1.2 Pessimistic Causal Breakpoint

In the case of PCB, each process breaks at the earliest state that reflects all events that "happened before" the breakpoint [13]. This is useful in that the states of processes in $Q$ are not allowed to pass beyond the causal state and possibly mask the cause of an error.

The algorithm used in [13] forces some processes in $Q$ to rollback if they have

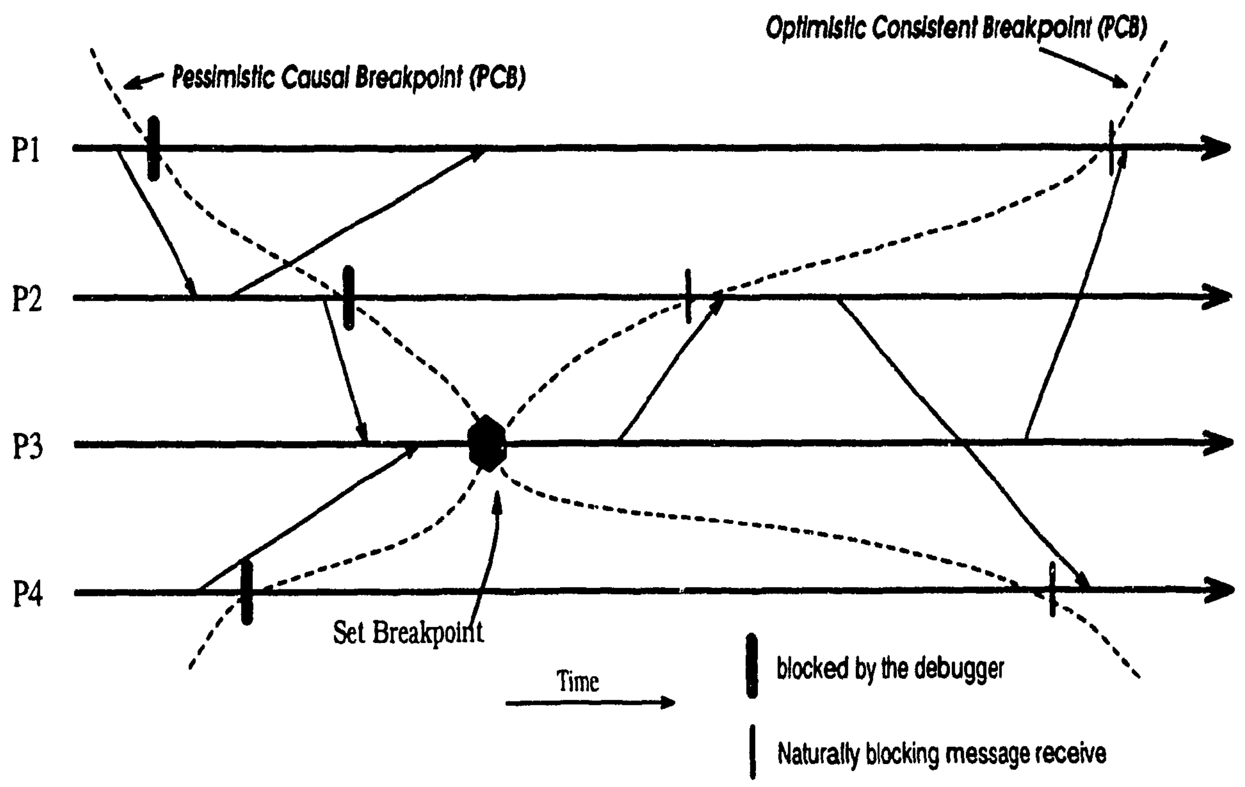
Optimistic Consistent Breakpoint (PCB)
Pessimistic Causal Breakpoint (PCB)
P1
P2
P3
P4
Set Breakpoint
Time
blocked by the debugger
Naturally blocking message receive

Figure 3.1: An example of OCB and PCB breakpoints

surpassed the **causality** requirement when a pre-set breakpoint is reached in $P$. Time-stamps must be maintained in order to rollback the processes in set $Q$ so that they reflect all events that "happened before" the breakpoint reached in a process from set $P$. An improvement to the latter method is proposed for achieving causal breakpoints in a more efficient manner in terms of space and time. A causal breakpoint is implemented without the need for time-stamps, checkpoints, rollback, or any special re-processing as required in [13]. This is achieved by pessimistic re-execution, which allows processes in set $P$ to proceed unhindered toward their pre-set breakpoints while only allowing processes in set $Q$ to *advance enough to unblock* processes in set $P$. For more details, refer to section 4.2.

## 3.2 Stepping

Stepping is another tool which is commonly used in conventional debugging for manually following the flow of a program, usually a small segment of it, and possibly pausing to investigate the state the program has found itself in. An added benefit of the OCB and PCB mechanism is that it can easily be transferred to a stepping facility.

In the case of OCB, one can single-step a single process and the other processes will automatically step **maximally** forward to the maximal prefix. This is naturally accomplished due to the enforcement of the happened-before relations due to the blocking receive paradigm. This type of stepping is useful for when the user doesn't care about examining the causal states of the other non-stepped processes and thus allows them to run beyond their causal relevance to the stepped process.

Related to PCB mechanism, a pessimistic form of stepping can also be used to actively step one process while not allowing the other processes to progress beyond the earliest causal event relative to the process that is currently being actively stepped by the user. This feature is provided so that the **causality** requirement is constantly maintained at the other processes while advancing a single process at a fine granularity (one statement at a time). This type of stepping provides the user with a "white-box" tool, which comes from the fact that the user is stepping the one process in isolation with a sequential debugger, which can step from statement to statement. The black-box comes from the fact that the other processes are only breaking at selected send events. The user can always switch to another process to do white-box analysis during a re-execution to try to account for the current state of the actively stepped process.

## 3.3 Database

The intended use of a database in the context of distributed debugging is to provide the user an alternative method for "replaying" an execution in a more abstract and selective manner than a re-execution. It is also useful in checking whether certain user-composed assertions held true or not during the re-execution. Selected events are recorded during the *replay* phase. The collection of the events must be done in the *replay* phase to avoid creating a significant probe effect in the *recording* phase. Since the user cannot predict beforehand if and where the bug will be, **all** events would have to be recorded, which would cause a massive probe effect if events were collected during the *recording* phase.

The user typically collects **selected** events during repeated re-executions, since recording all possible events, even during the deterministic *replay* phase, can be unnecessarily cumbersome, especially in terms of the large amount of data that can be collected. The data collected in the database is cumulative on repeated executions in a particular session of the *replay* phase. An off-line "filtered" replay or listing of a selected sequence of events can be demanded by composing an SQL-style query (based on predicate logic). This, in effect, provides the user with the **stepping** facility at an abstraction layer higher than at the level of the source code. Furthermore, assertions about the program can be composed and verified against the events recorded in the database using SQL-style database queries (based on temporal logic). Implicit bugs, which occur far from the location where the program explicitly violates its requirements specification, would be easier to detect if the user knows some useful assertions about the program that is being debugged. The approximate location of an assertion violation could also be provided with this type of query.

| |
|---|
| machine-id (sending from...) |
| task-id (sender) |
| thread-id (sender) |
| port-id (destination) |
| global-time |
| source code line # (of send primitive) (in case primitive was unsuccessful) |
| message tag value |
| confirmed delivery to destination port |
| message contents (actual or ptr to VM) |

Table 3.1: Attributes of message-transmission event

The EVENT FILE (database) is cumulative in that for each re-execution, one can record different selected events, in case the events recorded during the previous passes are insufficient in locating the bug.

### 3.3.1 Selecting Events to Monitor

The debugger is capable of recording "global" and "local" events in a user-named EVENT FILE that is uniquely associated with a specific run. The events are culled during *replay* mode only.

All logged "events" will include the event name, the machine-id, the task-id, the source code line number, and the global clock value, if not debugging in a multi-threaded environment. The global timestamp service is provided so that the user can determine the causal relationship between events, if any, during an off-line query of the database. Some events will record more information as described below:

#### Global Events

- message transmission. This will also record the message tag [1] (if any), and whether the message was successfully delivered to the appropriate port. If delivery was unsuccessful (*e.g.*, a timeout occurred) the "confirmed delivery to destination port" field would indicate this fact. The message contents trans-

[1]some operating systems, like Mach, allow the programmer to type messages so that the receiver can quickly determine what **type** (user defined) of message was received without needing to examine the actual message

| |
|---|
| machine-id |
| task-id |
| thread-id |
| port-id |
| port-set? |
| port-set-id |
| global-time |
| source code line # |
| message tag value |
| message contents (actual or ptr to VM) |

Table 3.2: Attributes of message-reception event

| |
|---|
| machine-id |
| task-id |
| thread-id |
| port-set-id |
| global-time |
| source code line # |

Table 3.3: Attributes of port-set-create event

mitted will be recorded, if they are actually contained in the message (in-line). If a pointer to virtual memory serves as the message, then only the pointer is recorded (see Table 3.1), since such as message can be as large as 2 Gigabytes.

- message reception. This event is defined as a thread dequeuing a message from a port into the receiving task's address space. The "port-id" field tells from which port the message was dequeued. If the dequeued port was a member of a port-set, the name of the port-set is provided, as well as which actual port was dequeued. For example, under the Mach operating system, receiving from a port-set can be a non-deterministic if two or more ports of the port-set are non-empty.

- port set create. When a port set is created, the event is recorded (see Table 3.3).

- port set destroy. When a port is destroyed, the event is recorded (See Table 3.4).

- port set add. Records the event when a port (port-id) is added to the port-set

| |
|---|
| machine-id |
| task-id |
| thread-id |
| port-set-id |
| global-time |
| source code line # |

Table 3.4: Attributes of port-set-delete event

| |
|---|
| machine-id |
| task-id |
| thread-id |
| port-set-id |
| port-id |
| global-time |
| source code line # |

Table 3.5: Attributes of port-set-add event

(port-set-id) (See Table 3.5).

- port set remove. Records the event when a port (port-id) is removed from the port-set (port-set-id) (See Table 3.6).
- create task (see Table 3.7). This event records that a specific task was just created.
- create thread. This event records that a specific task was just created.
- destroy thread. This event records that a specific task was just destroyed.

| |
|---|
| machine-id |
| task-id |
| thread-id |
| port-set-id |
| port-id |
| global-time |
| source code line # |

Table 3.6: Attributes of port-set-remove event

| |
|---|
| machine-id |
| task-id |
| thread-id |
| task-id of creator |
| thread-id of creator |
| global-time |
| source code line # |

Table 3.7: Attributes of create-task event

| |
|---|
| machine-id |
| task-id |
| task-id of destroyer |
| thread-id of destroyer |
| global-time |
| source code line # |

Table 3.8: Attributes of destroy-task event

- destroy task (see Table 3.8). This refers to the task that has just been destroyed. If a task kills itself, the task-id of destroyer will be the same as the task-id.
- port creation. This applies to the receiving port for normal communication messages only. All other ports that are created are not recorded (see Table 3.9). For example, under the Mach operating system, there is a port for all system resources, such as regions of virtual memory, task bootstrap ports, task exception ports, and task notify ports.
- port destruction. Same conditions as for port creation apply (see Table 3.10).
- task done (see Table 3.11)

| |
|---|
| machine-id |
| task-id |
| thread-id |
| port-id |
| global-time |
| source code line # |

Table 3.9: Attributes of port-creation event

| |
|---|
| machine-id |
| task-id |
| thread-id (if applicable) |
| port-id |
| global-time |
| source code line # |

Table 3.10: Attributes of port-destruction event

| |
|---|
| machine-id |
| task-id |
| thread-id |
| global-time |
| source code line # |

Table 3.11: Attributes of task-done event

| |
|---|
| machine-id |
| task-id |
| thread-id |
| global-time |
| reason (signal # ) |
| source code line # |

Table 3.12: Attributes of task-aborted event

| |
|---|
| name of user event |
| machine-id |
| task-id |
| thread-id |
| global-time |
| source code line # |

Table 3.13: Attributes of user event

- task aborted (see Table 3.12). It is noted whether the abortion was due to a normal or abnormal (*e.g.*, due to a hardware exception)"exit".
- user event. A user event is defined as a specifically named **label** (meaningful to the user who defined it) that the user inserts in some specific area of the source code. Each time the label is by-passed, the clock "ticks" and the user event is recorded into the database (see table 3.13).

**Local Events**

Each event tuple includes fields for the logical time, event identification, source code line, task-id, thread-id, machine-id, and data.

- assignment to a local variable
- assignment to a named pointer
- reaching a label

### 3.3.2 Selecting Which Events to Log during Replay

During the *replay* phase, the user can specify which events to record. The BNF for the form the commands can take can be found in appendix C. A similar BNF exists to selectively stop recording certain events during the *replay* phase.

Occasionally, the user won't be sure what task and/or threads were created and destroyed during the execution. This makes it hard to specify which task(s)/thread(s) one wants to record. Such information can be provided by replaying the execution and, for example, recording all the create-task and create-thread events, and then generating the list by using the query facility.

## 3.4 Checkpoint, Rollback and Recovery

The ability of allowing the user to initiate a checkpoint, as well as to rollback to a selected checkpoint within a debugger during the *replay* phase allows the user to reexecute a suspect area of the program repeatedly without reexecuting the entire program from the beginning each time. This facility is distinct from the checkpoints which are periodically taken during the *recording* phase in order to restart a recorded execution from a state other than the initial state, in the event that an early portion of the execution history had to be deleted in order to make room for more recent monitoring data.

The idea of reversible execution, executing "backwards" in time, is not a new one [23] [22] [12] [10]. Basically, one must "unexecute" logically from the location of the error to the associated bug. The proposed tool supporting the user during the *replay* phase allows the user to initiate checkpoints **manually** at global breakpoints. This allows the user to re-execute a selected subportion or the program iteratively, from a manually induced checkpoint to some upper bound (represented as a breakpoint). The intended goal, on each reexecution, is to reduce the suspected area in which the bug resides between a manually induced checkpoint to some upper bound (represented as a breakpoint), as well as the number of suspect processes. The user can select which manually-induced checkpoint to rollback to.

## 3.5 Graphical User Interface

A graphical user interface (GUI) provides a unified interface to all debugging facilities. It enforces the cyclic debugging methodology, as well as provides the user with a centralized location where he can issue commands and view selected states of the program. A flow-chart illustrating the sequence of steps a person debugging a distributed program must go through is illustrated in figure 3.2 and tables 3.14 and 3.15.

During the *recording* phase, the user is not debugging the program, so the GUI consists of just a simple menu system to set-up the *recording* phase. The GUI during

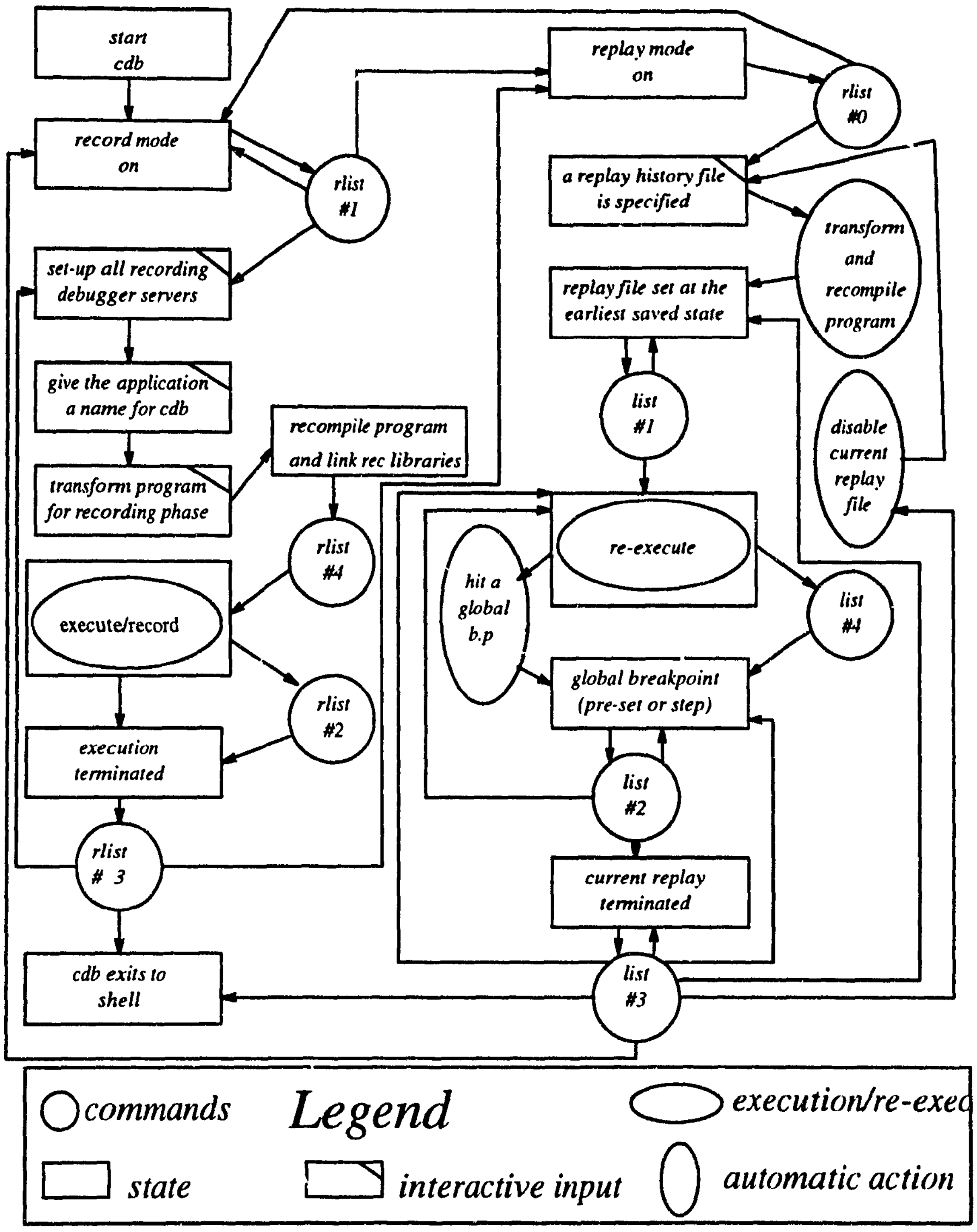
start cdb
record mode on
rlist #1
set-up all recording debugger servers
give the application a name for cdb
transform program for recording phase
recompile program and link rec libraries
rlist #4
execute/record
rlist #2
execution terminated
rlist # 3
cdb exits to shell
replay mode on
rlist #0
a replay history file is specified
transform and recompile program
replay file set at the earliest saved state
list #1
disable current replay file
re-execute
hit a global b.p
list #4
global breakpoint (pre-set or step)
list #2
current replay terminated
list #3
commands
Legend
execution/re-exec
state
interactive input
automatic action

Figure 3.2: Command/State Flowchart

| RLIST | COMMAND |
|---|---|
| rlist # 1 | change mode to replay |
| | delete execution history file |
| rlist # 2 | abort an execution being recorded |
| rlist # 3 | record another run |
| | exit debugger |
| | change mode to replay |
| rlist # 4 | start record |

Table 3.14: Groups of commands for *record mode*

the *replay* phase consists of a number of windows on a bit-mapped screen running a distributed windowing system such as the X-Window system [24], which presents the users the following:

**blocked table** This table lists all the task-threads which are blocked due to a "blocking" message receive (awaiting a message on some port the task has receive rights to). The location of the task (node-name) is also provided. This facility is helpful in allowing the user to manually detect deadlocks, and it is useful during stepping in that the table can assist the user in determining the cause of a blocked process, which can be caused by another blocked process.

**suspended table** This table is similar to the **blocked table** (see above item) except that the task-thread's listed have been explicitly *suspended* by the **task controller** (the modified sequential debugger *gdb*). The task is under control of the **task controller** and the task's state can be examined, if need be. The use of the suspended table is to enable the user to determine if a specified breakpoint has been reached.

**artificial delay indication** There may be occasions during the *replay* phase when an event that should happen (for example, a message that has definitely been sent should have been enqueued at the destination port) won't happen immediately. This can happen if the replay mechanism has noticed that the program is attempting to execute some non-deterministic event that is not currently in accord with the relevant execution history (like the receipt of a message "out

| LIST | COMMAND |
|---|---|
| List # 0 | change execution history |
| | switch to record mode |
| List # 1 | start re-execution |
| | selecting events to monitor to database |
| | delete execution history file |
| | breakpoints |
| | enable isolation run |
| List # 2 | query of monitored events in event database |
| | activate stepping mode |
| | set/unset breakpoints |
| | take manual checkpoint |
| | rollback to a manually induced checkpoint |
| | selecting events to monitor to database |
| | abort re-execution |
| | remove duplicate events in database |
| | delete event file |
| | save event file |
| | global resume (of re-execution) |
| | list manual checkpoints |
| | delete current saved checkpoints |
| | suspend recording of events |
| | resume recording of events |
| | quit database query |
| List # 3 | remove duplicate events in database |
| | delete event file |
| | save event file |
| | delete current saved checkpoints |
| | delete replay file |
| | rollback to a manually induced checkpoint |
| | query of monitored events in event database |
| | list manually induced checkpoints |
| | switch to record mode |
| | change replay files |
| | quit query |
| List # 4 | hit panic button (force immediate breakpoint) |

Table 3.15: Groups of commands for *replay mode*

of order") is delaying the event. These flags will indicate which task-thread is being artificially delayed. This facility is most useful in conjunction with the **stepping** facility.

**task controller** This is the front-end of the modified *gdb* sequential debugger, running as an inferior process of the **emacs** full-screen editor [25]. The full-screen editor must be used in order to interactively display the source-code of the program at a breakpoint or while the program is being stepped.

**ST diagram** A space-time diagram is drawn in a special window as the program progresses during the *replay* phase, giving the user an overview of the interactions between tasks (message sending and receiving events).

**database query window** Queries on the event database are composed in this window using Prolog queries (based on predicate logic).

**menu system** A cascading menu system is used to enforce the flow-chart described in figure 3.2.

**panic button** When things go wrong (like a deadlock has ensued), this facility, when activated by the user, will put all tasks of the program into a **suspended** state, giving complete control of each task to the **task controller** attached to it, thus allowing the user to probe the state of any task of the program.

An example of the description of such a GUI is found in figure 3.3.

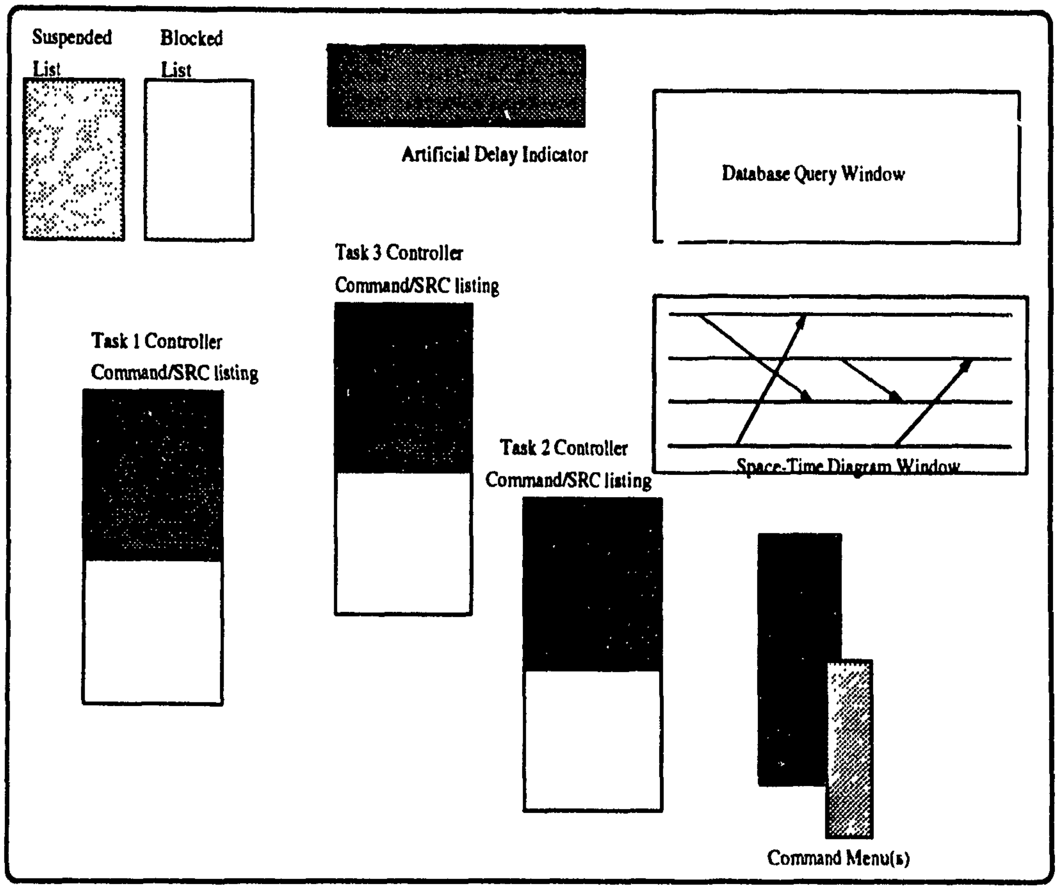
Suspended List
Blocked List
Artificial Delay Indicator
Database Query Window
Task 3 Controller Command/SRC listing
Task 1 Controller Command/SRC listing
Task 2 Controller Command/SRC listing
Space-Time Diagram Window
Command Menu(s)

Figure 3.3: Sample GUI during the *replay* phase

# Chapter 4

# Design, Interface and Integration of the Prototype Debugger

The design of a distributed debugger based on replay must support various requirements:

- the debugger must not mask errors that would otherwise have surfaced in the absence of the debugger. Thus the debugger must be designed to monitor programs in a decentralized manner, avoiding sending messages over the network as much as possible in order not to congest the network and to spread the debugger's probe across the distributed system. This must be done in order to avoid biasing the distributed program in any particular direction.
- the monitoring system must be efficient in order to minimize the probe effect at a given node, thus reducing the chance that a bias could take place over a short period of time before another probe causing a random delay can cancel out its effects.
- the user debugging one's program during the *replay* phase must not be aware, as much as reasonably possible, that reexecutions are controlled to recreate non-deterministic choices taken during the *recording* phase. There are two reasons for this. First of all, the user should not be aware of the inner workings of the debugger, since this can distract the user during a debugging session. For example, if the source code is augmented by the debugger for debugging

purposes, the user should not see the augmented code on the screen while "stepping". Secondly, during a coerced reexecution, the user should not be able to witness "impossible" occurrences, even though the coerced reexecution actually reproduces the same execution during the replay phase. For example, if a thread is waiting on a "condition_wait()" to enter a mutual exclusion construct (mutex), the debugger's replay controller must not allow the thread entry into the mutex until the appropriate signal is sent, even if the waiting thread is legally allowed to enter the mutex, according to the execution history.

- to incorporate as many off-the-shelf components into the design to expedite implementation and/or to unnecessarily avoid "reinventing the wheel".

An informal systems requirement specification (SRS) was written [20] to specify the basic facilities a "base" distributed debugger should have. An object-oriented design, based on the methodology expounded in [32], of the prototype was developed based on [20]. The implementation language was intended to be C++, but a lack of time to learn it, as well as some difficulty experienced in getting C++ to work in the Mach environment, influenced the decision to implement mostly in the C language. The latter problem was judged surmountable, but time constraints influenced the decision to abandon C++ more than any other reason.

A prototype (and numerous feasibility tests) has been implemented. The programming language used for the implementation was C. The platform that was used for this prototype consisted of one Sun 3/180 file server and four Sun 3/50's linked to each other on a local network (LAN). The operating system used was Mach 2.6 [1]. Two toolkits especially made for the Mach operating system that were used extensively were the C-Threads [6] and MIG (Mach Interface Generator) [9] packages. The application programs that can be used with the debugger are assumed to be written in C, with MIG and C-Thread support included. The implementation of application multi-threaded support was deferred to a later date.

## 4.1 Main System Structures

The debugger prototype is made up of seven basic components.

**Graphical User Interface and the Central Controller** As its name implies, the *Central Controller* (CC) coordinates *all* activities, subsystems, etc..., related to the debugger. The user's graphical user interface directs all requests for service to the CC. There is only one such module running at any one time on the distributed system, which implies a centralized form of control (see figure 4.1). The CC should not cause any additional probe effect during the *recording* phase since the CC is only active *before* and *after* execution of the program, starting and initializing various servers and modules, which operate without interacting with CC while a program is being monitored. During the *replay* phase, since the program has been rendered deterministic, centralized (*i.e.*, inefficient and probe-prone) interactions to the CC should not affect the outcome of the program, except for slowing it down somewhat. Furthermore, the mode of operation of most pieces of the debugger prototype behave in a decentralized manner during the *replay* and *recording* phases, rendering only a minimal slow-down during the *replay* phase. This efficiency becomes especially useful in large distributed systems.

**Transformed Program** The program to be debugged must be transformed specifically for either the *recording* phase or the *replay* phase prior to execution in the *recording* or *replay* phase respectively. Specific language and operating system primitives are transformed into calls to specific debugger library routines which are eventually linked to the application program (see description below and figure 5.3). Primitives are transformed for the purposes of monitoring, replaying the execution, receiving breakpoint instructions from the monitor, and for collecting events for the database. Furthermore, additional data structures are declared within the application tasks for use by the debugger, as well as the code for various debugging threads that are launched within each application task (see below).

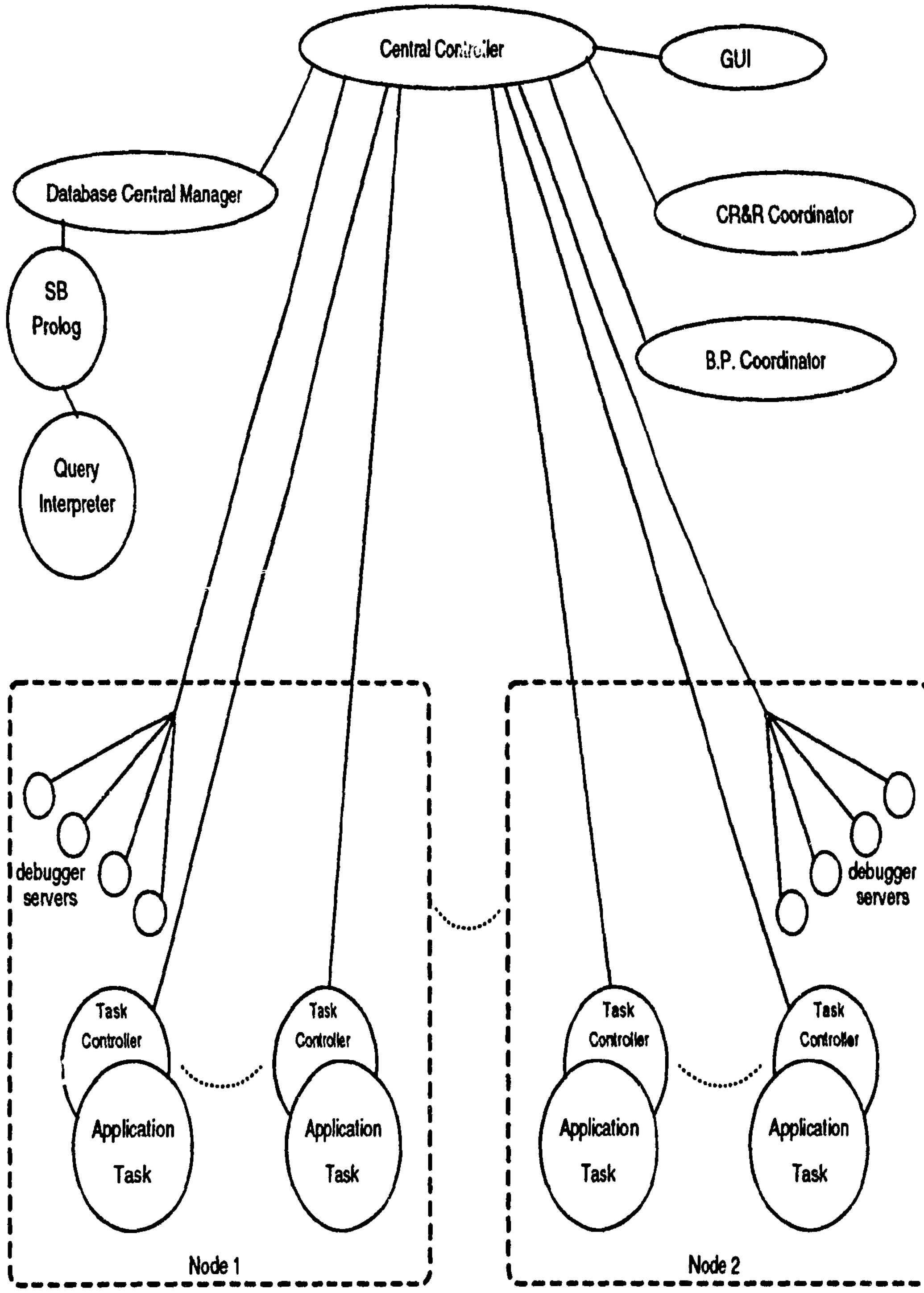

Figure 4.1: Central controller's relationship with various nodes

**Debugger Libraries** A transformed program will have had several of its operating system and/or language primitives replaced with specific library calls, which are then linked to the library routines. Using libraries is preferable to incorporating the additional code directly into the program code, since libraries can be made transparent to the sequential debugger (gdb) used to control every task in the distributed program being debugged during the *replay* phase by not generating a symbol table during their compilation. This is accomplished by compiling the library routines without the **-g** flag [28]. They are also conducive to greater general efficiency if the host operating system supports "shared libraries" (dynamically linked libraries).

**Task Controller** Every application task has a modified version of a sequential debugger (**gdb** [26], modified for thread support under Mach [4]) dynamically attached to it by the distributed debugger at the time of the (dynamic) task's birth. The Task Controller's use consists of setting breakpoints within particular threads in the application task, as well as gaining access to the task's address space. The human-machine interface is replaced with a machine-machine interface, enabling the distributed debugger to *remotely:*

- set breakpoints,
- determine if and when breakpoints were reached,
- examine the state of a given task,

as well as use the rest of a sequential debugger's features as components for setting up distributed breakpoints, checking distributed assertions, etc....

The distributed debugger's central breakpoint coordinator sets up a "distributed" breakpoint, in part, by setting up local breakpoints at the sequential debugger attached to the involved task(s). See section 4.2 for more details.

**Servers** On each node, for each program, there are four RPC servers which perform duties for *all* tasks of an application program *on that node*. Application programs send RPC messages to the servers via the debugging library requesting specific services related mostly to program monitoring and replay control. The different servers also use each other's services as required to minimize any duplication of services. A particular server's operations and knowledge are conceptually related, thus one can consider a server an *encapsulated* object. Since the public interface of a server hides its private representation and implementation, the servers follow the principle of *information* hiding. A description of the servers can be found in section 5.2.

**Debugger Threads Within the Application** On a task's birth, several debugger related threads are spawned within the address space of each application task. Figure 4.2 shows the various threads that are running within an application task being debugged, as well as the communication patterns between the various components of the debugger on given node with the central controller.

**Monitoring/Replay Utility Thread** As its name suggests, it performs a variety of services in support of the monitoring or replay control subsystem, depending on the phase the application program finds itself in (see section 5.3 for more details).

**Checkpoint Thread** This thread running within the application task will suspend the task to save the state (virtual memory and the CPU registers) of the task onto disk (checkpoint). This thread acts as a server in that it will take a checkpoint whenever it is requested to do so either within the thread itself (rules built-in) or from an external request (*e.g.*, the user manually request a checkpoint be taken at a breakpoint). See below for more complete details on checkpointing and how this thread fits into the picture.

**Rollback Thread** On demand, this server thread will rollback the computation (the task it is running in) to a previously saved checkpoint and restart the computation. For more details on how a distributed computation is rolled-back and coordinated using this thread, see section 4.3.

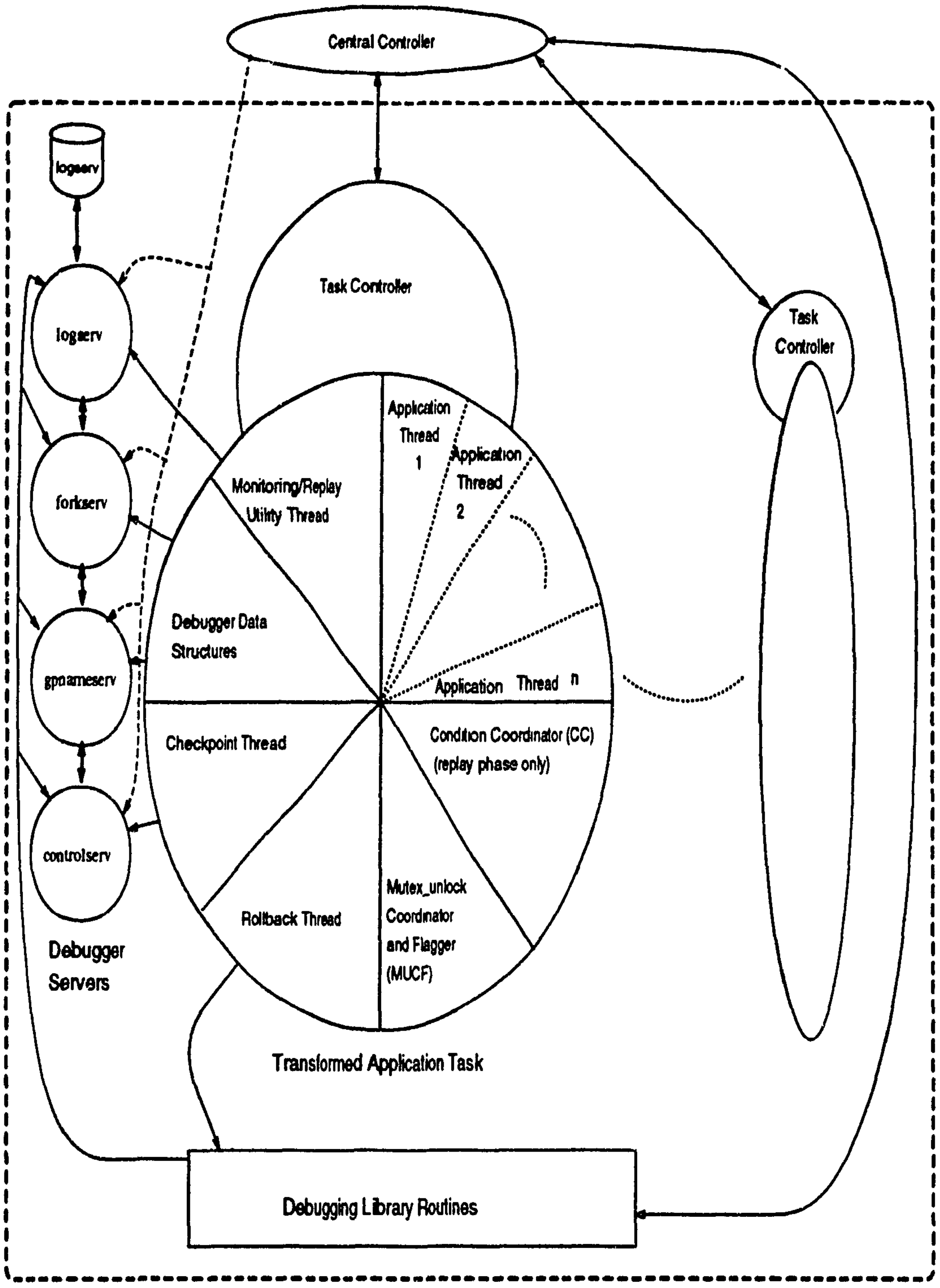

Figure 4.2: Workings of the debugger on a given node

**Mutex_unlock coordinator and flagger (MUCF)** During the *replay* phase, this thread detects an application thread's exit from a mutex, obtains the identity of the next thread that can have access to the mutex from the execution history and sends a "condition_signal" to all threads that are waiting to enter the mutex. Only the thread who has permission to enter the mutex will do so–the rest will go back to sleep.

**Condition Coordinator (CC)** This daemon thread ensures that the threads that sent a particular "condition_signal" during the *recording* phase will do so in the same order during the *replay* phase. The "condition_signal" will only be allowed to be sent when the threads that caught the signal during the *recording* phase are ready to catch the signal during the *replay* phase.

**Debugger Data Structures** During the phase-specific transformation of the source code of the application program to be debugged prior to the recompilation and commencement of the *recording* or *replay* phase, the debugger adds various data structures within the address space of each task. The additional data structures are for the specific use of various debugger threads within the task and certain linked debugger library routines when activated.

## 4.2 Breakpoints

Each task has attached to it a **task controller**, which is a sequential debugger with its man-machine interface transformed into a machine-machine interface in which some debugger controller task can send a given **task controller** a message containing a sequential debugger command such as a breakpoint setting command.

As mentioned in section 3.1, two types of distributed breakpoints are defined: Optimistic Consistent Breakpoints (OCB) and pessimistic causal breakpoints (PCB). When the user specifies the OCB and/or PCB, the set of all processes in the distributed program are partitioned into two sets $P$ and $Q$, such that for each process in P the user has specified a breakpoint.

### 4.2.1 Optimistic Consistent Breakpoints

The user sets a breakpoint via the **task controller**, which actually sets the breakpoint within the process. When the process hits the breakpoint, all the other processes in the system will eventually block at breakpoints set within them, or at points where they are blocked waiting to receive a message from another process that has hit a breakpoint or is blocked waiting to receive a message from another processes that has hit a breakpoint, or the remaining processes will execute until they finish. When the **central controller** of the debugger notices that all processes of the distributed system have become inactive (by virtue of hitting breakpoints, finishing, being blocked at a blocking msg_receive() primitive, or perhaps due to deadlock due to a programming error in the application program alone), the **central controller** of the debugger will send a message to all the **task controllers** attached to all the application tasks (see figure 4.1), asking the **task controllers** to put all the **blocked** processes into a **suspended** state (force a breakpoint) so that the user can investigate the global state of any task at the global breakpoint.

### 4.2.2 Pessimistic Causal Breakpoints

In order to have each process break at the earliest state that reflects all events that "happened before" the breakpoint, a **breakpoint coordinator** must allow processes to advance unhindered toward their pre-set breakpoints while only allowing processes without breakpoints specified within them to advance enough to unblock processes that have breakpoints specified within them, if the latter processes have not yet reached their breakpoints.

If more than two processes have breakpoints set (membership in set $P$), then these processes must be allowed to progress one at a time to their breakpoints, with the processes in set $Q$ (no breakpoints set within the process(es) in set $Q$) being allowed to move forward to unblock processes in set $P$. This is done to ensure a causal breakpoint if processes in set $P$ communicate with another member in set $P$. If a process in set $P$ hasn't reached its pre-set breakpoint, then it can only be suspended after a msg_send() primitive in order to preserve the state that "happens before" if another breakpoint in set $P$ is subsequently reached. The same PCB will be regenerated for every reexecution if the members of set $P$ progress toward their respective breakpoints, set in the code space, in lockstep (one after another, serially) in the same order every time.

Pessimistic causal breakpoints are implemented by having an application process *ap* detect a msg_receive(port) statement. If a PCB breakpoint has been set, the receiving process *ap* assumes that the sender task is suspended. The sender task must be advanced to its next msg_send() primitive, which should send a message to process *ap*. The process *ap* knows which process must send it a message by checking the *enqueuing history* recorded during the *recording* phase (see section 5.3.1) for the port it is receiving on from the REPCONTROLSERV server. Then, process *ar* sends a message to the **task controller** attached to the process that must now send a message to unblock process *ap*, telling it to execute the process until the next msg_send() primitive and then suspend itself once again.

## 4.3 Checkpoint, Rollback and Recovery

The **checkpoint** thread server will take the checkpoint of the task it is running within by *suspending all other threads* within the task (application and other debugger service threads), and forking a virtual memory image of the parent (including the stacks of **all** the threads and the current state of the CPU's registers). The checkpoint thread then *resumes all other threads* within the parent that were suspended prior to the checkpoint thread forking a virtual memory image (child) of the parent while concurrently writing (from the parent checkpoint thread) the virtual memory of the child onto secondary storage. The child task suspends itself immediately so that the checkpoint thread server within the parent can read and record the child's virtual memory image onto secondary storage.

The **rollback** thread server will realize rollback by forking a virtual memory image of the task it finds itself in. The forked task only serves as a template for the saved virtual memory image of the state the task is rolling back to. The forked task immediately suspends itself and the rollback thread server from the forking task replaces the virtual memory **completely** with the intended virtual memory image after killing all threads in the forked child. Each thread that existed when the task was checkpointed is re-created and restarted, and the state of the CPU registers is restored.

It is assumed that the type of distributed checkpoint and rollback scheme used is similar to that proposed by [29], commonly known as "optimistic recovery". This scheme must be integrated with the *replay* phase in order to be able to "rewind" the execution history when the distributed program is rolled-back. It is assumed that the necessary support needed to deal with duplicate and missing messages in a distributed environment are supported by the basic checkpoint and rollback scheme as proposed in [29] and that all ports are restored accordingly (see section 4.3.2).

It should be noted that the described checkpoint, rollback and recovery scheme was not implemented (lack of time). The following design is partially justified by some feasibility tests that were performed:

- the ability for a thread to checkpoint or roll-back the task it is running within was demonstrated,

- the ability to checkpoint a task on secondary storage was demonstrated, thereby allowing an unlimited number of checkpoints to be taken (limited only by the size of secondary storage). The use of the C-Threads library package [6] to implement a checkpoint, rollback and recovery scheme was found to be very useful.

- the ability to detach the **task controller** (modified sequential debugger [4]) prior to rollback and the ability to attach a **task controller** to a task that has rolled-back was demonstrated.

### 4.3.1 Checkpoint - Replay Phase Interaction

When a checkpoint is taken during the *replay* phase, several additional pieces of data must be logged in order to coordinate "rewinding" the execution history with the distributed state.

- All logical logs (execution histories) that are exclusive to the task being checkpointed (like the fork logs, system-call logs, etc... ) in which logs are normally entered by the task itself (via debugger library calls) have a **marker** uniquely identifying the checkpoint being recorded within the sequence of the log (implies location where checkpoint took place). All other logical logs that are not directly entered by the involved task (like the **enqueuing logs**, which the REPCONTROLSERV enters into the relevant logical log), are flagged by having the checkpoint server thread send a special message to all the enqueuing logs indicating that a certain task is undergoing a checkpoint. The recipient of the special message, an intercept port on the REPCONTROLSERV, will recognize the marker message and proceed to mark the involved log it is maintaining with a special marker indicating the checkpoint taken so execution histories can be rolled-back to the appropriate place matching the checkpoint desired.

- All the send rights to port(s) the task being checkpointed had successfully checked-in (but not yet checked out) to the local netname server, and thus sent to the local REPGPNAMESERV a mapping of the netname to the *virtual* port name the send-right capability refers to (via a debugging library) must be recorded. This is done so that the local REPGPNAMESERV and the *netname* server can be properly restored after a rollback to remap netnames to the virtual names of the send rights to ports they represent.

- The **task sequence number** in REPFORKSERV must be recorded so that it can regenerate the same *virtual* names for the newly forked tasks so that they can dynamically match-up with the appropriate execution history log for that particular task as recorded during the *recording* phase.

### 4.3.2 Rollback - Replay Phase Interaction

After a computation has been rolled-back to a checkpoint *ch*, the debugger must reset all execution history logs to the **marks** in the individual histories that are associated with checkpoint *ch*, and all the relevant re-execution controller threads (within the application tasks and the various server logs) must be re-initialized to the current value (at the time of checkpoint *ch*).

On rollback, the **task controller** (a modified *gdb* [26]) attached to each application task is detached (using *gdb*'s "detach" command) as a first step in the rollback procedure. Once a computation is rolled-back to a checkpoint, the **task controller** is once again reattached to each task using the modified *gdb*'s "attach" command.

### Restoration of Ports and Port Rights

For each checkpointed task, a list of the ports rights, their local name (as given out to the task's port name space by the O.S.) as well as their global virtual names are saved with the associated checkpoint. To restore the capabilities associated with the task at the time of the checkpoint, as well as the interception set-ups, the following steps are taken after the virtual memory, stacks and registers for the checkpointed task are restored:

1. All existing application ports and port name space of the restored task must be destroyed, since the port-name space is altered dynamically, and it may not be in the same state as it was during the instance that the checkpoint was taken.

2. For each port that a task had receive rights for, a port *p* is re-allocated by the rollback thread, and the (task-local) name of the port allocated is adjusted, if necessary, to the O.S. name the port was known as during the checkpoint. The restored port *p* is registered on the local netname server with a *special name* (*e.g.*, virtual port name - checkpoint ID) that is meaningful only to the checkpoint and rollback system so that all other tasks in the system that had send rights to port *p* at the time of the checkpoint can lookup and obtain the send rights (to restore their own port name space), as well as obtain the location (node-name) so that interception could be set-up immediately at the REPCONTROLSERV at the node where the receive rights to port *p* currently reside (see section 5.2.7).

3. All the send rights a rollback'ed task possessed to other application ports it didn't have receive rights to must be regenerated by having the task's rollback thread search each netname server (or until found) for the send rights registered by the rollback system using the *special name* that is associated with a particular checkpoint and the virtual name of the port. Once the send right is received (thus restored) in an application task, the rollback thread then proceeds to restore message interception for the send right it just restored by sending an RPC request to do so at the REPCONTROLSERV server at the node where the receive rights for the port reside.

4. All ports the task had checked into the netname server (but had not checked out) prior to the checkpoint must be rechecked into the netname server (under the same "name") by the rollback thread. By virtue of the previous step, the task should have all the send rights it held prior to the checkpoint to successfully restore the portion of the state of the local netname server that the rollback'ed task had affected (*i.e.*, all the netnames that the task had successfully checked in).

5. Finally, all the application tasks are resumed.

## 4.4 Database of Events

During the *replay* phase (database doesn't operate in the *recording* phase), all *global* events (see BNF in section 3.3.1) are transformed to send the event's occurrence (and its parameters) to the local **event-logger**, which will check if the event has been slated to be recorded in the database. *Local* events to be recorded are checked by the local **task controller** (modified sequential debugger) if they have been slated to be logged–if so, the **task controller** will send an entry to the database. Local events can be easily checked by the **task controller** by using its built-in conditional-breakpoint mechanism. Each node has a local database server in order to avoid unnecessary congestion on the network during reexecution.

Queries are composed at a central location (off-line) when the program in question is suspended (*e.g.*, a breakpoint) or has ended. A public domain Prolog compiler such as SB-Prolog [8] could be used as the query interpreter.

# Chapter 5

# Mechanics of Deterministic Re-execution

Ideally, the monitoring/replay subsystem of a distributed debugger must take into account every possible legal and illegal action a user application program may take, bounded only by the constraints of the language and operating system used in one's program. The amount of recorded data can be very large. An execution history is a collection of sequential history logs. The debugger's monitoring/replay subsystem must monitor (during the *recording* phase) and control the execution during the *replay* phase so as to reproduce the same environment interaction and, if the program itself is non-deterministic, the same program choices. Any operating system scheduling choices that may have influenced a non-deterministic program's behavior would also have to be monitored. Also, the recording and replay phases use mechanisms normally not expressible in the programming language itself, which makes both implementation and reasoning about it more complex [14].

This chapter describes, in detail, a mechanism of deterministic reexecution, how the non-deterministic choices are collected during the *recording* phase and how they are enforced during the *replay* phase. The interaction between the augmented source code of the application program, the debugger servers, the debugger libraries, etc..., are described. Deterministic reexecution is a service provided to the debugger. The monitoring/replay service developed targets a micro-kernel's capabilities, which encompasses more than just the process-model of single-threaded tasks communicating only via asynchronous messages, thus allowing this mechanism to be relevant to other models.

## 5.1 Overview and Assumptions

### 5.1.1 Approach Taken

In the literature, various approaches have been used for the development of a deterministic reexecution system for a distributed debugger. These approaches can be generalized into two distinct categories:

**implementation based** With this approach, the implementation of the run-time system, the operating system, and the compiler is modified to support *replay*.

**language based** This approach involves transforming the source code of the application program.

The approach taken in the prototype debugger is somewhat different to these approaches since the replay sub-system is designed to work at the *micro-kernel* level. The approach is partially language-based, since the program is transformed prior to compilation, and partially implementation-based, since the C-Threads library package used is altered, and a few specific operating system techniques at the system call level (port-capability manipulation) are used to transparently intercept messages in transit. The traditional argument against implementation-based replay systems is that much effort is required to port the system to various platform/language combinations. This argument is less relevant when applied to a micro-kernel. This is because the alterations in the C-Threads library is at the source code level (in user space) and no changes to the kernel are made (transparent interception of messages is accomplished by using system calls only).

### 5.1.2 Phases of Recording and Replay

A debugging session is split into two phases: the recording phase (monitoring) and the replay phase.

| PHASES | ACTIVITIES |
|---|---|
| Recording (Monitoring) Phase | Monitor execution of a target system<br>Log the execution (record non-deterministic choices) |
| Replay (Debugging) Phase | Check all non-deterministic behavior<br>Correct reexecution to correspond to execution history |

Table 5.1: Two Phases of the Testing and Debugging System

Typically, testing of a program is done with the debugger in *recording* mode. In this manner, if an error is detected during testing, the user can immediately go into phase 2 and deterministically reexecute the program as many times as it takes to find the bug using various debugging tools designed for cyclic debugging methodology.

In short, during recording, all non-deterministic behavior is monitored and recorded into various logical logs that symbolize the activities of particular non-deterministic events. All relevant program objects (tasks, threads, ports) are given *virtual* names during the debugger's *recording* phase in order to correctly map the objects to their respective execution histories when the *replay* phase is invoked. This is necessary because the ID's given by the operating system to allocated resources are not reproducible on subsequent reexecutions. During the recording phase, the collected execution history is put into a form that is meaningful to the replay phase (using a formatting convention).

During the *replay* phase, potential non-deterministic constructs in the code of the application being debugged are transformed. If execution of the transformed non-deterministic constructs are attempted, the replay system of the debugger will first check if the proposed activity is in the proper order and/or which choice in the non-deterministic construct was chosen during the recording phase. If there is a scheduling inconsistency with the recorded history, the replay system will reorder accordingly. If the reexecution is attempting to enter a non-deterministic construct, then the replay subsystem will ensure the same original choice is taken.

### 5.1.3 Target Environment

The mechanism for the replay system presented here makes a few assumptions about the platform in which the application programs will be written. Specifically, a message-based micro-kernel operating system is assumed, and the prototype is geared toward the Mach [1] operating system (version 2.5). Mach-2.5 can be considered a logical micro-kernel, even though physically it is not. Distributed programs are assumed to be written by using a combination of a sequential language (C), operating system primitives (Mach) to support at least concurrent constructs and capability-based message services, an RPC package (MIG), and a thread run-time library (C-Threads).

To support deterministic reexecution on this platform, the following software tools are used:

- The Mach operating system primitives port_insert() and port_extract(). With these calls, messages can be transparently intercepted by a **debugger** task by altering the capabilities of the sending or receiving task, giving the user the impression that messages are being normally sent.

- The GNU C-Preprocessor [27] is used to transform the source code of the application program to a form suitable for **recording** or **replay**. When the application program is compiled, the pre-processor is automatically invoked prior to actual compilation. Specific operating system and language constructs are transformed into calls to debugger-specific library routines.

- The MIG RPC package is used to build debugger servers (described later in section 5.2) that reside on each node of the distributed systems.

- The C-Threads package is used to render certain debugger MIG servers "multi-threaded". Furthermore, each application task is required to use C-Threads, even if the application is single-threaded, in order to accommodate several daemon threads, one of which is dedicated for recording/replay purposes.

### 5.1.4 Sources of Non-Determinism

Table 5.2 gives an brief indication of what type of non-determinism is possible within the confines of the application platform (as described in section 5.1.3). See Appendix B for more details:

| EVENT TYPE | POTENTIAL NON-DETERMINISM |
|---|---|
| message send | sending task & thread (enqueuing order at the port) |
| message receive | receiving task & thread (dequeuing order at a port) |
| netname lookup | success or failure code<br>if success, from which node |
| netname check-in | success or failure, in case of race |
| netname check-out | success or failure, in case of race |
| kernel calls | result of a non-det kernel call |
| timeouts | timed-out or not<br>for message sends and receives |
| thread synchronization | entry to mutex construct<br>list of threads that sent a signal<br>tally of threads which caught a signal |
| port sets | which port in set was dequeued |
| miscellaneous | external input and<br>random number generators |

Table 5.2: Potential Non-Determinism

Since multiple tasks can acquire send rights to a given port, the *enqueuing* order at a given port can alter from execution to execution. Therefore, the sender task of each message must be identified. In a multi-threaded environment, the sending thread must be identified as well, since a given task's send rights to a given port is accessible to all threads within that task.

For the receipt of messages, the dequeuing order of messages must be logged during *recording* and enforced during *replay*. In a multi-threaded environment, all threads within a task have the *potential* to dequeue from any port the task has receive rights to, thereby justifying the reason for logging the identity of the dequeuing thread. Furthermore, since receive rights can be transferred to another task, the identity of the "dequeuing" task must also be logged.

The *netname server* (see appendix A for description) system calls are generally non-deterministic. Checking a name into the server can fail if the same netname is attempted to be checked in (by the same or different task/thread) and it already exists on the target netname server. Furthermore, it is possible that a race condition may exist during a netname check-out in that a different task/thread will check-out a name from the netname server (the other task/thread(s) will receive an error code) during different executions.

Netname look-up's are also subject to race conditions (and are thus non-deterministic) in that a look-up can either be successful or not during different executions, depending if the task/thread that checks-in the name is delayed or not. Furthermore, if a broadcast look-up is involved, this is another source of non-determinism if more than one server has the searched-for netname. The result of the look-up and any ancillary data (such as the node where search was successful) is logged for subsequent enforcement during the *replay* phase.

Certain operating system primitives return non-deterministic results. For example, memory can be allocated at a certain specific location or the program can ask the operating system to allocate a contiguous segment at a location of its (the O.S.'s) choosing. In such a case, then during the *replay* phase, memory allocation must be coerced to occur at the same location.

Timeouts on message sending and receiving primitives are not guaranteed to recur from reexecution to reexecution. Therefore, it must be logged as to whether timeouts have occurred or not during the *recording* phase so as to regenerate the timeout, if necessary, during subsequent reexecutions.

If a task/thread is receiving messages on a port-set, it is not guaranteed that the sequential thread will dequeue in the same sequence from the various ports in the set. If more than one port in the set is non-empty, the operating system can make a non-deterministic choice as to which port to dequeue. If only one port is non-empty, the receiver must dequeue from that port, but communication delays can alter the possible port(s) a task/thread can dequeue at a given message-receive construct in the code across various reexecutions. Furthermore, ports may be added and removed from a particular port-set dynamically. Delays in a task/thread adding or removing a port from a port-set can also affect the possible port(s) a task/thread can dequeue at a given message-receive construct in the code across various reexecutions.

Thread synchronization is dependent on the scheduler as well as other miscellaneous delays. As a consequence, entries to specific *mutex* constructs by various threads can vary greatly across several reexecutions. The fact that threads can be created and destroyed dynamically adds to the complexity since a delay in the spawning of a new thread can affect the order in which threads gain entry to particular mutex's. In order for a reexecution to be deterministic, the various threads that were granted (exclusive) access to a given mutex construct must access the mutex construct *in the same order* as during the *recording* phase. Furthermore, as **condition_signals** to wake-up sleeping threads waiting on a *condition* variable can be sent to any one such thread waiting for such a signal (on a condition_wait) or to all the threads, it is imperative during the *replay* phase that the same threads that actually received the wake-up calls during the *recording* phase will receive the signal (at the appropriate place in their respective code) during reexecutions. Problems such as the fact that a thread that originally received a signal during the *recording* phase may not be asleep yet (or hasn't even been created created yet) due to various delays when the signal is sent during a reexecution must be overcome in order to render an authentic reexecution.

Other sources of potential non-determinism are random-number generators, external data input into the program (if not the same for each execution) and unprotected shared memory (*i.e.*, not properly protected by a mutex construct). The latter source of non-determinism often is influenced by the hardware's policy (or lack of it) on atomicity of operations on shared variables.

### 5.1.5 Virtual Naming System

A system to label (or name) system resources, specifically threads, tasks, and ports, is required in a replay system in order to name various logs of execution history so that the logs can be correctly matched up to the activities of various resources during the *replay* phase. The identity of resources such as task, threads, and ports is typically given by the operating system, but the identities given out are almost always different for each execution of a program, even if the program being debugged is deterministic. Thus, the system of resource naming must ensure that system resources are given the same *virtual* names during both the *recording* and *replay* phases.

## Threads

The threads in each task are given I.D.'s as they are created (within only a given task). Within each task is a debugger-specific integer variable `vir_thread_id` that is initialized to 0. Each time a thread is created within that task, the new thread is given the current value of `vir_thread_id` and it is incremented. The `vir_thread_id` is never reset (assuming that a task will never create more threads than can be described by an unsigned 32-bit integer). By default, the initial application thread of a task is assigned the name `0`. Also within each task is a debugger specific data-structure that is kept in the shared memory mapping existing threads in the task to their virtual names. A thread wishing to determine its virtual name (typically via a debugger library routine inserted during the transformation of the program) can look it up in this data-structure.

In order to give out the same virtual names to the same threads during the *recording* and *replay* phases, so as to refer to the correct thread during the *replay* phase, all thread_fork() and virtual thread-name assignments operations in a multi-threaded application program must be made to be *atomic* within each such multi-threaded task. This is achieved by protecting the two operations with a mutual exclusion construct (mutex) allocated for this sole purpose. During the *recording* phase, the virtual name of the parent thread that forked a child thread is recorded in the **log of thread creation** (see table 5.3). During the *replay* phase, the monitoring/replay subsystem *guarantees* that the same sequence of threads that forked a thread during the *recording* phase would fork them again in the same sequence so that the thread's granted *allocation number* would be the same during both phases of debugging.

| | |
|---|---|
| Who adds to log | task-thread that just forked a new thread |
| Kind of data | the virtual ID of the *parent* thread that just forked a thread |
| Who created log | Application task (via init_cdb()) when the task is created |
| Location of log | physical log file dedicated to a particular application task |

Table 5.3: History of thread creation

### Tasks

Since newly created application tasks on the Mach 2.5 operating system are given names by the operating system that cannot be reproduced on each reexecution, a system is needed to grant names to application tasks that will be regranted to the same tasks on each reexecution. This facility is needed in order to match a task with its respective execution history during the *replay* phase.

The responsibility for giving out identities for tasks lies with **forkserv** (see section 5.2.3), a server operating as a distinct (non-application) task on each node of a distributed system. Before dynamic allocation of a new task (either locally or remotely), all tasks that are about to spawn a new task must request permission to do so from the **forkserv** on the node they desire to spawn a new task on. This is because the new task spawned asks the **forkserv** it is created on for a virtual name. The newly spawned task must be guaranteed to be the only task making such a request to the local forkserv so that when it does make such a request, the **forkserv** will unambiguously send the new virtual name to the task (see sections 5.2.3 and 5.2.4. If there is no such guarantee of exclusive node-specific forking rights, then if two tasks are in the process of being spawned concurrently, their respective virtual names they receive during the *recording* phase from the **forkserv** may be different from the name they are given during the *replay* phase, depending on whose RPC to the **forkserv** arrives first. This situation would make it impossible to match the tasks with the correct execution history files, compiled in the *recording phase*, during the *replay* phase. In short, the FORKSERV ensures that the order of the birth of tasks belonging to a specific distributed program ***on a given node*** are the same during both the *recording* and *replay* phases. This is necessary because tasks are given **virtual** names (integers) as they are created, starting with "0" and incremented for each new task spawned on a node.

## Ports

When a task/thread allocates a port, it immediately gives it a virtual name. The virtual name of a port consists of the concatenation of four elements to ensure that the port has a globally unique name[1]:

1. the virtual name of the *thread* that allocated it,
2. the virtual name of the *task* in which the port was allocated,
3. the IP (Internet Protocol) number of the *node* in which the port was allocated,
4. the current value of the task's *port allocation number* `vir_port_id`.

As with the naming of threads, each task contains a debugger specific integer variable `vir_port_id` that is initialized to 0. Each time a port is allocated within that task, the new port is given the current value of `vir_port_id` and it is incremented, although this port allocation number becomes only one component in a port's virtual name. The `vir_port_id` is never reset (assuming that a task will never allocate more ports than can be described by an unsigned 32-bit integer). By default, the initial application port of a task is assigned the name "3" because the debugger reserves the first 3 values for other purposes (described below). The virtual name of a port must consist of the above mentioned four components in order to give the port an unambiguous global name throughout the distributed system. The task's port allocation number is not a sufficient global name, since other tasks also use a port allocation number scheme and thus several ports could have identical global names. To compensate, the virtual names of the task and thread that had allocated the port are concatenated to the port allocation number. Since there is still a possibility that

[1]The Mach kernel has a local port name space

identically (virtually) named tasks/threads exist on other nodes (it is assumed that tasks/threads do not migrate to other nodes) with identically named ports, the IP number of the node in which the port was allocated on is used as a component of a port's virtual name to render it globally unique. The virtual name of a port remains constant throughout the port's life, even if its receive rights are moved to another task on another node. If a port is deallocated, its name is not recycled.

In order to give out the same virtual names to the same ports during the *recording* and *replay* phase, so as to match up port activity with the correct execution history during the *replay* phase, all port_creation() and virtual port-name assignments operations in a multi-threaded application program must be made *atomic* within each such multi-threaded task. This is achieved by protecting the two operations with a mutual exclusion construct allocated for this sole purpose. During the *recording* phase, the virtual name of the thread that allocated a port is recorded in the **log of port creation** (see table 5.4). During the *replay* phase, the monitoring/replay subsystem *guarantees* that the same sequence of threads that allocated ports during the *recording* phase would allocate them again in the same sequence so that the port's granted *port allocation number* would be the same during both phases of debugging.

| | |
|---|---|
| Who adds to log | task-thread that allocates a port |
| Kind of data | the virtual name of the task and thread that allocated the port |
| Who created log | Application task (via init_cdb()) when the task is created |
| Location of log | Physical log file dedicated to a particular application task |

Table 5.4: History of port creation

Within each application task is a debugger data structure that maintains a mapping between its local (O.S. given) port rights and their associated global port names.

### 5.1.6 Transparency of the Debugger

There are two issues that determine whether a d bugger interferes with an application. During the *recording* phase, the main concern is the ***probe*** effect [15]. During the *replay* phase, the priority becomes trying to hide the fact from the person using the debugger that the observed execution is really a *controlled* execution.

In general, the probe effect will mask bugs/errors only if the probe effect is heavily biased toward a particular direction. For example, if a disproportionate amount of a debugger is centrally located on some node that also hosts portions of the distributed application program being monitored, the debugger on this node may disproportionally slow down the application program on that node.

If the probe effect is reasonably random, it should only slow down the program slightly without masking any bugs/errors. During the *recording* phase, the *probe* effect is minimized by using efficient algorithms and methods in the debugger, most notably in the *logserv* server (see section 5.2.1). Once the *recording* phase has begun, the collection of monitoring data is done on each local node in that each node has identical monitoring servers and local secondary storage, thus monitoring data is not all sent to some central location. Each application task also works during the *recording* phase to record all non-deterministic activity by virtue of the fact that the application task's source code is transformed to perform tasks related to debugging. Thus each task carries an equal burden related toward monitoring, and on average, the probe effect should not be biased in any manner.

During the *replay* phase, the data-structures and macros used by the debugger within an application task are declared in each application program with a single statement (`#include "cdb.h"`, see figure 5.2). The library calls an application makes (via macros) to special debugger routines are transparent to the GDB (task controller) interface [26] while *stepping*. During both phases, the messages the application tasks pass between themselves are transparently intercepted and forwarded by *controller* tasks in that the sending and receiving tasks are unaware of the interception. Figure 5.1 describes how the *transparent* interception scheme is set-up. Furthermore, the application message being passed is unaltered—all control information is passed in separate **control** messages and the user has no way of even knowing of the existence of the various control messages.

### 5.1.7 Miscellaneous Assumptions and Policies

**dynamic creation of resources** The monitoring and the replay subsystem is able to support the dynamic c.eation and destruction of ports, tasks, and threads.

**Start-up of a distributed program** Since the dynamic creation of tasks is supported by the debugger and the host O.S., a distributed program always starts out as a single single-threaded task. Subsequent local and remote spawning by any application task will create a parallel/distributed program. During the *replay* phase, the initial task must start on the same node as during the *recording* phase in order to properly regenerate the same virtual names generated during the recording phase.

**application threads** It is assumed that the user will always employ the services of the C-Threads [6] package for multi-threaded programming and will refrain from using the low-level Mach thread primitives [2] directly. This is because the monitoring/replay subsystem monitors and replays thread activity at the C-Thread level. It is reasonable to assume that most programmers will not use the low-level thread primitives, since they were primarily designed to be used as building blocks for thread packages [17] and custom synchronization constructs.

**shared memory protection** It is assumed that the application programs have not left any shared variables (between threads) unprotected and thus subject to possible violations of hardware sequential consistency. This is not a realistic assumption. There don't seem to be any practical solutions to this problem in the literature.

**name servers** Since the micro-kernel being used is Mach 2.5 [2], it comes with two distinct name servers, the "Environment manager" [30] and the "netmsgserver" [18]. It is assumed that only the "netmsgserver" is used by application

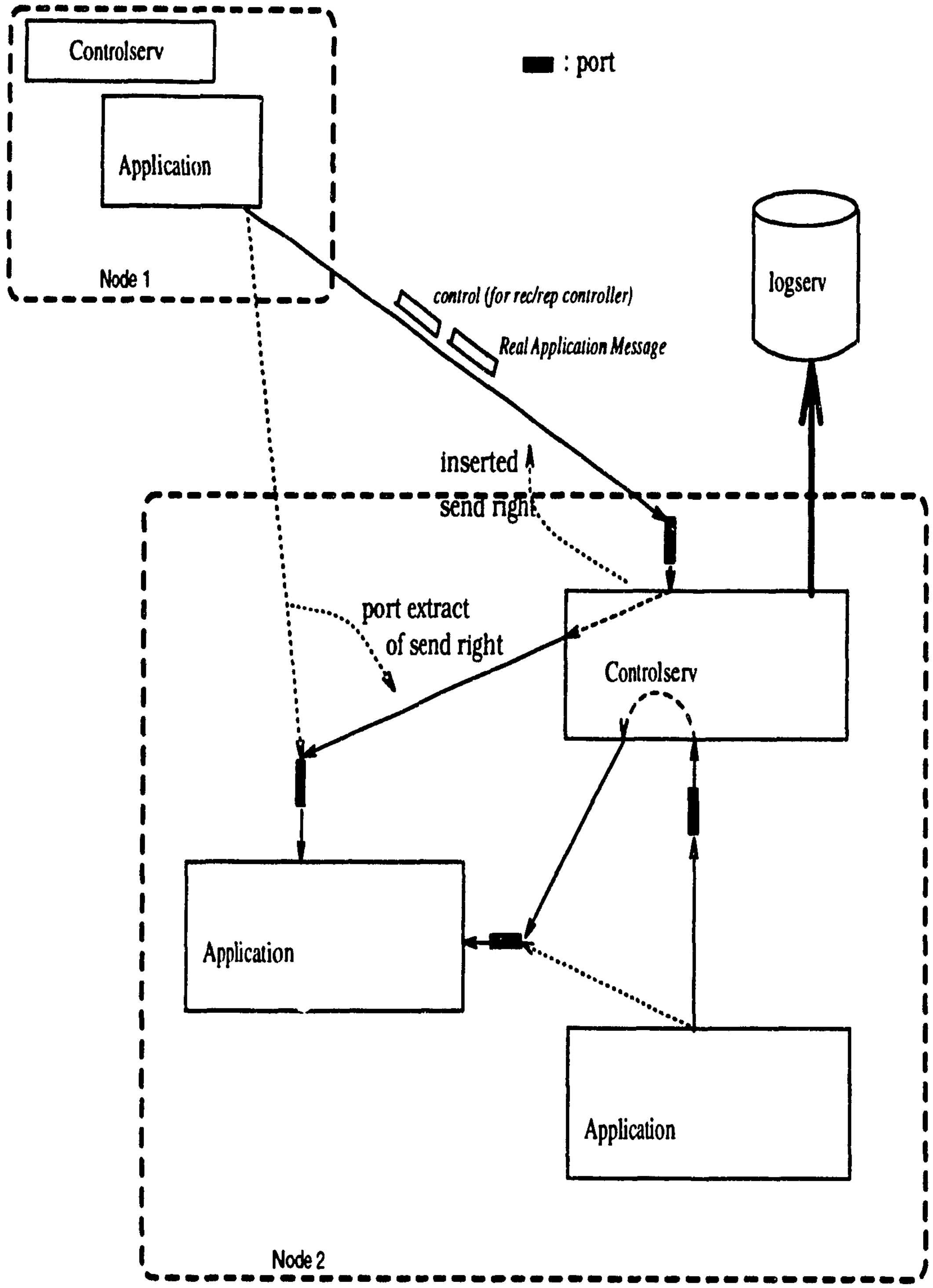

Figure 5.1: Interception Scheme

programs, since the "netmsgserver" services are a super-set of those of the "Environment manager".

**restriced kernel calls** Certain kernel calls are not permitted in a user's application program. Since the transparent message interception is achieved by using Mach's port_extract() and port_insert() system calls, their use is forbidden in an application program. Furthermore, it is assumed that no application task will send kernel calls to kernel ports that are not their own, since this makes tracking non-determinism much more difficult, and the necessary support has not been built into the debugger. The two restrictions just described should not affect most applications, since the described kernels calls are most often used by debugger developers or kernel emulation library developers.

As well, Mach external memory manager calls [2] are not permitted since they are presently unsupported in the replay system (due to time constraints).

**remote spawning of a task** Since Mach 2.5 does not come with an implemented primitive for forking a task on a remote node, and there are some technical difficulties in forking a task on the local node (see below), it is assumed that the application programs will use the following Unix C-Shell primitives for spawning new tasks:

**local fork:** `system(` *executable_program* `)`

**remote fork:** `system( rsh` *node* `-n` *executable_program* `)`

The technical difficulty of forking a task locally is that the **task controller** attached to the task will fail on a fork. The ultimate solution to this problem is to detach the **task controller** from the forking task temporarily until the fork operation has completed, and subsequently reattach the parent task.

**additional daemons** Within each application task, there will be various daemons such as a checkpoint, rollback, among others (see section 4.1 and figure 4.2). All such daemon threads are created using "C-Threads" primitives upon program initialization.

```
#include "cdb.h" /* first declaration */
/* other declarations go here */

main() {

  /* declarations */

  init_cdb(); /* first line of code */

  /* program goes here */

}
```

Figure 5.2: Format of an executable program

**standard format of an application task** Each executable program that makes up a distributed program must have the format described in figure 5.2.

Compilation involves linking to a special library (`-lcdb`) and defining whether the program is being transformed for the *recording* or *replay* phase (*i.e.*, `-DRECORDING` or `-DREPLAY`) (see figure 5.3). Furthermore, the application program must be given a large random identifier (a large unsigned integer) during compilation (*i.e.*, `-DID=`*unsigned integer*), the same number during the *recording* and *replay* phase, so that 1) more than one invocation of the debugger can exist on the system concurrently (by the same or different users[2]), and 2) so distinct execution histories can be labeled as belonging to a certain application program.

**compiler and "local" debugger** Each task will have *attached* a sequential debugger **gdb** [26], which was modified for multi-threaded debugging [4] and was subsequently modified by replacing the human-machine interface by a machine-to-machine interface. Furthermore, it is assumed that the application programs will be compiled with **gcc** (version 1.3 or 1.4) [28] [27]. This is done because the symbol table that **gcc** produces is compatible with **gdb** [4] and debugger li-

[2]debugger service ports are given netnames that have as components the application program's identifier

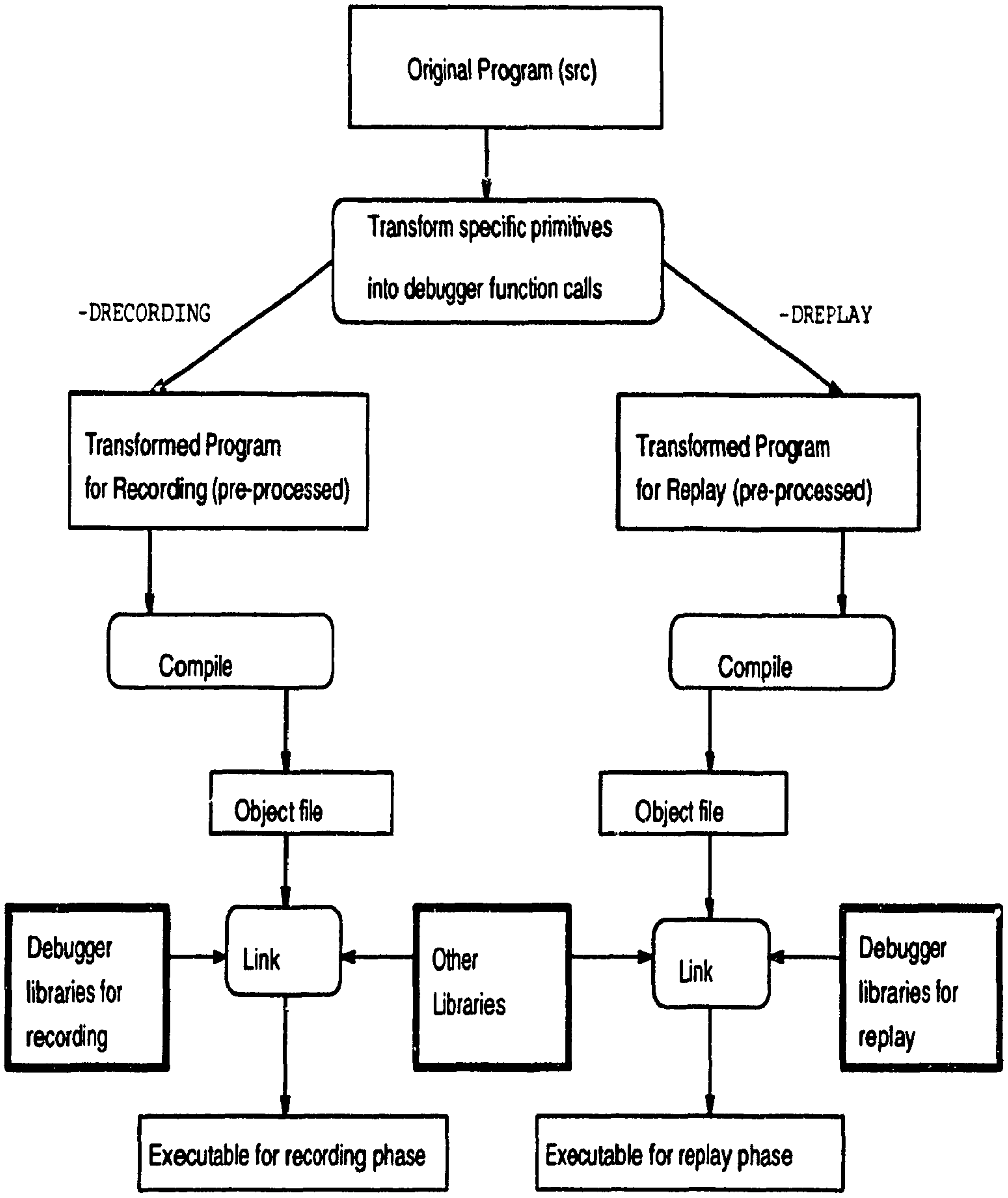
Original Program (src)
Transform specific primitives into debugger function calls
-DRECORDING
-DREPLAY
Transformed Program for Recording (pre-processed)
Transformed Program for Replay (pre-processed)
Compile
Compile
Object file
Object file
Debugger libraries for recording
Link
Other Libraries
Link
Debugger libraries for replay
Executable for recording phase
Executable for replay phase

Figure 5.3: Transformation of the Application Program

braries compiled with it do not generate any additional debugging information, thus rendering the debugger library routines transparent to **gdb**.

## 5.2 Standard Servers

Upon debugger invocation for a new program, four distinct phase-specific (*recording* or *replay*) servers, which directly support the *recording* or *replay* phase mechanism, are initiated on each node of the distributed system. The servers are initiated prior to the commencement of the application program to be monitored (or replayed). Figure 5.4 describes the interface between the debugger servers and the application program. Each set of servers, which are dedicated to a specific program, have uniquely named service ports (incorporating the program's ID) so application programs and other servers (via debugger library routines) can call the correct server, if more than one distributed program is being debugged on the system at the same time. For the *recording* phase, the servers are called LOGSERV, FORKSERV, GPNAMESERV, and CONTROLSERV. For the *replay* phase, the same names are used, but a "**rep**" prefix is added to the aforementioned names.

### 5.2.1 Logserv

The LOGSERV provides a generic logging service during the *recording* phase. In short, the LOGSERV manages the entry of large quantities of small pieces of data (as it is collected during the *recording* phase) into specific labeled logical sequential logs which are intertwined within a sequential physical file. No assumptions are made as to how many logical logs will occupy a physical log, nor are any assumptions made as to when a logical log begins within a physical log (*i.e.*, logical logs are created and named dynamically). The logical logs consist of a series of primary **elements**, which can be a non-empty ordered set of integers and/or a non-empty ordered set of character strings (each no longer than 19 characters, plus the *null character*) (see figure 5.5). Each element has two pointers: one to link the log elements to create a logically sequential log, and the other pointer so that a set of data (*e.g.*, the names of all the threads that caught a broadcast signal) can be grouped together and still

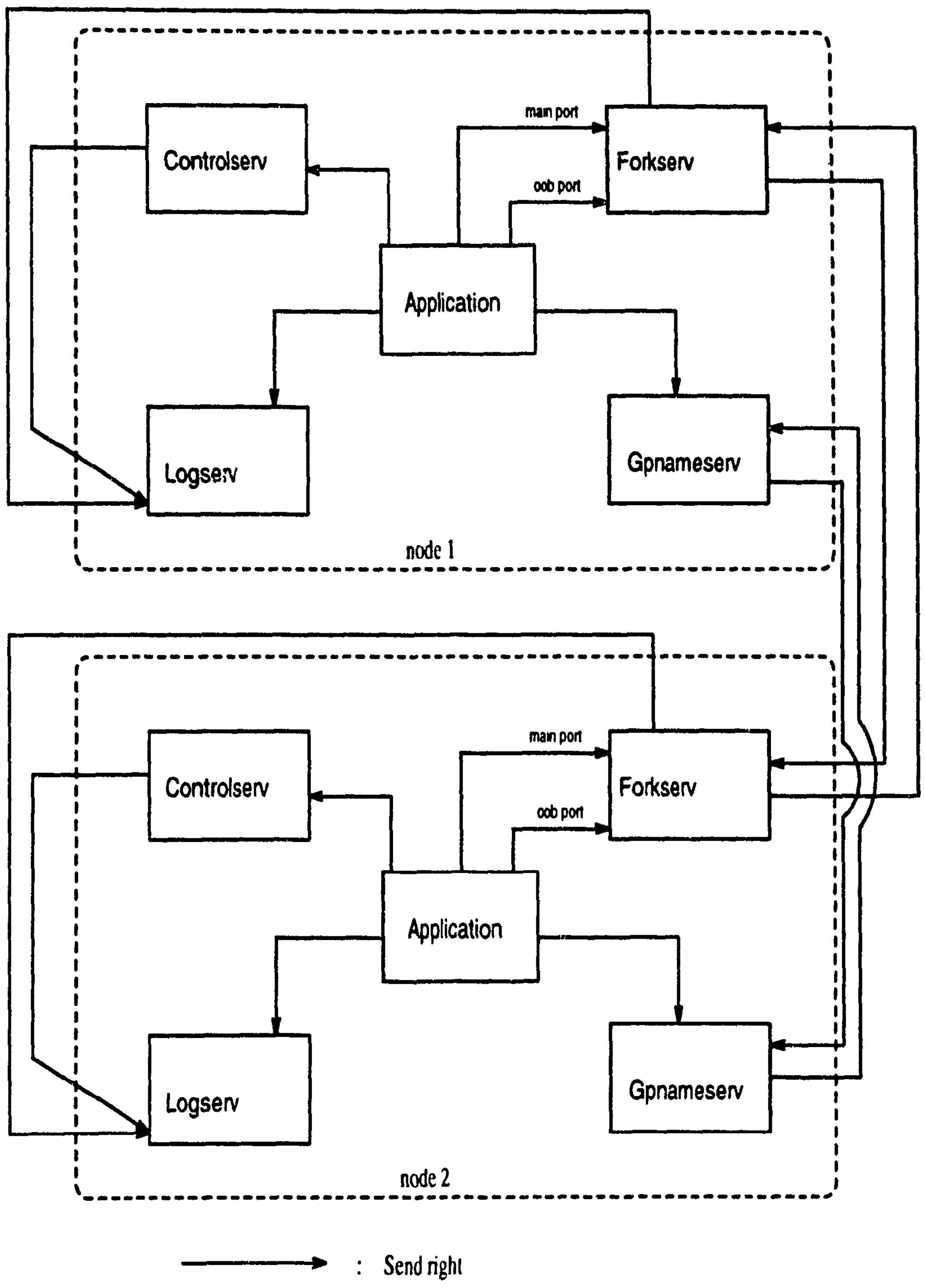

Figure 5.4: Configuration of debugger servers

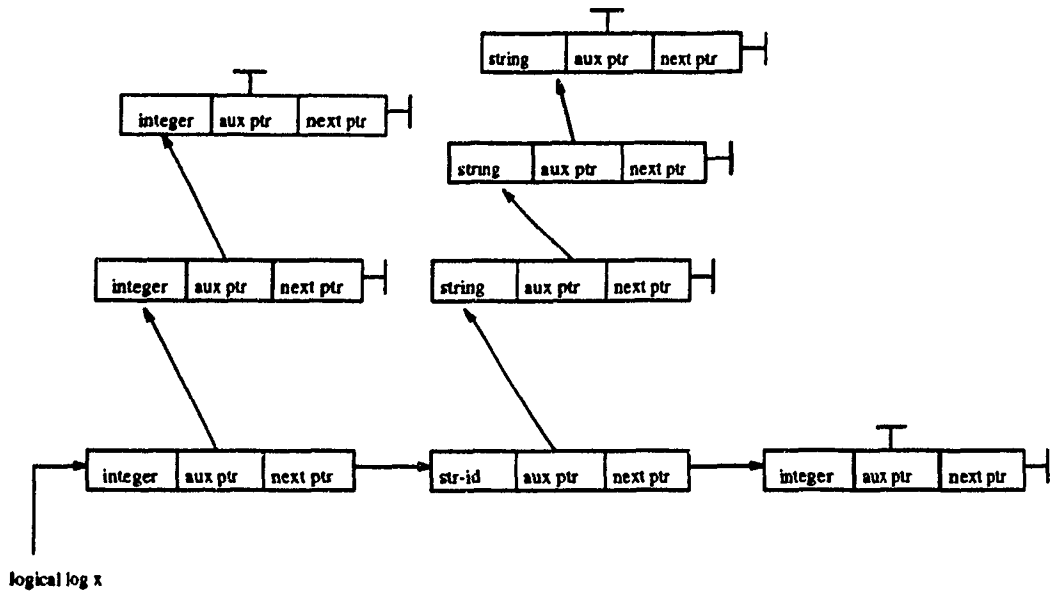

Figure 5.5: Basic elements of a log

be considered to belong to one composite element of a logical log.

### Interface

The services offered by the LOGSERV are as follows:

**new_physical_log(server_port, physical_log_id)** Open a new physical log file with the name physical_log_id.

**close_physical_log(server_port, physical_log_id)** Close an existing physical log file with the name physical_log_id that the specified LOGSERV is managing. This call must be invoked at the end of the recording phase so as to flush out all logical log buffers to the file proper.

**new_logical_log(server_port, physical_log_id, logical_log_id, log_type)** Open a new logical log file with the name logical_log_id *within* a specified existing physical file.

**add_to_logical_log(server_port, physical_log_id, logical_log_id, data,log_type)** Add a log **element** to a specified logical log within the specified physical log or append a new integer or string to the last log **element** (an ordered set of integers or string).

### Noteworthy Features

- The LOGSERV dynamically creates a log-file that is meaningful to the REPLOGSERV. This implies that all individual **elements** of each logical log are linked to each other in the order they are logged, and a separate file of logical "start" locations (of various logical logs) is constructed on-the-fly. No intermediate processing of the monitored data is required before it can be used for the *replay* phase.

- A circular buffer equivalent to 4 virtual memory pages is maintained for each physical log the LOGSERV is maintaining in order to minimize the number of system calls used to write monitored data to permanent storage, thus reducing the probe effect. When the circular buffer becomes full, the oldest window is flushed to permanent storage, thus freeing a window for subsequent new logged data. The fact that a window's size is equivalent to the operating system's page size implies an efficient transfer to permanent storage.

- The LOGSERV explicitly converts all logged data to "character" type in a meaningful format so that the REPLOGSERV can interpret it. The "character" format is used so that the REPLOGSERV can map the physical file of logs directly into its virtual memory (using Mach's map_fd() primitive [2]) and allow the operating system's virtual memory system to handle the transfer between memory and permanent storage in an optimal manner.

## 5.2.2 Replogserv

The REPLOGSERV server manages the retrieval of log elements from the named physical/logical log, which is kept in a sequential physical file that is intertwined with several logical logs. The REPLOGSERV keeps track of the last element (and sub-element) retrieved from each log and it always advances the log forward (never backwards) after each retrieval.

### Interface

The services offered by the REPLOGSERV are as follows:

**rep_open_physical_log** (serv_port, physical_log_id): Open a physical log file with the name physical_log_id.

**rep_close_physical_log** (serv_port, physical_log_id): Close an existing physical log file with the name physical_log_id that the specified REPLOGSERV is managing.

**rep_new_logical_log** (serv_port, physical_log_id, logical_log_id, logical_log_type): Open a new logical log file with the name logical_log_id.

**rep_get_logical_log_element** (serv_port, physical_log_id, logical_log_id, data, log_type): get the next element (or sub-element) in the logical log specified.

### Noteworthy Features

- maps the physical file of logs directly into its virtual memory and allows the operating system's virtual memory system to handle the transfer between memory and permanent storage. This is achieved by using the map_fd() primitive within the Mach operating system [2], which makes the transfer very efficient.
- shields the user from the complexities of maintaining the "current location" of each logical log.

### 5.2.3 Forkserv

The purpose of this server is hand out "virtual" names to all tasks of an application program so that the same instances of tasks named during the *recording* phase will be given exactly the same names during the *replay* phase. This is necessary in order to map instances of tasks to their respective execution histories (that were logged during the *recording* phase) when the tasks are spawned during the *replay* phase. The names of resources (*e.g.*, tasks, threads, ports) that operating systems hand out are not reproducible from execution to execution, thus a **virtual** naming system is needed, not least of which is for naming tasks.

In short, the FORKSERV must:

- ensure that no forks (spawning of new tasks) occur concurrently so that the report to the forkserv concerning a forking operation (via a message) is unambiguous (no race conditions to the FORKSERV's RPC service port due to other tasks which are spawning child tasks on the same node from the same or different nodes).

- ensure that only one *newly* forked task can send a message to the forkserv asking for a **virtual task id** at any time (avoid race conditions). Recall that when a task is spawned, it asks for an ID itself via the init_cdb() debugging primitive, which is supposed to be on the first line of the source code.

- in general, the sequence of task creation on each node, whether initiated locally or remotely, is logged. All task spawning activity on each node is serialized, if necessary. For locally initiated forks, the virtual task and thread ID of the forker is logged. If the forking initiator is remote, then only the node name of where the remote log was initiated from is logged by the forkserv on the node where the new task was created on. During the *replay* phase, the same tasks are spawned in the same order as during the *recording* phase in order to grant the same **virtual** names to the application tasks during both phases, since names are distributed as new tasks are created, and the names are taken from a sequence of integers.

- each *forkserv* has two service ports, each serviced by a separate thread:

  1. the main service port is where application tasks (via macros) send requests for exclusive forking rights. The FORKSERV sends an acknowledgment message to inform the requester of when it has been granted rights. The *forkserv* does not dequeue a request from this port if some task already has exclusive fork rights for that node.

2. the out-of-band service port is where application tasks send requests to the *forkserv* thai must be serviced immediately (*i.e.*, the thread servicing this port always responds to any requests on this queue immediately). The requests that must be responded to immediately include:
   (a) a task relinquishing its exclusive forking rights for a node (after it has spawned a new task), and
   (b) a newly spawned task asking for a new virtual ID when it is just spawned.

The following is a brief description of the two threads that service the two service ports of FORKSERV. The *main* service port is where tasks send requests for forking permission (see Algorithm Z). If a remote fork is requested, a remote FORKSERV will send a request to the *main* port to request permission for the requesting task that wants to fork.

In short, the task requesting a remote fork must get exclusive forking rights on both the local and the remote nodes (via the FORKSERV's on both nodes) before it can spawn a new task. This is necessary to ensure that the forking sequence is properly recorded during the *recording* phase and that the same forking sequence is enforced during the *replay* phase so that tasks are given identical **virtual** names during both phases. The out-of-band port is where tasks (or remote FORKSERV's) send indications that they have just forked and are now surrendering their exclusive fork rights (algorithm Y). A separate port for the latter purpose is necessary since there may be multiple fork-right requests blocked-up in the main service port. Figures 5.6 and 5.7 describe the protocol that is followed involving FORKSERV when a task intends to spawn a task locally or remotely, respectively.

```
Z1: FOR (;;)
Z2:     IF (go_to_sleep_main_loop)
Z3:         condition_wait(wake_up_call_for_main_loop).
Z4:         go_to_sleep_main_loop = FALSE.
        ENDIF
```

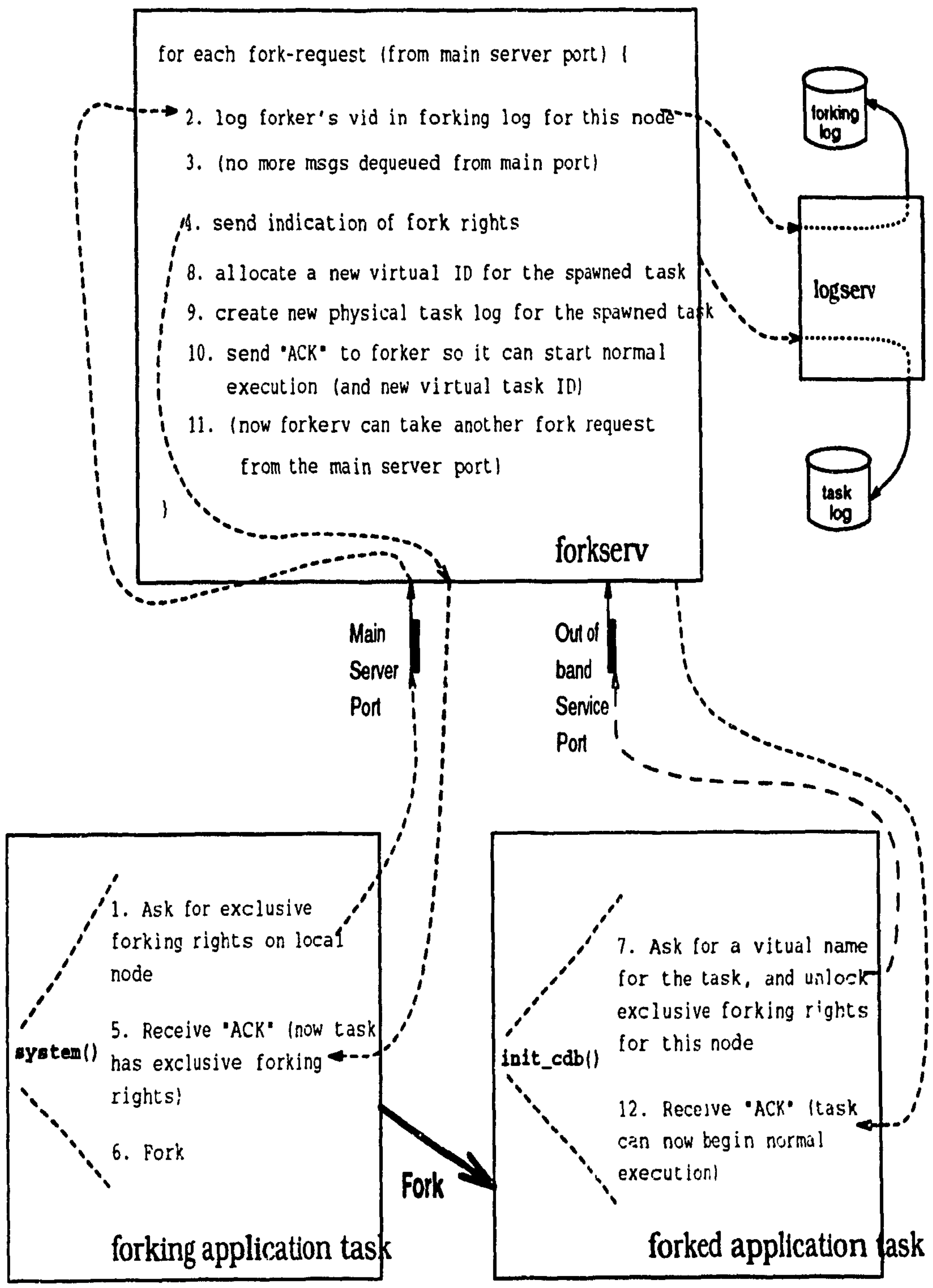
for each fork-request (from main server port) {
2. log forker's vid in forking log for this node
3. (no more msgs dequeued from main port)
4. send indication of fork rights
8. allocate a new virtual ID for the spawned task
9. create new physical task log for the spawned task
10. send "ACK" to forker so it can start normal execution (and new virtual task ID)
11. (now forkerv can take another fork request from the main server port)
}
forkserv
forking log
logserv
task log
Main Server Port
Out of band Service Port
1. Ask for exclusive forking rights on local node
system()
5. Receive "ACK" (now task has exclusive forking rights)
6. Fork
forking application task
Fork
init_cdb()
7. Ask for a vitual name for the task, and unlock exclusive forking rights for this node
12. Receive "ACK" (task can now begin normal execution)
forked application task

Figure 5.6: Forking on the local node

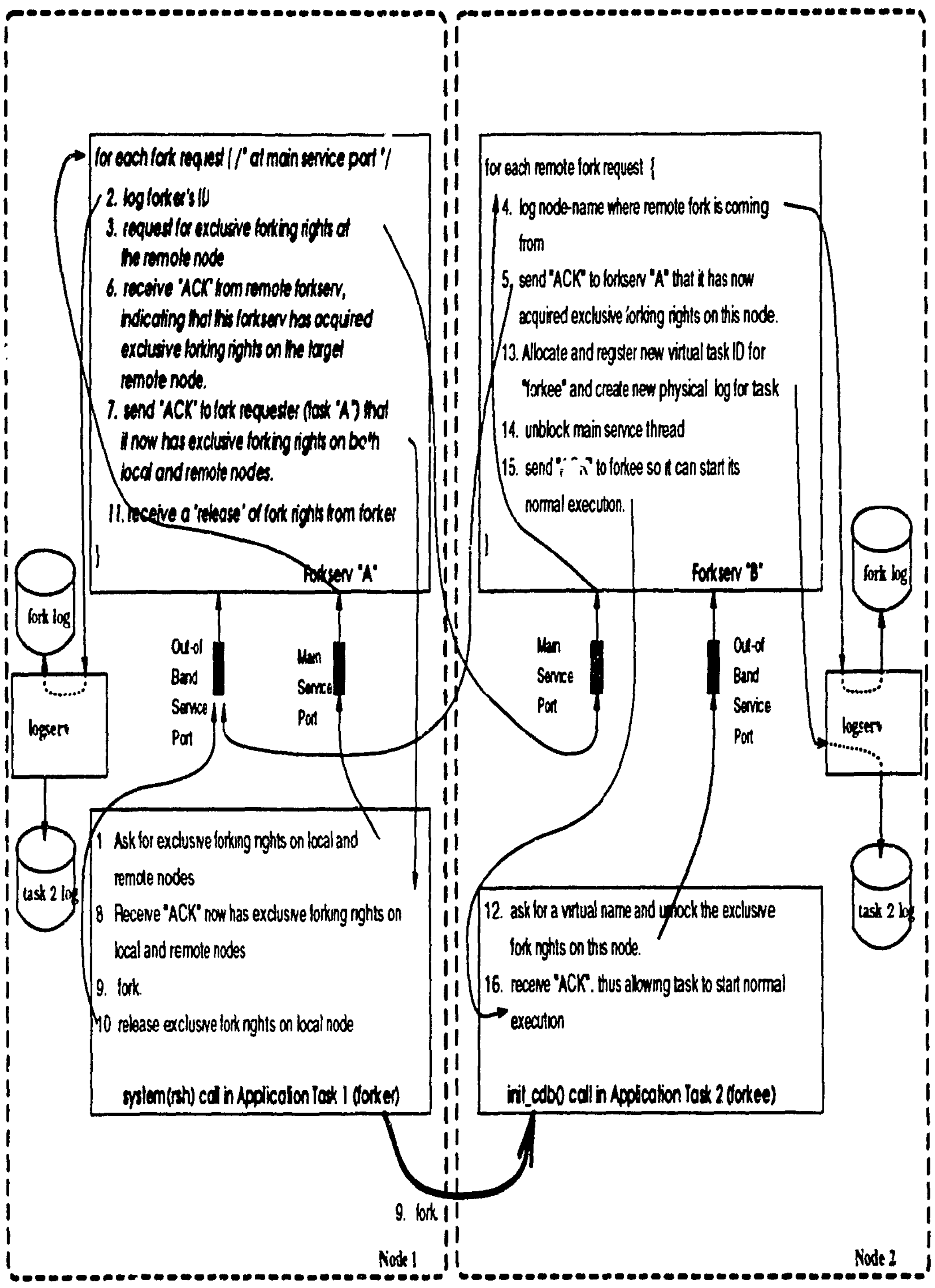
for each fork request { /* at main service port */
2. log forker's ID
3. request for exclusive forking rights at the remote node
6. receive "ACK" from remote forkserv, indicating that this forkserv has acquired exclusive forking rights on the target remote node.
7. send "ACK" to fork requester (task "A") that it now has exclusive forking rights on both local and remote nodes.
11. receive a 'release' of fork rights from forker
}
Forkserv "A"
for each remote fork request {
4. log node-name where remote fork is coming from
5. send "ACK" to forkserv "A" that it has now acquired exclusive forking rights on this node.
13. Allocate and register new virtual task ID for "forkee" and create new physical log for task
14. unblock main service thread
15. send "ACK" to forkee so it can start its normal execution.
}
Forkserv "B"
Out-of Band Service Port
Main Service Port
Main Service Port
Out-of Band Service Port
fork log
logserv
task 2 log
fork log
logserv
task 2 log
1 Ask for exclusive forking rights on local and remote nodes
8 Receive "ACK" now has exclusive forking rights on local and remote nodes
9. fork.
10 release exclusive fork rights on local node
system(rsh) call in Application Task 1 (forker)
12. ask for a virtual name and unlock the exclusive fork rights on this node.
16. receive "ACK", thus allowing task to start normal execution
init_cdb() call in Application Task 2 (forkee)
9. fork
Node 1
Node 2

Figure 5.7: Forking on the remote node

```
Z5:       dequeue a request from the main request queue.
Z6:       result = procedure_requested.
          /* see algorithms W and U on pages 71 and 72 respectively */
Z7:       send back reply message to client.
    ENDFOR

Y1: FOR (;;)
Y2:       dequeue a request from the oob request queue.
Y3:       result = procedure_requested.
          /* see algorithms X and V on pages 70 and 72 */
Y4:       send back reply message to client.
    ENDFOR
```

The interfaces (and their associated algorithms) offered by the FORKSERV are as follows:

The local_unlock_fork_lock() primitive is typically called by a task when it is just spawned *before* executing any of the task proper (the initial thread, that is) to inform the local FORKSERV of its birth, thus releasing the exclusive forking rights it had on this node.

```
 X: local_unlock_fork_lock(local_forkserv_oob_port,
                           pid_of_the_newly_spawned_task,
                           virtual_thread_id_of_the_newly_spawned_thread)
X1: grant a new v_task_id from the current distribution.
X2: put mapping of "v_task_id to pid" into forkserv's data space.
X3: increment forkserv's current distribution v_task_id.
X4: start a new physical log for the new task (call local LOGSERV).
X5: IF (forker_node != NULL) /* set by remote_get_lock_to_rfork() */
        /* forkserv's variable indicates that
        this new task was remotely forked */
X6:     log the node_name where the remote fork originated on.
```

```
     ENDIF
X7:  indicate to thread servicing main fork-request that it may now be
     free to dequeue another request for exclusive_fork_rights.
X8:  send back ACK to calling task that it is free to start with its
     normal execution now.
```

When a task wants to fork remotely, it calls `local_get_lock_to_rfork()` (see algorithm W) to get forking rights on the local FORKSERV and the FORKSERV then calls the remote FORKSERV to obtain forking rights there using this RPC:

```
 W:  local_get_lock_to_rfork(local_forkserv_main_port,
                             forker_real_pid,
                             forker_virtual_thread_id,
                             destination_node)
W1:  v_forker_task_id = lookup_forker_task_id.
     /* log the v_task_v_th of the forker to the
     local_log_of_fork_sequence (for this node) for this program */
     add_to_logical_log( local_logserv,
                         TASK_CREATE_LOG for this node,
                         virtual_task_thread).
W2:  IF (remote_rsh)
W3:      remote_get_lock_to_rfork(forkserv on remote node, my_node_name).
         /* acquire exclusive forking rights on remote node, and provide
         requester the name of the node from which remote forking will
         take place */
     ENDIF
W4:  tell main server of forkserv servicing main queue not to dequeue
     any more requests from the main queue after this servproc returns.
     /* since at this point, the task that made this request now has
     exclusive forking rights on this (and the remote) node */
W5:  send back ACK message to requester that it now has exclusive
```

forking rights for the type of fork (local or remote) it had asked for.

When the forker task has finished forking remotely, it must release the exclusive forking rights it has on the local node (the remote node's exclusive forking rights will be released when the newly spawned task sends a special message to the FORKSERV on the remote node), so it uses the following RPC to accomplish this (algorithm V):

```
V: remote_unlock_fork_lock(remote_forkserv_oob_port,
                           forker_real_pid,
                           forker_v_thread_id)
V1: unblock the main forkserv thread servicing the main port.
    /* no one is holding exclusive forking rights for this node,
    so we are free to accept an new request */
```

When a FORKSERV receives a local message from a task requesting exclusive forking rights on some remote node, it must first grant the task exclusive forking rights on the local node, and then the local FORKSERV must obtain for the requesting local task exclusive forking rights from the remote FORKSERV. Remote rights are needed since that is where the fork will actually take place and the local rights are needed so that fork will not be initiated until the fork rights are obtained at the remote node. The following RPC is used by a local FORKSERV to obtain forking rights from the remote FORKSERV on the target node (algorithm U):

```
U: remote_get_lock_to_rfork(remote_forkserv_main_port,
                            target_node_where_forking_is_desired)
U1: forker_node = the name of the node where the task wants to fork.
    /* just by virtue of dequeuing the request (and thus invoking this
    servprocs), the task wanting exclusive forking rights on this
    node has now "got" them */
    /* forker_node is in this server's global space */
U2: forker_node = the name of the node where the task wants to fork.
U3: tell main server of forkserv servicing main queue NOT to dequeue
    anymore requests from the main queue after this servproc returns,
    since at this point, the task that made this request now has
    exclusive forking rights on this node.
U4: send back indication (ACK) to requester forkserv that it now has
    acquired exclusive remote forking rights for the task the
    requesting local forkserv is currently trying to get them for.
```

During the *recording* phase, a **system()** call is used as the forking primitive (the csh **rsh** command is used within **system()** if the fork is remote). When the application program is transformed for the *recording* phase, all **system()** primitives are transformed into a function call **rec_system()** which is linked after transformation and compilation of the application program. The **rec_system()** routine does the following (algorithm T):

```
T: rec_system(string)
T1: determine virtual name of thread self.
T2: IF (not a remote fork) /* not a "system(rsh...)" type call */
T3:     targethostname = the node_name mentioned in "string".
T4: ELSE /* a remote fork */
T5:     IF (myhostname() == targethostname)
T6:         error_code(convention states that rsh only spawns on remote node).
        ENDIF
T7: local_get_lock_to_rfork(local_forkserv_port,
                            pid_of_myself,
                            my_virtual_thread_id,
                            destination_node).
    /* ask for forking rights on this node, and when obtained,
    indirectly ask for forking rights on the remote node, and
    when obtained, return. */
    ENDIF
T8: system(string). /* real call */
T9: IF (a remote fork)
        /* surrender exclusive forking rights on the local node by
        sending a message to the "out-of-band" port (note the
        forking rights on this remote node will be surrendered by
        the remotely forked task (see Figure 5.7 ) */
T10:    remote_unlock_fork_lock(oobforkserv_port, my_pid, my_virtual_thread_id).
    ENDIF
```

When a new task is spawned, it immediately executes `init_cdb` (see 5.2), which immediately calls `local_unlock_fork_lock()`.

### 5.2.4 Repforkserv

The mission of this server is similar to that of the FORKSERV (see 5.2.3), except it is designed to run during the *replay* phase instead of the *recording* phase. The main difference between the FORKSERV from the REPFORKSERV is that the latter enforces the forking sequence on each node so that it corresponds to the **forking log** that was generated during the *recording* phase. This must be done so as to give the same "names" to the instances of execution so that they can be matched-up to the correct execution history (log) that was generated during the *recording* phase.

In short, the REPFORKSERV must:

- ensure that no forks occur concurrently so that the report to the REPFORKSERV concerning a forking operation (via a message) is unambiguous. This is to ensure that no race conditions due to another forked child task sending an RPC (algorithm X) to FORKSERV's OOB port, to have the new virtual id mapped to its O.S.-given I.D., can occur (see next item).

- ensure that only *one* newly forked task can send a message to the REPFORKSERV asking for a **virtual task id** at any time (avoid race conditions). Recall that when a task is spawned, it asks for an ID itself, via the init_cdb() debugging primitive, which is supposed to be on the first line of the source code, and ID's are given out as tasks are created.

- if a request for exclusive-forking-rights is received from a task/thread (or remote node) and according to the execution history, it is not that task/thread's (or remote node's) turn to fork, the request is deferred. The task/thread (or node) asking for exclusive forking rights doesn't receive the acknowledge message informing it it may proceed with the fork until the REPFORKSERV sends the acknowledgment, thus the client (the entity requesting exclusive forking rights) is blocked. The REPFORKSERV is a multi-threaded server that forks (and detaches) a new thread for each request (message) that is dequeued. Thus if a forking request is out of sequence relative to the execution log, the request thread is put to sleep. When a task has been given exclusive forking rights and the task has completed its fork, notice is given to the REPFORKSERV to advance the execution history and all sleeping request threads are awakened so that each can check whether it is their turn to spawn a new task. Otherwise, the request threads put themselves back to sleep.

The main service loop and the request thread of the REPFORKSERV works as follows:

```
S: main_service_loop()
S1: FOR (;;)
S2:       allocate memory for next incoming request.
S3:       receive request message m.
S4:       detach_thread ( fork_thread (RequestThread, pointer to message-m ) ).
    ENDFOR
```

The algorithm for REPFORKSERV's request thread (algorithm R) is as follows:

```
R: RequestThread()
R1: demultiplex and call the appropriate server routine and wait for the reply.
R2: send back the reply in a message.
R3: deallocate memory used for request message.
R4: exit(). /* kill this thread */
```

The pseudo-code for the REPFORKSERV's service procedures (don't forget that the following routines execute as request threads, and that they are reentrant routines) are as follows:

The routine local_get_lock_to_rfork in REPFORKSERV is different from the one in FORKSERV in that when it dequeues a request for exclusive forking rights, it checks whether the task/thread asking for forking permission is in the same sequence as during the *recording* phase, and if not, it defers granting the requester exclusive forking rights (algorithm Q).

```
Q: local_get_lock_to_rfork(main_server_port_of_local_repforkserv,
                           forker_real_pid,
                           forker_v_thread_id,
                           destination_node)
Q1: look-up forker's virtual task id in repforkserv's mapping data-structure.
    /* is the thread-task that wants to fork in the correct sequence? */
Q2: WHILE (next_forker_virtual_id != v_pid_v_tid)
Q3:         condition_wait(someone has finished forking and has given
            up his exclusive rights).
    ENDWHILE
Q4: IF (remote-fork)/* if this is a remote fork, get exclusive
                  forking rights from the remote repforkserv
                  before sending the forker an ACK that it has
                  exclusive forking rights */
Q5:               remote_get_lock_to_rfork(server_port_on_destination_node,
                                           my_local_node_name).
    ENDIF
Q6: send ACK to caller that it now has exclusive forking rights on the
    remote node.
```

Again, the **remote_get_lock_to_rfork** system call is analogous to the one in FORKSERV, except that here it checks whether the REPFORKSERV making the request for forking rights is doing it in the correct sequence–if not, the request is deferred (algorithm P):

```
P: remote_get_lock_to_rfork(main_server_port_of_remote_repforkserv, forker's_node)
P1: WHILE (it's not yet time to give exclusive forking rights to a
              remote task which is requesting from "forker's_node")
P2:           condition_wait(someone has finished forking and has given
              up his exclusive rights).
    ENDWHILE
P3: send ACK to caller (typically a REPFORKSERV) that it now has
    exclusive forking rights on the remote node.
```

When a new task is spawned, the first thing it does is to execute `local_unlock_fork_lock()`, which lets the local REPFORKSERV know that forking has been completed and that it is surrendering the exclusive forking rights it presently possesses. `local_unlock_fork_lock()` gets the next log element from the forking log and signals to any sleeping fork requests that they can wake up and check if it is their turn to fork (algorithm O).

```
O: local_unlock_fork_lock(main_server_port_of_local_repforkserv,
                          real_pid,
                          virtual_thread_pid)
O1: acquire a new virtual_task_id for the new task and add a
    pid-to-virtual_task_id mapping in the repforkserv's space.
```

O2: open associated physical log (execution history) for the new task
    that was recorded during the replay phase.
O3: advance the forking log for this repforkserv (representing this node).
O4: signal to all sleeping request threads that someone has
    released his exclusive rights (on this node) and the log has been
    advanced (so please wake up and check if it's you).

The **remote_unlock_rfork_lock** call is essentially the same as in FORKSERV (algorithm M).

```
M: remote_unlock_rfork_lock(main_server_port_of_remote_repforkserv,
                            forker_real_pid,
                            forker_virtual_thread_id)
M1: advance the forking log for this repforkserv (representing this node).
M2: signal to all sleeping request threads (on this node) that
    someone has released his exclusive rights and the log has been
    advanced (so please wake up and check if it's you).
```

### 5.2.5 Gpnameserv and Repgpnameserv

The GPNAMESERV provides a mapping service between names that are currently "checked-in" at the local netname server (relevant to the distributed program the GPNAMESERV is currently serving) and the debugger-given *virtual name* of the port it represents. This service is essential when a task looks-up a name in a node's local netname server (*e.g.*, the **netname** server [18] in the Mach [2] operating system). The GPNAMESERV is also used to keep track of which node a port's *receive right* is currently on. A lookup for the location of a receive-right is necessary in order to properly set-up transparent message interception by the CONTROLSERV at that location in order to decentralize the interception of messages. The REPGPNAMESERV is identical to the GPNAMESERV. This server also maintains a list of ports[3] whose

[3]that do not have anything to do with the netname server

receive rights on that node currently **cannot** be transferred. Clients (typically the CONTROLSERV) can add, subtract, and check the list using the port's **virtual** name. The CONTROLSERV checks whether a port *p*'s receive right can be moved before doing so in order to avoid conflicts with any intercept set-up operation in progress involving the same port *p* (see section 5.3.3).

The GPNAMESERV is a simple **single-threaded** database. It can add, delete, modify, and lookup information using various search keys. The query language is a set of MIG [9] RPC's, since the only client of this server is the debugger itself, thus the type of services required are few and quite specific. There are two types of tuples that are maintained by the GPNAMESERV:

| | |
|---|---|
| virtual_name_of_port | netname |

Table 5.5: Netname to virtual port-id mapping tuple

| | |
|---|---|
| virtual_name_of_port | unmoveable |

Table 5.6: Location of receive rights tuple

The tuple described in table 5.6 includes one **status** variable, **unmoveable**, which is set and checked by the CONTROLSERV (and REPCONTROLSERV if in *replay* mode). This status variable assists the CONTROLSERV in avoiding race conditions (and possible deadlock) if a port *p*'s receive right is in the process of transferring to another task while port *p* is in the process of having interception set-up on it by the local CONTROLSERV (more details in section 5.3.3).

### 5.2.6 Controlserv

The CONTROLSERV can be considered to be the most important server since it is responsible for transparently intercepting various types of messages and logging their enqueuing order. The CONTROLSERV is an active client of the LOGSERV and GPNAMESERV. There is one CONTROLSERV per node.

The transparent interception of messages is accomplished by having the CONTROLSERV extract an application task's "send-right" capability to a port immediately following the application task's acquisition of the capability, thus in effect the controller steals the capability for itself. The name $n$ the application knows the capability (port right) by remains unchanged. Now the CONTROLSERV has send rights needed to forward messages to the target port. The CONTROLSERV then allocates an **interception port** and inserts its "send-right" capability for the **interception port** it just created in the application task under the **capability name** $n$. Thus, message interception is *transparent* to the person debugging one's program, since the send-right known as $n$ did not change its name and the messages are appearing to be getting through to the destination port. The primitives used on the Mach O.S. [2] to implement the above described transparent message interception were the **`port_insert()`** and **`port_extract()`** series of primitives.

The general duties of the CONTROLSERV involve:

- intercepting "ordinary" and "notify" (see Appendix A) messages heading toward "application" ports and "notify" ports respectively, log their order of arrival at the intercept port, and forward the message to the target destination port.

- process *recording* phase control messages that are associated with and follow each application messages (see later for more details), one control message per application message.

- acting as a MIG RPC server for setting up interception on request and for setting up a mechanism to detect an application port's death so that the event can be recorded in the database if necessary and execution history files can be properly closed.

- provide the location (node name) of a receive right on demand. Assuming that interception is already set up for the port in question, a task/thread having a send right to some port $p$ will send a specially marked message to port $p$, knowing that it will be intercepted by the CONTROLSERV on the node where port $p$'s receive rights are located. The CONTROLSERV will not forward the message to port $p$. Rather, the CONTROLSERV will return the location of $p$'s receive rights (the node where the specially marked message was intercepted). The location request is serviced by a port's "Intercept Thread" on the CONTROLSERV.

There is one CONTROLSERV server per node, per program. The various service ports of this server contain the program's ID so that multiple debugger sessions (different programs, possibly run by different users) can co-exist. Interception is set-up to occur at the *node where the task that holds the receive-rights of the target port resides.* If the receive-rights to a port migrates to another task *on the same node*, then no adjustment of the interception scheme need be done. If the receive-rights to a port migrates to another task *on a different node*, then an adjustment of the interception scheme must be performed so that interception continues to occur at the node where the task that holds the receive-rights of the target port resides.

Interception is always set-up to occur at the *node where the task that holds the receive-rights of the target port resides* in order to facilitate moving the interception set-up if and when migration of a receive-right occurs (see figure 5.8 and section 5.3.3 for more details). The interception scheme must be adjusted dynamically if receive rights move to another node to ensure that the probe effect remains balanced during the *recording* phase. This way, no task holding send-rights to the port whose receive rights is moving needs be readjusted. Furthermore, having interception occur at the location (node) of a port's receive right allows a port's **enqueuing log** to be localized on the same node as where the receive rights for that port are. A port's enqueuing log will be spread across various nodes if a port's receive right migrates between tasks situated at **different** nodes, keeping at the local nodes only the portion of the execution history that actually took place there. It is assumed that receive-rights are not moved frequently, if at all, in a typical program. For example, fault-tolerant distributed applications, which would be the chief users of migrating receive right capabilities, typically do not constantly "fail", so the probe effect of moving the interception scheme should be, on average, small, since it is an event that doesn't occur frequently.

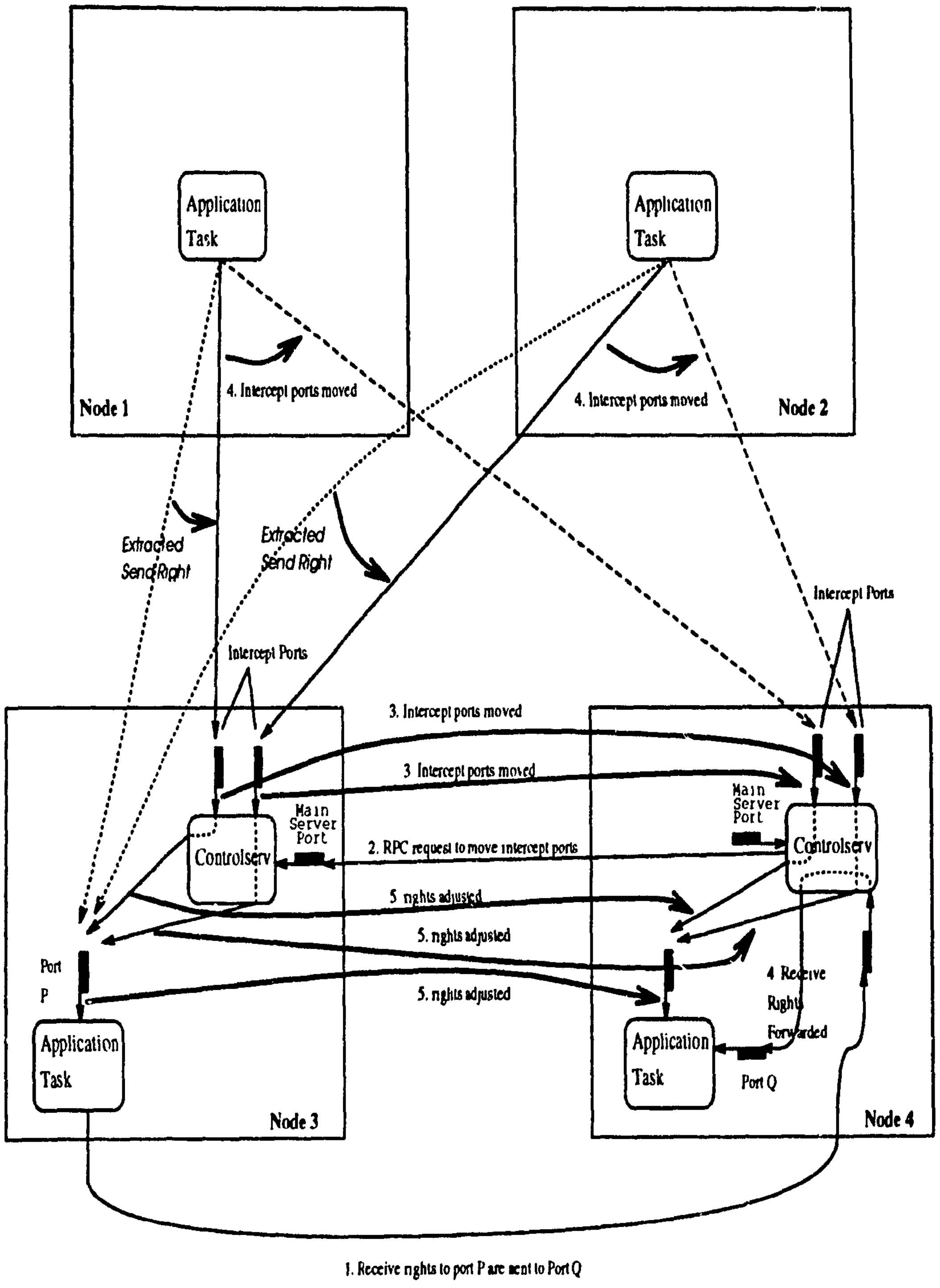

Figure 5.8: Migration of receive-rights

Transparent message interception is set-up whenever a task acquires a send right from the *netname* server or from an application message carrying a send-right directly. The request for interception (RPC) is then directed to the CONTROLSERV *on the node where the task that holds receive rights to that port resides.*

Often, message-based operating systems provide each task with a special port in which to receive special asynchronous "notify" messages from the kernel concerning various events that have happened outside the recipient task that it may need to be informed of. Since "notify" messages are produced in response to events external to the task that receives them, the notify message's order of reception can vary from one execution to another. Thus, all "notify" messages must be intercepted for each task, have their enqueuing order logged, and then have the message forwarded to the appropriate task's port. Interception of task's notify port is arranged immediately after a task is created. The non-migration of tasks to other nodes is assumed by the CONTROLSERV (and the debugger in general). The interception is transparent to the user of the debugger, since the debugger sets up interception by extracting and inserting port-rights of an application's notify port. This is done in a fashion similar to the interception of ordinary application messages, except that the CONTROLSERV extracts the receive rights of the notify port from the application task, allocates a substitute notify port it forwards the notify messages to and gives the application task receive rights to the substitute notify port under the original "name" the application task knew the notify port as, thus achieving transparent interception.

It is assumed that the messages a given thread sends to a given port are guaranteed to arrive (enqueue) in the order sent[4]. It is also assumed that each message can only hold a limited amount of data and that the message (at the level of the O.S. interface) does not contain a field indicating from which task/thread/node the message came. Since the *recording* phase needs to know which task/thread/node sent each message, as well as other miscellaneous information (see later), a **control** message follows each application message with the latter information. The alternative of appending additional "control" information within an application's message may interfere with

[4]the Mach O.S. guarantees this

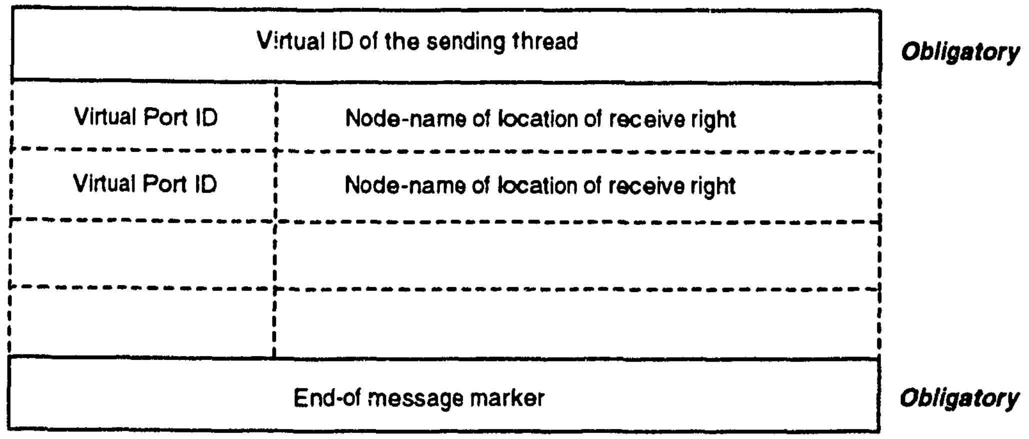
Virtual ID of the sending thread
Obligatory
Virtual Port ID
Node-name of location of receive right
Virtual Port ID
Node-name of location of receive right
End-of message marker
Obligatory

Figure 5.9: Format of the rec/rep control message

the normal functioning of the application program (as well as making the debugger more visible during the *replay* phase) and may introduce a error in the application program itself if there is not enough space in an application message for the debugging "control" information. Figure 5.9 shows the format of the control message.

The CONTROLSERV is a multi-threaded task in which various threads perform distinct specific sets of functions while sharing access, to varying degrees, to certain server global data-structures. The threads consist of:

**Intercept Service Thread** Assuming that receive rights to a given port can only be held by one task, an intercept service thread is allocated to intercept all messages being sent to a specific application port, and the dedicated intercept service thread for the port is allocated on the node where the receive right exists for the given port. There can only be one *intercept service thread* per port. If a receive right migrates to another node, the *intercept service thread* must be killed at the old node's CONTROLSERV and resurrected at the new node's CONTROLSERV.

Each existing send-right to the existing port by a local or remote **task** (not thread) is represented on the CONTROLSERV by an **intercept port**, and all such intercept ports are serviced by port-specific "intercept" service threads, which receive from all the intercept ports that represent send rights of various tasks *for a specific port.*

A separate intercept port is required for *each task with send rights to a specific port* in order to avoid having control and application messages from different tasks that have send rights to a common port from getting interleaved, making it impossible to match application messages (with no field as to who sent them) with their respective control message. A task can ensure that no interleaving between the real and control messages sent by two or more of its threads to the same port can occur. This can be done by having the sending task making sure that only one thread within its address space can send a real and a control message to a given port. In other words, the act of a thread sending a real message and its associated control message to a given port is a mutually exclusive event within a given task. This is accomplished transparently by transforming the O.S.'s message sending primitive to a routine that will ensure such mutual exclusion before actually sending the real and control message. There is no such practical method to ensure that interleaving of various real-control message groups doesn't occur if 2 different tasks (on same or different nodes) have send rights to the same port. Thus, each task that has send rights to a given port $p$ must have its own intercept port on the CONTROLSERV that is on the node where port $p$'s receive rights reside.

The **control message** is generated within the routine that is called after the O.S.'s message-sending primitive is transformed (for the *recording* phase). The control message contains (in sequence):

- the **virtual** thread ID of the sending thread,
- if the message is carrying port right(s) within the message, the **virtual** name of the port $p$ whose rights are being passed as well as the location (node-name) of the task holding receive-rights to port $p$. If more than one port right is passed in a message, the control message will contain a sequence of virtual port names and its receive-right location in the same sequence as the port-rights occur in the message, enabling the receivers

of the application and control messages to map the descriptions of the port-rights in the control message to the actual port rights being passed.

- a flag to indicate the end of the control message, since its length depends on how many port rights are passed in the message.

**Notify Service Thread** This thread intercepts all notify messages that are sent by the kernel to the application tasks on the local node, logs the enqueuing order, and forwards the message to the intended task. There is only one notify service thread per CONTROLSERV, intercepting all notify messages that come in from the local kernel to the local application tasks, logging the message, and then forwarding the message appropriately. The data logged for the intercepted notify message consists of the "type-of-notify-message" involving "virtual-port".

**Control Notify Service Thread** After interception is set up (*i.e.*, the CONTROLLER extracts send rights from the real sender), some **notify** messages that concern the extracted send-rights of application tasks would now be sent to the CONTROLSERV *instead* of the intended application task. This is because the CONTROLSERV has send rights to the real port, and the sending task has send rights to an **intercept** port on the CONTROLSERV after interception is set-up. Thus, the CONTROLSERV's own notify port must be monitored by a thread, which determines if notify messages it receives from the kernel were really intended for an application task, and if yes, the CONTROLSERV generates and forwards the notify message to the intended notify port. On the Mach platform the prototype debugger was designed for, forwarding a **notify** message to a task $t$'s notify port when one of the ports that $t$ has send rights to dies is accomplished by simply deallocating the **intercept** port for the dead port.

**Main Service Thread** The CONTROLSERV also acts as a MIG RPC server for interception set-up requests from **transformed** application programs. A dedicated thread dequeues such requests from a dedicated port (where RPC requests are directed), calls the appropriate server routine, and sends the server reply message (if applicable).

The general scheme of CONTROLSERV is illustrated in figure 5.10. In this figure, the manner in which the CONTROLSERV intercepts messages destined for "normal" and "notify" ports are illustrated, as well as the fact that the CONTROLSERV has send rights to a "secret" port to detect a task's death. The CONTROLSERV receives death notifications from the kernel via its own "notify" port. Currently, the CONTROLSERV in figure 5.10 is intercepting two application ports.

### 5.2.7 Repcontrolserv

The REPCONTROLSERV generally serves the same purpose during the *replay* phase as does the CONTROLSERV during the *recording* phase, except that instead of logging an execution history, the REPCONTROLSERV ensures that all non-deterministic choices an application program made that a particular node-specific CONTROLSERV recorded during the *recording* phase make the same choices during the *replay* phase.

Transparent interception is also set-up in the the same manner as for the CONTROLSERV (see 5.2.6).

The general duties of the REPCONTROLSERV involves:

- intercepting ordinary and notify messages (and their associated control messages) heading toward application ports, checking with the execution history (for the enqueuing log of that port) whether the message it received should be forwarded to the destination port or should it be stored for forwarding at a later, more appropriate time in order to enqueue a given port in the same order as recorded in the execution history log,

- immediately forwarding any messages it has temporarily stored (and their associated control messages) to their intended ports when it accords with the execution history,

- processing *replay* phase control message that immediately follow all application messages,

- acting as a MIG RPC server for setting up interception and for registering an application task on the server.

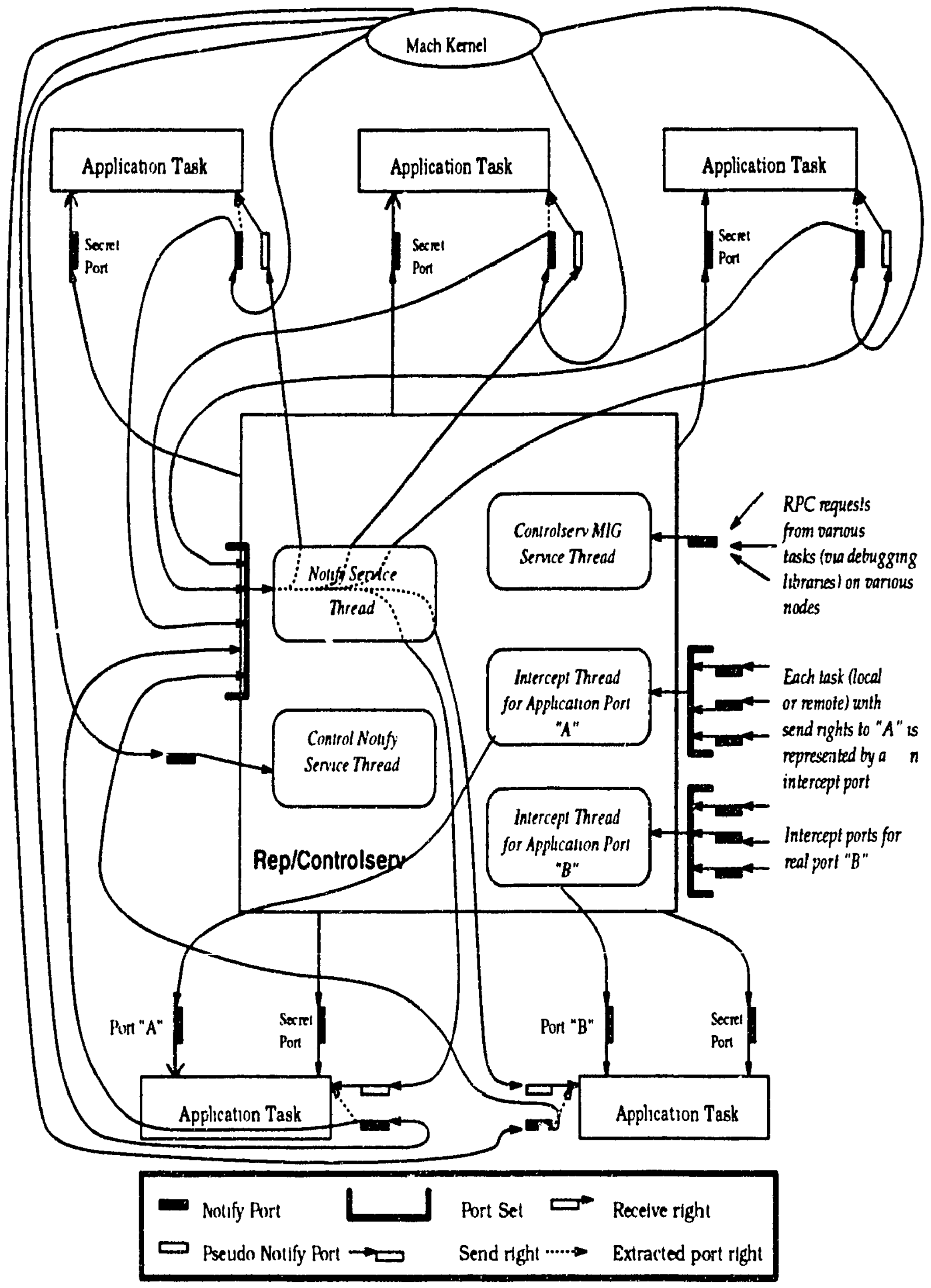
Mach Kernel
Application Task
Application Task
Application Task
Secret Port
Secret Port
Secret Port
Notify Service Thread
Controlserv MIG Service Thread
RPC requests from various tasks (via debugging libraries) on various nodes
Intercept Thread for Application Port "A"
Each task (local or remote) with send rights to "A" is represented by a n intercept port
Control Notify Service Thread
Intercept Thread for Application Port "B"
Intercept ports for real port "B"
Rep/Controlserv
Port "A"
Secret Port
Port "B"
Secret Port
Application Task
Application Task
Notify Port
Port Set
Receive right
Pseudo Notify Port
Send right
Extracted port right

Figure 5.10: General Scheme of CONTROLSERV

- as in the CONTROLSERV, on demand, return the location of a port's receive rights to the requester.

The same assumptions about set-up and configuration that were stated in section 5.2.6 for the CONTROLSERV apply to the REPCONTROLSERV. The same service threads exist in the REPCONTROLSERV as in the CONTROLSERV, except for two important differences—the threads that intercept application and "notify" messages, called "intercept service thread" and "notify service thread" spawn new **request** threads for each message intercepted. This is done so that if the **request** thread handling the intercepted message realizes the message cannot be forwarded to the target destination (lest it violate the execution history logged during the *recording* phase), the **request** thread puts itself to sleep, awaiting a wake-up call. On waking, the **request** thread must again verify with respect to the execution history if it can now forward the message—if yes, the **request** thread forwards the message, along with the message's associated control messages, and then the **request** thread exits..., if no, then the **request** thread puts itself to sleep again.

## 5.3 Non-Deterministic Events: Monitoring and Reproduction

Monitoring and logging non-deterministic choices involves cooperation between special *monitoring* phase libraries (called by the application program being debugged after their calls are inserted into the code during the program's transformation) and the various debugger servers on each node of the distributed system. The type of debugger libraries called (and which portion of them is used), and which debugger server nodes are used depends on the type of non-deterministic activity the application program intends to execute. Execution history logs are always recorded on the node where the activity took place by either the [REP]CONTROLSERV or the application task (via a debugging library). In short, most of a task's non-deterministic activities are recorded in a distinct *physical* file (consisting of many logical sequential logs) associated with that task. All port **enqueuing** histories of a program are kept in the same physical files, one enqueuing physical file per node, and the enqueuing history logs are maintained by the [REP]CONTROLSERV on each node. The logs of the *enqueuing sequence* for all application-allocated ports and all notify ports that were allocated by the kernel for the application tasks, as well as the **forking sequence** for a node, are kept in the same physical file, one per node. Otherwise, the amount of open **physical** files would be very high and it could exceed an operating system's limit for open files.

### 5.3.1 Message Sending

#### Recording Phase

During transformation of an application program specifically for the recording phase, all instances of the message sending primitive (`msg_send()`) are replaced with the debugger library call `rec_msg_send()`, which calls the real message passing primitive, among various other *recording* phase duties it performs (see algorithm A on page 92). When the task is about to send a message to some port *p*, if the task indeed has send rights to port *p*, message interception will have already been set-up to be directed to the CONTROLSERV on the node where the receive rights for that port reside. Thus the `rec_msg_send()` library call and the service thread(s) that dequeues from the "intercept" (substitute) ports on the CONTROLSERV are involved in monitoring message sending activity. The CONTROLSERV is concerned with the *order in which messages are enqueued at given ports by various thread/tasks which have send rights to them*, thus recording the interleaving of messages from various sources (see table 5.7 for log element recorded when application tasks send to application ports). It is assumed that a port receiving messages from a given task/thread combination will always receive the messages in the order sent.

Application task notify-ports receive their messages from the kernel only. Since the kernel sends notification messages to the notify port in response to external events (initiated by certain specific activities initiated by application tasks, like the death of a port), the same sequence of notify messages cannot be guaranteed to be re-generated on each re-execution. Thus, all messages that are sent by the kernel to the

| | |
|---|---|
| Who adds to log | The controlserv on receipt of intercepted message, if forwarding was successful |
| Kind of data | The name of the virtual task-thread that sent the message |
| Who created log | Task that got send rights to port P sends a "set-up interception" request to the controller on the node where receive rights to port P are found. The logical log is created when the first request for an intercept-setup for a given application port comes in. |
| Location of log | In the physical log file on a controlserv which is dedicated to enqueuing logs. The enqueuing log is kept on the controlserv on the node where receive rights are located when the message was enqueued at the intercept port. If the receive rights to port P move to another node, then the enqueuing log to port P will be scattered across several controlservs. If the receive right moves back to a node it once occupied, the existing enqueuing log for port P is then appended to it. |

Table 5.7: History of message enqueuing at an application port

notify port are intercepted during the *recording* phase (see table 5.8 for log element logged when kernel sends to notify ports).

| | |
|---|---|
| Who adds to log | The controlserv on receipt of intercepted message, if forwarding was successful |
| Kind of data | The type of notify message and the port involved. |
| Who created log | Application task when task is created (in init_cdb()) It then requests interception set-up from the local controlserv of the task's notify port. |
| Location of log | In the physical log file (for enqueuing logs) on the local node (assuming notify port's receive right never moves) which is dedicated to enqueuing logs. |

Table 5.8: Log of application notify port enqueuing

As well, a task having send rights to some port $p$ may pass copies of the send right to any task $t$ it has send rights to. Before doing so, the task (via a debugging library) sends a control message to port $p$ requesting the location (node) where the receive-rights are presently located. The message directed toward port $p$ should be intercepted by the CONTROLSERV at the node where the receive rights reside. The CONTROLSERV recognizes requests for the location of specific receive rights, so the request message is not forwarded to the real port-instead the CONTROLSERV returns the node-name where the receive-rights to port $p$ are located. The location of the receive-rights are important to the recipient of the send rights in order for it to request message interception to be set-up for it on the CONTROLSERV on the correct node.

Algorithm A shows the relevant parts of **rec_msg_send()** in order to record asynchronous message passing (without considering time-outs):

```
A0: mutex_lock(super_lock[sending_port]).
A1: OS_return_code = msg_send(). /* send the real message */
A2: IF OS_return_code is not successful.
        return OS_return_code to application program.
    /* compose control message */
A3: determine sending thread's virtual ID from its real (OS-given) ID.
```

```
A4: put the sending thread's virtual ID in the control message.
A5: FOR each port right passed in the message /* from first to last */
A6:     determine sending thread's virtual ID from its real (OS-given) ID.
A7:     put the port's virtual ID in the control message.
A8:     IF send rights (to some port p) are being sent
A9:         send a request message to port p asking
            on which node the task that has receive right to it is, and wait for the reply.
            /* message should be intercepted by CONTROLSERV
            on the node where the receive rights reside, and
            the CONTROLSERV sends back the node's name */
A10:        put the port's receive right location (node) in the control message.
        ENDIF
    ENDFOR
A11:put end-of-message marker in control message.
A12:send the control-message to the same port as the application message.
A13:mutex_unlock(super_lock[sending_port]).
A14:return OS_return_code (see A1) to the application program
    which called msg_send().
```

On the CONTROLSERV, each task that has send-rights to some application port $p$ that a CONTROLSERV is intercepting is represented by an **intercept** port, which intercepts all messages sent to port $p$ from a specific task. For each real port $p$ that a CONTROLSERV intercepts, there exists a dedicated *intercept thread* that dequeues from all such *intercept* ports for the application port $p$. The **intercept** port expects to receive the real application message, followed by a *recording* phase **control** message which describes which task/thread sent the message, as well as the virtual names of any port rights being passed in the message (and possibly the location of the receive rights of the port if a send right is being passed). Algorithm B shows the *intercept* thread's duties. Note that in line B4 that the CONTROLSERV is marking a receive-right as "unmoveable". The only time a request is made for the location of port-receive rights (B3) is when a thread in a task is about to pass send rights in a message it already has ***and for which interception is already set up*** but it doesn't currently hold receive rights to (see section 5.3.1).

```
B1: WHILE (TRUE) /* infinite loop */
B2:         dequeue a message from any non-empty intercept ports
            representing various task's send rights to an application port p.
B3:         IF (message is a request for location of port-receive rights)
B4:            send back the name of this node to sender.
B5:            add the virtual name of the application port this
               "intercept" thread is intercepting to the list of
               "unmoveable" receive rights in the local GPNAMESERV.
B6:            go to top of loop (B1).
            ENDIF
B7:         IF a real application message was dequeued
B8:            go to top of loop (B1) and wait for another intercepted message.
            ENDIF
            /* control message arrived, which describes the previously
            dequeued application message on this intercept port */
B9:         forward the real and control messages. /* real msg first */
B10:        IF (the real and/or control messages were NOT successfully forwarded)
B11:           intended port died, so application's msg send considered unsuccessful.
B12:           go to top of loop (B1).
            ENDIF
            /* log successful msg_send() in enqueuing log */
B13:        log virtual ID of sending task in enqueuing log for
            intended port on local CONTROLSERV.
B14:        log virtual ID of sending thread in enqueuing log for
            intended port on local CONTROLSERV.
    ENDWHILE
```

For **synchronous** message sending (a primitive that sends a message $m$ to a port and will block until it dequeues a return message from the port it gave send-rights to in message $m$), the transformation of the application program will call a routine **rec_msg_rpc()**, which will split the synchronous primitive into a **sending** primitive (see algorithms A and B) and a message receiving primitive (see algorithm C). No other special treatment is needed to record synchronous message-passing events, since a synchronous message-sending primitive really is a combination of a msg_send() and msg_receive() primitive.

**Time-outs** on message sending primitives are also monitored. A send time-out is defined to have occurred if the message was unable to enqueue at the target port within a pre-set time limit. A logical log for each application thread in an application task is maintained by the debugger to record a sequential thread's history. The message sending primitive (asynchronous or synchronous) is transformed into a call to a debugger library routine, which checks during the *recording* phase whether the message send primitive involves a **send time-out** or not. If it does involve a **send timeout**, the debugger routine logs whether the send operation timed-out or not (see table 5.9). Since messages are transparently intercepted at the CONTROLSERV, a time-out occurs if the message sent by the application task cannot be enqueued at the **intercept** port before the pre-set time limit and not the actual target port. This is generally not a problem, since if the target port is full, then the pseudo intercept port(s) will eventually fill up, causing the original sender to possibly time-out when the intercept ports become full.

### Replay Phase

All instances of msg_send() primitives are replaced with a call to a special debugger library routine (rep_msg_send()) for dealing with this primitive during the *replay* phase. The algorithm for the latter routine is identical to the algorithm describing rec_msg_send() (see algorithm A) except the manner in which send timeout's are handled and the fact that each msg_send() operation results in a control message being sent immediately following the real application message:

| | |
|---|---|
| Who adds to log | task-thread (via debugger library routine) that does a send or receive that involves a specified time-out bound. |
| Kind of data | The result of the time-out (timed-out or not). |
| Who created log | Application task-thread when thread is created (by forker thread, or init_cdb() if initial thread). |
| Location of log | Physical log file dedicated to a particular application task. |

Table 5.9: *History of time-out results for an application port*

- the **replay control message**, which carries the same information as during the *recording* phase, as well as information concerned with the mechanics of checkpointing, rollback and recovery and a global time-stamp for use by the database tool.

On the REPCONTROLSERV, just as on the CONTROLSERV, application messages are intercepted transparently, with the difference being that the REPCONTROLSERV intercepts a control message following receipt of the application message that carries more information than the recording control message during the *recording* phase. As in the *recording* phase, the control message is sent by the sending thread (via the rep_msg_send() library routine) immediately after the application message to the same destination port $p$ and guarantee that no other thread in the *same* sending task will send any message to port $p$ until both messages (real and control messages) have arrived at their destination, which is an intercept port on the REPCONTROLSERV. In other words, the sending of the real and the control message must be considered to be *atomic*.

The main difference between the REPCONTROLSERV and the CONTROLSERV is that the latter logs the sender of the message (the enqueuing order) and simply forwards the message to the real destination port, while the former must reorder the application messages (considering the application message and its two associated control messages as one logical message) received according to the enqueuing history, and then forward them to the intended application port. As in the CONTROLSERV, each send right a task has to a port a particular REPCONTROLSERV is intercepting is represented by an intercept port on the REPCONTROLSERV. A separate **intercept thread** for each application port $p$ the REPCONTROLSERV is intercepting dequeues from the group of intercept ports that represent send rights various tasks have to port $p$.

Message reordering is accomplished by using a multi-threaded server approach. Each series of intercepted messages (application and control) are considered a "request" to a multi-threaded server, where the application port-specific thread dequeuing messages from the associated set of **intercept ports** can be considered the "multi-threaded" server. When the group of three messages are received by the server, it spawns an independent "request thread" which serves to forward the three messages *when it accords with the history log* and then die, having served its purpose. Otherwise, the "request thread" puts itself to sleep until the history log is advanced, where it again checks whether it is its turn to forward its group of messages. Algorithms AA and BB show how this is accomplished:

```
AA: InterceptRequestThread(pointer to memory m)
    /* in case the same thread sends to the same intercept port and
    there is a back-log of InterceptRequestThreads from one port,
    then the FIFO ordering of message sending between a certain
    thread and port will be lost, so each thread carries a local
    variable "priority" which maintains the FIFO order within the
    pending InterceptRequestThreads threads */
AA1: IF there are other pending InterceptRequestThreads representing
     messages sent by a thread of a particular task
        my_priority = number of pending InterceptRequestThreads.
AA2: ELSE
        my_priority = 0.
     ENDIF
AA3: WHILE ((task that sent message !=
        current task in enqueuing history log for real port) OR
        (thread that sent message !=
        current thread in enqueuing history log for real port) OR
                (my priority != 0)) /* remain in "waiting" loop */
AA4:    IF (((task that sent message ==
        current task in enqueuing history log for real port) AND
        (thread that sent message ==
        current thread in enqueuing history log for real port)) AND
        (my_priority != 0))
AA5:            my_priority = my_priority - 1.
        ENDIF
AA6:    go to sleep, to be awakened only when the enqueuing
        history for the port in question has been advanced.
    ENDWHILE
AA7: at this point, forwarding rights have been acquired, so forward the
    real and replay control messages to their ultimate destinations.
AA8: advance the enqueuing history log for the real (application) port in question.
AA9: signal (wake-up) all pending InterceptRequestThread's waiting for
    permission to send to the port in question.


BB: MainInterceptServiceThread(intercept_port_set)
   /* intercept_port_set represents an application port */
BB1: receive (dequeue) intercepted message from an intercept port.
BB2: determine from which intercept port it just received a message from,
    thus determining which virtual task from which a thread just
    sent this message from.
BB3: IF all messages of group came
BB4:    allocate memory m for 2 messages.
BB5:    copy the two received messages (one from buffer b and one from
        the message receive buffer) into the allocated memory m.
BB6:    fork and detach an "InterceptRequestThread" to forward the
```

```
        "group of two messages in memory m to their intended
        destination when the execution history says it can.
BB7: ELSE
BB8:    store received message in a buffer b that is associated to the
        intercept port just received from.
     ENDIF
```

**Timeouts** are handled by the rep_msg_send() library routine before any message is actually sent. The library routine notes whether the send primitive has specified any timeout. If yes, the library routine checks the timeout log history for the thread wishing to perform a timeout with a send operation. If the history says that the send operation timed-out during the *recording* phase, then the library routine will not perform any send operation–instead it will simply simulate an O.S. return code signifying that the operation "timed-out". Otherwise, the library routine will guarantee that the send operation will not time-out by executing a real "msg_send()" operation without any timeout specified.

### 5.3.2 Message Receiving

#### Recording Phase

Just as for message sending, all instances of message receiving primitives are transformed into a function call (rec_msg_receive()). This call is linked to a debugging library routine which monitors all received messages (dequeued from a port), as well as actually receiving the message. Each task has within its address space (secretly maintained by the debugger) a list of data structures consisting of elements that map a port right that the task currently has rights to with that port's **virtual** name. For each application message that is received, the debugger routine (rec_msg_receive()) expects a **control** message describing the newly arrived application message to immediately follow the message. Both the application and control message are forwarded by the CONTROLSERV serving the local node for the distributed program in question.

On receipt of a message, rec_msg_receive() must log which thread received (dequeued) the message and set-up message interception if new send rights were received. See section 5.3.3 for the handling of transferal of receive rights. The algorithm for rec_msg_receive() is described in Algorithm C:

```
C1:  OS_return_code = msg_receive().
C2:  IF dequeued message from the notify port
C3:      log which **virtual** thread dequeued message
         into the notify port log for this virtual task.
C4:      return OS_return_code (see C1) to the application program
         which called msg_receive().
     ENDIF
C5:  dequeue control message.
C6:  FOR each port-right that was passed in the message just received
C7:      IF (port-right already exists in this task)
C8:          unmark the "unmoveable" flag for matching receive rights
             for the duplicate port rights this task has just
             received at the GPNAMESERV at the node where
             the receive rights are at (ie. send RPC to that
             GPNAMESERV).
C9:          go to C6.
         ENDIF
C10:     IF a send right $s$ was received
C11:         create a new port-to-virtual-port mapping in recipient
             task's address space and enter data.
C12:         set-up interception so that all messages sent to $s$
             will be intercepted. A request RPC is sent to the CONTROLSERV
             on the node where receive rights are located, as indicated
             in the control message for the port whose send right was received
             requesting that interception be set-up.
         ENDIF
```

```
    ENDFOR
C13:return OS_return_code (see C1) to the application program
    which called msg_send().
```

The CONTROLSERV's "intercept set-up" RPC routine does the following (see algorithm CY). Note that the capability to extract and insert port capabilities is greatly simplified if the operating system provides primitives to accomplish the operations (as Mach does). The name of the send-rights to the intercept port is given the same name as the original extracted capability in order to make interception transparent to the user of the program (see line CY3). For line CY4, recall that the "unmoveable" flag was set when a request for location of receive rights came in, a prelude to an intercept set-up request.

```
CY: setup_interception()
CY1: extract sending rights to port $p$ (with name $n$) from the task
     $t$ that just received send rights (the CONTROLSERV
     now has send rights, and the task $t$ currently has no
     send rights to associated with name $n$).
CY2: allocate an intercept port.
CY3: insert send rights to intercept port to task $t$
     under name $n$.
CY4: unmark the "unmoveable" flag on port $p$'s receive rights
     in the local GPNAMESERV.
```

A thread in a multi-threaded application program can dequeue from any port that the task has receive rights for. Thus, the possibility of a race condition exists (ie. non-determinism). Thus, dequeuing logs must be maintained for each port (see table 5.10).

There also exists a separate dequeuing log for notify ports (see table 5.11).

| | |
|---|---|
| Who adds to log | task-thread that dequeues from application port (via macro) |
| Kind of data | name of virtual thread that dequeued the application port |
| Who created log | The application task when it receives the receive right for the application port (port_allocate or receiving receive rights in a message) for the first time. If the task gives away the receive right and then later receives it again, then it will keep using the existing dequeuing log for that port. |
| Location of log | In the physical log file dedicated to a particular application task |

Table 5.10: History of message dequeuing at an application port

| | |
|---|---|
| Who adds to log | task-thread that dequeues from application port (via macro) |
| Kind of data | name of virtual thread that dequeued the application port |
| Who created log | The application task when task is created (init_cdb) |
| Location of log | In the physical log file dedicated to a particular application task |

Table 5.11: History of message dequeuing at a notify port

The result of **timeouts** on msg_receive() primitives are monitored and are logged in the same thread-specific log as the results of timeouts on msg_send() primitives (see table 5.9 in section 5.3.1).

## Replay Phase

The rep_msg_receive() macro is essentially the same as algorithm C, except that it receives extra information in the **control** message (between lines C12 and C13) like marker-messages, which is then used by the checkpoint and rollback-related code in the rep_msg_receive() routine as well as a global time-stamp, which is used by the database tool. Furthermore, the rep_msg_receive() macro differs from its *recording* phase counterpart in that rep_msg_receive() must ensure that the same threads receive from a given port in the same order as during the *recording* phase. The rep_msg_receive() macro achieves this with the following steps described in algorithm LL below (algorithm LL is a component of the rep_msg_receive() macro):

```
LL1: super_mutex_lock(p).
LL2:    WHILE (not my turn to receive from port p)
LL3:            condition_wait(super_mutex_unlock(p), wake_up_call).
LL4:    msg_receive(p).
LL5:    advance_log(p).
LL6: super_mutex_unlock(p).
```

Furthermore, **timeouts** are handled differently in rep_msg_receive() than in the rec_msg_receive() routine. If the latter routine notices that a msg_receive() has a time-out specified, it checks the calling thread's *timeout* log to determine whether the timeout during the *recording* phase occurred or not.

- If the msg_receive() had timed-out during the *recording* phase, a real msg_receive() is not executed. Instead, the rep_msg_receive() routine returns a simulated O.S. return code to the calling program, indicating that the msg_receive() operation had timed out.

- If it had not timed-out during the *recording* phase, a real msg_receive() is executed **without** any time-out specified, thus guaranteeing that the msg_receive() primitive will not timeout during *replay* phase.

### 5.3.3 Transfer of Receive Rights

The transfer of a port's receive right from one task to another, especially when the tasks reside on separate nodes, must be dealt with by the recording/replay controller (the transfer of send rights in a message is covered in sections 5.3.1, 5.3.2, and 5.3.4). The reason for which the replay system must specially handle this type of transaction is to ensure that interception of messages always occurs at the node where the task that currently holds the receive rights for a certain port resides. A shift of a receive right between tasks on the same node doesn't trigger any interception set-up adjustment being taken on the part of the recording/replay subsystem of the debugger.

It is assumed that port receive capabilities can only be passed to another task in a message (no netname server can be involved) from the current holder of the capability to the intended task, and that only one task can hold the receive rights to any port at any **instance** of time. Also it must be noted that a receive right can only be sent to another task *t* if the sending task has send rights to a port that task *t* has receive rights on. This implies that message interception should already be set-up, thus the receive right will be intercepted by the [REP]CONTROLSERV.

#### Recording Phase

A receive port right being passed in a message is followed by a control message indicating the virtual name of the sending task-thread, the node from which the receive right is coming, and the virtual name of the port whose receive right is being passed. When the CONTROLSERV notices that it just intercepted a receive right in transit from **another** node, it sends an RPC to the CONTROLSERV, on the node where the receive right in-transit formally resided, to request the receive-rights to all the **intercept** ports that represented send rights other tasks had to the port whose receive rights has just moved be moved to the CONTROLSERV on the new node. Then, the CONTROLSERV that intercepted the receive right in-transit forwards the receive rights it received toward the intended port. The following algorithms (EA, EB, EC and ED), all in abridged format, demonstrate the relevant portions of various debugging libraries during the *recording* phase for transferring the interception set-up when a receive right is transferred to a task on a different node. A graphic description of the protocol in found in figure 5.8.

Algorithm EA describes the relevant portions of rec_msg_send() that apply to transferring receive rights. The delay of 1 second on line EA4 is compensated by the fact that it is highly unlikely, but nonetheless possible, that the port will be considered "unmoveable" (due to an interception set-up in progress) when a task decides it wants to move its receive rights to a task on another node. This is because interception set-up of a send-right to a port only occurs once for a given task throughout its lifetime.

```
EA : rec_msg_send() /* abridged algorithm */
EA1: /* it has been determined that a receive right to some port
     p is being transferred */
EA2: send RPC to GPNAMESERV to determine if receive rights to port p
     can be transferred (no set-up interception currently in progress).
EA3: WHILE (receive rights to port p cannot be moved)
EA4:         wait for 1 second.
EA5:         send RPC to GPNAMESERV to determine if port
             p can be transferred (no set-up interception
             currently in progress).
     ENDWHILE
EA6: msg_send()./* send the receive rights-will be intercepted by
                the CONTROLSERV on the remote node */
EA7: send control message, which indicates the virtual name of
     the sending task-thread, the virtual name of the receive rights,
     and the node where receive rights used to be.
```

The InterceptThreads, which each dequeues from a group of intercept ports that each represent send rights to a particular application port at the CONTROLSERV at the node where a receive right is arriving, performs the following actions (algorithm EB) on the receipt of a receive right:

```
EB: InterceptThread()
EB1: receive (by interception) the receive right.
EB2: send RPC request to CONTROLSERV on the remote node where
       the receive rights are coming from, requesting a transfer of
       the interception to this node.
EB3: receive receive-rights to transferred interception ports (reply
       to EB2 RPC request).
EB4: IF (enqueuing log for port whose receive rights it just
       received don't yet exist on this node's LOGSERV)
EB5:     start such a logical log.
         /* if the logical log exists, then future logs to this
         logical log will be appended to the existing log */
       ENDIF
EB6: forward the receive right to the intended port.
```

Each CONTROLSERV has a dedicated service port for RPC requests, such as for requests to transfer interception schemes for a real application ports whose receive rights have moved to another node. The CONTROLSERV intercepting such traveling receive rights will send an request to the CONTROLSERV on the other node demanding the service of transferring interception schemes be performed (see algorithm EC).

```
EC: RPCServerThread() /* at CONTROLSERV at node from
                         which receive right has just been sent from */
EC1: receive RPC from remote CONTROLSERV that just intercepted
       a receive-right that was sent in a message.
```

```
EC2: stop dequeuing application messages for the intercept port(s) representing
     the various tasks that have send rights to the port whose receive
     rights have been transferred to another node.
EC3: mark in enqueuing log that receive rights were transferred to
     another node at that point.
EC4: forward remaining messages, that were dequeued from the intercept
     ports that are associated with the port whose receive rights are
     moving to another node, to the intended (application) port.
EC5: stop forwarding messages to the intended port.
EC6: send receive-rights of relevant intercept ports to the
     calling RPC client (CONTROLSERV).
```

Finally, algorithm ED shows what special action may have to be taken when the receive right is finally received by the application on the different node:

```
ED: rec_msg_receive()
ED1: IF (a dequeuing log for the port it just got receive rights for
     doesn't exist yet on this node)
ED2:    create a new logical log for the dequeuing log of port
        it just got receive rights for.
     ENDIF
```

## Replay Phase

During the *replay* phase, transferral of a receive right is handled in essentially the same manner as during the *recording* phase. The only complication is at the REPCONTROLSERV, where during the *recording* phase, the CONTROLSERV dequeues a specific number of messages which it forwards to the intended port before the interception scheme is moved to a CONTROLSERV on a new node. The REPCONTROLSERV must dequeue the same messages before allowing the interception scheme to be moved.

The following abridged algorithm (EF) is for the InterceptThread in REPCONTROLSERV on the node that has just intercepted a receive right that is being transferred from another node in a message. The sleeping InterceptRequestThread's comes from algorithm AA (see page 97):

```
EF: InterceptThread()
EF1: receive (by interception) the receive right.
EF2: send RPC request to REPCONTROLSERV on the remote node
     where the receive right is coming from, requesting a transfer
     of the interception to this node.
EF3: get receive rights to the transferred interception ports
     (in reply message to the RPC request [in EF2]), as well as
     receive data on any sleeping InterceptRequestThread's that could not
     forward to the port whose receive rights are being transferred
     on the old node due to the fact that the execution history
     of the "moving" port resumes on this node.
EF4: The transferred pending InterceptRequestThreads are restarted on
     this REPCONTROLSERV.
```

The "sleeping InterceptRequestThread's" mentioned in line EF3 refer to messages that the REPCONTROLSERV received out-of-order with respect to the execution history. Thus the REPCONTROLSERV is delaying the forwarding of these messages (see algorithm AA on page 97) to the intended port.

The following algorithm (EG) is for the RPCServerThread in REPCONTROLSERV on the node where a task is giving away a receive right. Note that the InterceptThread is multi-threaded during the *replay* phase in that it spawns InterceptRequestThreads which serve to forward the message to the destination port at the correct time. This implies that the REPCONTROLSERV may have pending RequestThreads that represent intercepted messages that were intercepted at a different node during the *recording* phase. The following algorithm takes this possibility into account (see lines EG5 and EG6).

```
EG: RPCServerThread()
EG1: receive RPC from remote REPCONTROLSERV that just
     intercepted a receive right that was sent in a message
     requesting transferral of the interception scheme to another
     node.
EG2: IF (all the messages that were intercepted and then
     forwarded before the RPC request came during the recording
     phase have not happened yet)
EG3:     wait for the missing messages to arrive and
         forward them.
     ENDIF
EG4: stop dequeuing application messages for the intercept port(s)
     representing the various tasks that have send rights to the
     port whose receive rights have been transferred to another node.
EG5: send receive-rights of the relevant intercept ports to
     the calling RPC client (REPCONTROLSERV), as well as the
     specifics of any pending request-threads that represent
     intercepted messages that were originally intercepted on another
     node during the recording phase.
EG6: kill pending RequestThreads that were "transferred" to
     the remote node where the receive rights were moved to.
```

### 5.3.4 Netname Server

#### Recording Phase

The debugging platform assumes that there exists a **netname** server on each node of the system, which allows arbitrary tasks the ability to obtain send rights to well-known (by some text string) application ports. Only ports whose send rights are explicitly registered under a well-known pseudo-name at a specific node's netname server can be "looked-up". Furthermore, a **broadcast** request to all nodes (netname servers) for a specific pseudo-name can be performed. Each netname server is assumed to work independently on each node and is considered a minimal "bootstrap" server on which more sophisticated name servers can be built upon.

Again, all netname server related primitives are transformed into special function calls to primitive-specific debugger routines during the transformation of the program prior to the start of the *recording* phase.

The chief difficulty of incorporating a netname server into a monitoring/replay system that dynamically arranges for message interception as new ports and/or new or existing port rights are distributed or rearranged is that a typical netname server does not maintain or provide the **virtual** name (see section 5.1.5) of the port it is distributing (on request), thus making monitoring difficult. Another difficulty is that a recipient of a "looked-up" send-right cannot determine on which node the receive rights are located, making the process of interception set-up seemingly impossible (message interception is always arranged at the CONTROLSERV on the node where a port's receive right resides).

When receiving a port right in an application message, the control message that follows it contains the above mentioned required information (virtual port ID and location of receive rights). A netname server is a public server which cannot be recompiled at will, and is often incorporated into the kernel itself (in some O.S.'s).

In order to achieve the goal of making the debugger as portable as possible, the netname server was not altered for the purposes of monitoring (or replay). Instead of having the netname server send a **control** message describing the port rights it is sending, there exists a server (GPNAMESERV, see section 5.2.5) on each node that keeps track of the virtual names of all send rights to ports which are registered on the same node's netname server, as well as the location of the registered port's receive rights. Specific debugger library routines request the necessary information when required from the GPNAMESERV. For `rec_netname_checkin()` (which replaces the real `netname_checkin()`), it is described by Algorithm E. It is assumed that a task can only check in a port it has send rights to, and a task can only check in to the local netname server.

```
E1: OS_return_code = netname_check_in().
E2: IF OS_return_code == failure
E3:     return OS_return_code (see A1) to the application program.
E4:     record type of netname_check_in() failure.
E5:     exit this routine.
    ENDIF
E6: place an entry in the local GPNAMESERV, mapping the
    netname string to the virtual name of the port the local
    netname server now has registered.
    /* virtual port to O.S. port mapping is found in a data within the task itself */
E7: record type of netname_check_in() success-code.
E8: return OS_return_code (see A1) to the application program.
```

The result of **netname_checkin()** calls (success or failure codes) are recorded in the calling thread's netname system call history log (see table 5.12). The combination of these calls can be non-deterministic in that it is possible numerous tasks can be involved in race-conditions to attempt to request various services of the netname server which cannot not be guaranteed to be reproducible between re-executions. For example, two tasks can be racing to check-in the same "netname" on the same node's netname server.

The **rec_netname_checkout()** routine sends an RPC to the local GPNAMESERV, instructing it to remove the netname-to-virtual-name tuple that contains the netname specified in the **netname_checkout()** call. The results of this system call are also recorded in the netname system call history log associated with the calling thread.

For **rec_netname_lookup()**, it logs the result of each look-up attempt (success or failure) in the task-thread's history log (see 'able 5.12), as well as the node where the look-up succeeded if a broadcast look-up was involved. Algorithm F explains the procedure:

```
F1: IF (a netname broadcast lookup attempt)
F2:     look-up each node's netname server in some random sequence
```

```
          until the name is found (and thus logged) or all nodes
          have been searched once without finding the specified netname.
F3:       IF (netname lookup a failure)
F4:           log error code in the task-thread log.
F5:           exit this routine.
          ENDIF
F6:       log the name of the node where the look-up succeeded
          in the task-thread log.
F7:  ELSE /* direct lookup at a specific node */
F8:           record the result of the netname look-up.
     ENDIF
F9:  IF send right to port p just received was not previously owned
     by this task
F10:      go to GPNAMESERV on node where the netname lookup
          succeeded, obtain the virtual name of the port send
          rights it just obtained, and put the virtual name to
          portname mapping within the task's debugger data structure
          mapping virtual to real port names.
F11:      broadcast to all gpnameserv's (on all nodes) to determine
          location l of the receive rights of the port p.
F12:      send RPC to CONTROLSERV on node l asking it
          to arrange message interception for newly acquired send right.
     ENDIF
F13:return OS_return_code to the application program
     which called netname_lookup().
```

## Replay Phase

The netname system calls are transformed into *replay* phase specific library calls, which call primitive-specific debugging routines, which perform additional duties in

|  |  |
|---|---|
| Who adds to log | The task-thread that makes a netname lookup call (via a macro). |
| Kind of data | The result of the netname call. If a broadcast lookup, the node from which the lookup succeeded is recorded as well. |
| Who created log | Application task-thread then thread is created by forker thread, or init_cdb() if initial thread. |
| Location of log | The physical log dedicated to a particular application task. |

Table 5.12: History of netname system call results

addition to actually executing the calls they are emulating.

Both the rep_netname_check_in() and rep_netname_check_out() debugger library routines check the history of netname system call results for the calling thread before actually executing it. If the history log states that the netname call was a success during the *recording* phase, the call is executed immediately–otherwise, the netname call is not executed and the library routine returns the error-code stated in the history log to the application program, thus simulating a faulty netname call. Due to random delays during the *replay* phase, a successful netname_check_in() or netname_check_out() call during the *recording* phase can still fail during *replay* phase. To compensate, their respective debugging library routines will repeatedly retry the failed netname call until the call succeeds, and only then will the debugging library routine return a success code to the calling application program (see algorithm CC).

For the netname_lookup() primitive that doesn't involve a broadcast lookup (*i.e.*, the look-up is directed at a specific node), just as for the netname_check_in() and netname_check_out() calls, the debugger library routine associated with the call first checks the netname execution history for the result of the call during the *recording* phase. If it was a failure during the *recording* phase, a failure is simulated during the *replay* phase. Otherwise, the call is tried until it eventually must succeed (see algorithm CC).

For the netname_lookup() primitive that involves a broadcast lookup, the associated debugger library routine determines from the netname history log (for the calling thread-task) whether the broadcast was successful, and if yes, from which node did the broadcast succeed in acquiring send rights (in the case when more than one netname server has the same "name" registered). This way, the debugger routine can make a "persistent" netname look-up *only at the node* where the broadcast lookup succeeded during the *recording* phase. Algorithm CC explains the rep_netname_lookup() routine:

```
CC: rep_netname_look_up(hostname, portname, portid)
CC1: get the next log element for this virtual task-thread from the
     special netname log (for the calling task-thread), which will
     consist of the result of the netname_look_up primitive.
CC2: IF (netname failed during the recording phase)
        actual call not attempted; instead, the error code recorded
        during the recording phase is simply passed back to the
        application program.
CC3:    exit this routine.
     ENDIF
CC4: IF (this is a "broadcast" lookup)
CC5:    get from the execution history (for the calling thread-task)
        the node name where the look-up succeeded.
     ENDIF
     /* execute the netname_lookup(), but instead of a broadcast,
     lookup netname at the node where it succeeded during the
     recording phase */
CC6: return_code = netname_look_up(nodename).
CC7: WHILE (return_code != KERN_SUCCESS)
CC8:    pause 1 second.
CC9:    return_code = netname_look_up(nodename).
     ENDWHILE
CC10:IF (send rights to this port don't already exist)
        /* ie. interception not yet set-up for this send right */
```

```
CC11:   broadcast a request message to all gpnameserv's, for
        the purpose of asking where (on which node) the receive
        rights are located for the send rights just received
        from the netnameserv.
CC12:   send a request for interception set-up for the newly
        acquired send right to the repcontrolserv on the node
        where the receive rights to the port reside.
    ENDIF
```

### 5.3.5 Non-deterministic system calls

Most modern operating systems have primitives that allow the user to choose some value to a parameter or to allow the operating system to select it. One prime example is the allocation of virtual memory-by default, the operating system may allocate the requested size at a location of its choice or the user may specify a starting address. The result of all such non-deterministic calls in which the operating system is allowed to choose a parameter is logged during the *recording* phase (in the calling task/thread's specific log, see table 5.13). The system call is then re-executed in the *replay* phase by explicitly specifying the same parameter that was chosen by the operating system during the *recording* phase, thus rendering the system call deterministic during the *replay* phase.

| | |
|---|---|
| Who adds to log | task-thread that does a non-det. system call |
| Kind of data | the result of the non-deterministic call |
| Who created log | application task/thread when thread is created (by forker-thread or init_cdb if initial thread of task) |
| Location of log | In the physical log file dedicated to a particular application task |

Table 5.13: History of non-deterministic system calls

### 5.3.6 Threads and Synchronization

In a multi-threaded environment where threads share a common address space, the shared spaces are often protected by explicit mutual exclusion constructs. Synchronization primitives (condition variables) together with mutual exclusion constructs are used to constrain the possible interleavings of threads' execution streams. The resultant approach (as is used in C-Threads [6], which is used in the debugger prototype) separates the two most common uses of Dijksta's `P()` and `V()` operations into distinct facilities and thus basically implements monitors, but without the syntactic sugar [6].

The order in which threads gain exclusive entry to given mutual exclusion constructs can influence the paths and states of execution the threads take. This form of non-determinism can thus mask bugs and errors in the program from re-execution to re-execution, depending on CPU scheduling and random delays. Thus, the sequence of thread entry to every mutual exclusion construct must be recorded during the *recording* phase and enforced during the *replay* phase.

As well, "condition signals" sent to sleeping threads must be recorded as to which thread they woke-up, since a particular signal will wake-up exactly one specific thread, even if more than one thread is waiting on a particular condition_wait() primitive. If a broadcast signal is sent, the identities of all recipient threads of the broadcast must be recorded (during the *recording* phase) so that it can be arranged during the *replay* phase that exactly the same group of threads will receive the signal[5]. This is more complex than it seems, since for example, during the *replay* phase, the original recipient of a condition signal may not be ready to receive a signal (hasn't gone to sleep on the appropriate condition_wait() primitive) when the sender thread of the signal is ready to "fire". The *replay* phase must coordinate the reconstruction of non-deterministic events among threads.

The events of interest to the monitoring/re-execution controller are:

1. the sequence and instance in which threads gain **entry** to mutual exclusion constructs.

[5]a thread must be waiting in a condition_wait()

2. the instance in which threads **surrender** exclusive access to mutual exclusion constructs (applies to the *replay* phase). This is done so that the replay controller can grant access to the mutex to the next thread listed in the execution history.

3. the threads that **received** a "signal" to "wake-up" from their sleep state (on a condition_wait() primitive) and which thread sent the signal.

It is assumed that debugging will take place in a **pre-emptive** multi-threaded environment (not a co-routine implementation) and that the C-Threads package is used [6]. During transformation of the application program, most C-Thread primitives are selectively replaced with specific debugger library calls, which perform the C-Thread call they replace, as well as relevant monitoring/replay duties related to the call. Only the C-thread routines related to condition_signal()'s, so that the debugger can obtain the O.S.'s thread-id(s) that just caught a signal sent, needed to be changed. This is necessary because an efficient mechanism for obtaining the aforementioned thread-id's by simply augmenting the source code was not found.

### Virtual Naming of Thread Resources

Condition variables and mutex's are given identical virtual names during the *recording* and *replay* phases so that the execution histories can match up with the correct thread resources during a reexecution of a multi-threaded application program. This is achieved by doing the following: on allocation of a condition variable or mutex, the virtual name of the thread that allocated it is recorded in a **log of condition variable allocation** or **log of mutex allocation**, as the case may be (see tables 5.14 and 5.15). The allocation of new condition variables and new mutexes are sequentialized respectively by making the allocation of the resource class and the granting of a virtual name for it *atomic* by protecting the operation with a mutual exclusion construct for that purpose. During the *replay* phase, the replay subsystem *guarantees* that the same *sequence* of threads that allocated condition variables or mutexes during the *recording* phase would allocate them again in the same sequence so that the granted *virtual name* would be the same during both phases of debugging. The virtual name consists of an integer that is incremented after assigning a name to each resource (condition variable and mutex) respectively as they are created.

| | |
|---|---|
| Who adds to log | task-thread (via debugger library routine) that successfully allocates a condition variable |
| Kind of data | the virtual name of the allocating thread |
| Who created log | application task (via init_cdb()) when task is first created |
| Location of log | physical log file dedicated to a particular application task |

Table 5.14: History of condition variable allocation

| | |
|---|---|
| Who adds to log | task-thread (via debugger library routine) that successfully allocates a condition variable |
| Kind of data | the virtual name of the allocating thread |
| Who created log | application task (via init_cdb()) when task is first created |
| Location of log | physical log file dedicated to a particular application task |

Table 5.15: History of mutex allocation

### Detection of Mutex Exit

During the *replay* phase, detection of the exit of a thread from a mutual exclusion construct is important so that the replay controller can give permission to the next thread in the mutex's log to enter.

The following special conditions must be watched for and handled in a specific manner in order to detect a mutex becoming unlocked:

- Mutex unlock operations on *already unlocked* mutexes must be detected, since they are *false* mutex unlock operations (logical no-ops). This can be remedied by executing "mutex_try_lock(*lock*)" once before executing the real mutex unlock operation (either mutex_unlock(*lock*) or condition_wait(cond, *lock*)[6]). If mutex_try_lock(*lock*) returns TRUE, it means *lock* was not locked, and therefore, the mutex unlock operation is a false mutex unlock (debugging library routine must mutex_unlock(*lock*) immediately). If mutex_try_lock(*lock*) returns FALSE, then the macro must check if the lock is held by *another* thread (thus making the mutex_unlock(*lock*) false as well). This can be accomplished by maintaining a special data structure within the task that indicates which thread presently has the lock for a given mutex. This data structure is updated each time a mutex is entered and exited.

- If a thread aborts (or exits) while holding some lock, the lock will be permanently locked. This is normal C-Thread behavior, and therefore the monitoring/replay subsystem will not take any special action to defeat this normal behavior.

- If a condition_wait(*c*, *lock*) is not properly preceded by a mutex_lock(lock), the implicit mutex_unlock(lock) when condition_wait(c, lock) executes will be a no-op. The calling thread will sleep on the wait queue until *c* arrives.

1. **mutex_unlock(lock)** A macro during the *replay* phase will transform mutex_unlock() to rep_mutex_unlock(), which performs the following routine:

   Note: There is no need for a "super-mutex"[7] to surround the entire macro (*i.e.*, to sequentialize all mutex_unlock(lock) operations), since all that must be determined is whether the calling thread that wants to unlock "lock" truly owns "lock". During the *replay* phase, a "mutex_unlock()" is never artificially blocked, but it must be reported so that the next thread that should enter the mutex according to the execution histories is given permission to do so.

[6]implicit mutex_unlock() on execution

[7]a mutex transparently created by the debugger to assist it controlling the monitoring/execution of multi-threaded tasks by restricting threads from executing certain calls when need be

```
boolean_var = mutex_try_lock(lock);
if (boolean_var == TRUE) { /* lock was unlocked => false mutex
                              unlock */
    mutex_unlock(lock); /* let go of acquired lock immediately */
    /* do not inform a mutex was unlocked, since it really wasn't */
    mutex_unlock(lock); /* real, although a logical noop */
}
else  { /* check if lock is currently held by the CALLING thread */
    cthread-id = lock_holder_lookup(lock);
    if (cthread-id == cthread_self())  { /* lock is currently held
                                            by calling thread */
        mutex_lock(super_lock(lock)); /* make report atomic with
                                         mutex_unlock so that no
                                         other thread can unlock
                                         "lock" until this thread
                                         reports AND unlocks */
            inform_MUCF_of_mutex_unlock(lock);
            mutex_unlock(lock); /* real */
        mutex_unlock(super_lock(lock));
    }
    else   /* cthread-id != cthread_self() */
        mutex_unlock(lock); /* real, even though logical noop */
```

2. **mutex_unlock(lock)** No macro during the *recording* phase is needed, since only mutex entry is recorded during this phase "as it happens" (thread gets access, and doesn't just block at mutex_unlock()).

3. **condition_wait(c, lock)** A macro during the *recording* phase will **not** expand this call for the purposes of mutex unlock detection (but it is expanded for when this primitive implicitly takes the lock, see below).

   There is no need to inform the Mutex_unlock Coordinator and Flagger daemon thread (MUCF, see page 131 for more details) of the implicit mutex_unlock() during the *recording* phase, since all that must be done is to execute condition_wait() and log when the calling thread actually enters the mutex.

4. **condition_wait(cond, lock)** A macro during the *replay* phase will expand this call to the following algorithm. "condition_wait() statements are artificially held back in this macro in order to prevent threads from receiving the wrong signals at the wrong time, relative to the execution history:

```
/* ****************** utility function of 4.******************* */
rep_control_condition(virtual-thread-id, condition, mutex)
{
    /* ''cc'' thread coordinates replay of threads */
    tell_cc_ready_to_do_condition_wait_when_allowed(virtual-thread-id,
                                                     mutex);

    mutex_lock(super_lock(cond)); /* the mutex is condition-specific,
                                     so that the cc can update which
                                     thread can execute the real
                                     condition_wait() (a shared memory
                                     variable) next without being
                                     subjected to any race
                                     conditions */

        /* the calling thread will have to check its ID with the
           variable that's set by the cc in order to determine
           whether the thread can proceed to execute the real
           condition_wait. */
        while !(cc_gives_me_permission_to_execute)
            condition_wait(cc_c(c), super_lock(c));
            /* "super_cond(cond)" is a condition specific signal */


    mutex_unlock(super_lock(cond));

    /* at this point, the thread has permission to execute the real
       condition wait  */
    /* a policy decision: the replay controller only gives permission
       to wake-up a thread (among other conditions) when it will be
       the ONLY candidate to be scheduled to actually get the
       lock. */

    inform_MUCF_of_mutex_unlock(lock); /* due to upcoming
                                          condition_wait() */

    condition_wait(cond, lock);
    /* real, although mutex_unlock is noop */

    inform_MUCF_of_mutex_lock_acquired(lock, virtual-thread-id);
    /* due to condition_wait() receiving a signal */
 }

 /* *******************end of utility function**************** */
/* ****************** main body of macro*********************** */
   /* does "lock" really belong to me? */

   boolean_var = mutex_try_lock(lock);
   if (boolean_var == TRUE) { /* lock was unlocked => false
                                 mutex unlock */
       mutex_unlock(lock); /* let go of acquired lock immediately */
       /* do not inform of mutex_unlock, since it's a noop */

       rep_control_condition(virtual-thread-id, cond, mutex);

   }
   else
       {  /* check if lock is currently held by the CALLING thread */
          cthread-id = lock_holder_lookup(lock);
          if (cthread-id == cthread_self())  { /* calling thread
                                                  holds lock */
              /* no super-mutex required here, since the replay
                 controller will never allow more than one thread
                 to be waiting for a lock.  Since the waiting
                 thread will never get the lock until the previous
                 thread has let it go (within the MACRO), there
                 is no chance of a race, even without using a
                 super-mutex */
              inform_MUCF_of_upcoming_mutex_unlock(mutex);

              rep_control_condition(virtual-thread-id, cond, mutex);


          }
          else  /* thread-id != cthread_self() */

              /* mutex_unlock is unreported, since it's a noop */

              rep_control_condition(virtual-thread-id, cond, mutex);
```

### Detecting and Handling Mutex Lock

When a thread wants to enter a mutex, it must coordinate with the recording/replay controller so that:

1. during the *recording* phase, the mutex entry is logged only when the calling thread has in fact entered the mutex, since a thread that has executed mutex_lock() can be put on a wait queue if the mutex is presently locked and,

2. during the *replay* phase, the thread (within a transparent debugging library routine) must ask and be given permission to enter a mutex from the MUCF before it actually executes a real mutex_lock(). In otherwords, all mutex_lock() operations are intercepted and actual entry into the mutex are serialized and order adjusted, if necessary, by the replay monitor according to the recorded data.

During the *replay* phase, a thread which has been given permission to enter a mutex *must be guaranteed* that the mutex will be entered immediately after the thread executes mutex_lock() by guaranteeing that the lock is unlocked at that time and that no other thread can possibly beat it to the mutex by not giving any other thread to permission to execute mutex_lock() on the lock in question. The thread that now has permission to enter the mutex is placed in the sleep state *alone* waiting for that mutex to become free (if it isn't so already) so that it won't have to race with any other thread to gain exclusive access to that mutex. A log exists for e· ·ry mutex that is allocated by an application task. A log element of a mutex entry is shown in table 5.16.

| | |
|---|---|
| Who adds to log | task-thread that was granted entry into mutex (via debugger library) |
| Kind of data | the virtual thread-id that gained entry into the mutex |
| Who created log | Application task-thread when it allocates a new mutex |
| Location of log | In the physical log file dedicated to a particular application task |

Table 5.16: History of mutex entry in a specific mutex

1. **mutex_lock(lock)** During the *recording* phase, this primitive will be expanded to:

```
mutex_lock(lock);
log_mutex_entry(virtual_thread_id, virtual_mutex_id);
```

2. **mutex_lock(lock)** During the *replay* phase, this primitive will be expanded to the following routine. A lock-specific "super-mutex" is needed to to give the thread exclusive access when it checks a shared variable to determine whether it is its turn to grab the mutex lock:

```
mutex_lock(super_lock(lock));  /* mutex specific */

   while (!(do_I_have_permission_to_enter(virtual-thread-id,
                                          mutex)))
        condition_wait(mucf_c, super_lock(lock));

   mutex_lock(lock);
   inform_MUCF_of_mutex_lock_acquired(lock, virtual-thread-id);

mutex_unlock(super_lock(lock));
```

Handling the "mutex_try_lock()" primitive is handled somewhat differently (with respect to the blocking mutex_lock() primitive). Since it is a non-blocking call, if the call failed, the thread may have done other things, possibly changing the state of its holding task, possibly changing the state depending on how many times mutex_try_lock() failed, which can vary from execution to execution (see Table 5.17).

1. **mutex_try_lock(lock)** During the *recording* phase, a macro will expand this call to:

```
mutex_lock(super_lock(lock));  /* mutex specific */
  if (mutex_try_lock(lock)) {
       log successful mutex_try_lock() attempt in thread's execution
       history of ''mutex_try_lock's'';
       log_mutex_entry(virtual_thread_id, virtual_mutex_id);
  }
```

| | |
|---|---|
| Who adds to log | The task-thread that just attempted mutex_try_lock() |
| Kind of data | The return code of the mutex_try_lock() primitive |
| Who created log | Application task-thread when it is created (via init_cdb()) |
| Location of log | In the physical file dedicated to a particular application task |

Table 5.17: History of mutex_try_lock()

```
  else /* mutex_try_lock(lock) failed, so log failure attempt */
       log unsuccessful mutex_try_lock() attempt in thread's execution
       history of ''mutex_try_lock's'';
mutex_unlock(super_lock(lock));
```

2. **mutex_try_lock(lock)** During the *replay* phase, the mutex_try_lock(lock) primitive will expand to:

```
mutex_lock(super_lock(lock));  /* lock specific */

   if (thread's execution history of ''mutex_try_lock's'' indicates
       that the call failed)
       /* simulate a failed mutex\_try\_lock(lock) by returning error
       code to the application program */
       return(failure code);

   /* if this point reached, thread's execution history indicate that
      the mutex\_try\_lock(lock) should succeed at this point */

   if (do_I_have_permission_to_enter(virtual-thread-id,
                                         virtual_mutex_id)) {
      bool = mutex_try_lock(lock);  /* guaranteed to work */
      if (!bool)
          exit(-1); /* something wrong with replay monitor,
                       this should not happen */
      else  { /* bool==TRUE */
          inform_MUCF_of_mutex_lock_acquired(lock,
                                                     virtual_thread_id);
          return TRUE;
```

```
    }
    else  /* I don't have permission, so just pretend
             mutex_try_lock had returned false (didn't
             attempt to execute a mutex_try_lock) */
          return FALSE;
  mutex_unlock(super_lock(lock));
```

### Handling Condition Signals

During the *recording* phase, the (virtual) ID of the thread that sends a signal must be recorded, and the resulting thread(s) that caught the signal (via a condition_wait() statement), if any, must be recorded as well. This is done because the receipt of a signal will eventually unblock the recipient thread, which is then free to execute and possibly alter the state of the task it finds itself in. The C-Thread library is altered in order to provide the debugger with the thread id's that "caught" a signal sent (multiple thread-id's provided if more than one thread caught a "broadcast" signal). A log of signals sent, for *each specific signal declared within a task*, is collected during the recording phase (see table 5.18).

| | |
|---|---|
| Who adds to log | The task-thread that just sent the signal (via a macro) |
| Kind of data | The name of the virtual task-thread that sent the signal |
| Who created log | Application task-thread when it allocates a new "condition variable" |
| Location of log | In the physical log file dedicated to a particular application task |

Table 5.18: History of signals sent

As well, for *each specific signal declared within a task*, the threads that "caught" a signal are recorded (see table 5.19).

- It is assumed that all signals of a given task are sequentialized (for each condition_signal()) during the *recording* and *replay* phases in order to eliminate any

| | |
|---|---|
| Who adds to log | task-thread that sent the signal (it requests and receives the information from the *altered* C-Thread library) |
| Kind of data | The virtual name(s) of the threads that "caught" the signal (if more than one, then **all** names are placed in the same compound log entry) |
| Who created log | Application task-thread when it allocates a new "condition variable" |
| Location of log | In the physical log file dedicated to a particular application task |

Table 5.19: History of a signal being caught

possible race conditions during both phases with respect to the occurrence of a signal and its recording (during the *recording* phase) or a signal's permission to be executed (during the *replay* phase). The **condition coordinator** (CC) thread (see page 132) coordinates this activity.

1. **condition_signal(c)** A macro during the *recording* phase will expand this call to:

```
mutex_lock(super_c(c));  /* a condition specific lock,
                            no other thread can signal "c" before
                            log is written */
   thread_awakened = condition_signal(c);  /* could be NULL,
                                              if nobody waiting */
   /* C-Thread's condition_signal() libthreads is modified to be
      able to return "thread-awakened" */
   log_condition_sent(virtual-calling-thread-id,
                      condition, thread_awakened);

mutex_unlock(super_c(c));
```

2. **condition_signal(c)** A macro during the *replay* phase will expand this call to the following algorithm. The CC (Condition Coordinator) will ensure that the thread that caught the signal during the *recording* phase will do so during the *replay* phase. Otherwise, if the signal that originally caught the signal has

not executed the appropriate condition_wait() primitive, the signal may not be caught by *any* thread when sent:

```
mutex_lock(super_c(c)); /* a signal specific lock,
                           no other thread can signal "c" or start
                           a new "condition_wait(c, *)", so that a
                           specific signal will be caught by a
                           specific thread */

  while (!(do_I_have_permission_to_send_c(virtual-thread-id, c))
          condition_wait(cc_c(c), super_c(c));
          /* if I don't have permission, give the thread that has
             permission a chance to exercise its permission and
             wait for the CC to signal when this thread MIGHT
             have permission */

  /* The thread executing this code really doesn't need to
     let go of the "super_c" lock here,
     since no other thread that wants to signal "c" can do so
     until this one has done so. But, this is a convenient
     way of expressing it, to make sure that the two conditions
     are met before a thread signals  */

  while (!(is_thread_ready_to_receive(c)))

     /* at this point, no other thread has permission to
        send "c", so CC knows which thread is making
        this request. Also, this thread is guaranteed that
        only one original thread is activated to
        receive "c" when it should. Once the
        receiving thread gets the "c", it will NOT have to
        compete with any other thread to get the lock
        (max 1 on the wait queue), since the replay
        controller guarantees that there will be no
        other threads on wait queues waiting for the
        mutex. */

      condition_wait(cc_c(c), super_c(c));


   condition_signal(c); /* real */
   advance_log(c);   /* by activating Condition Coordinator (CC) */

mutex_unlock(super_c(c));
```

3. **condition_broadcast(c)** A macro during the *recording* phase will expand this call to the following algorithm. The order in which the threads that were awakened is recorded in the "condition_wait()" macro expansion.

```
mutex_lock(super_c(c));  /* condition specific */

   threads_awakened = condition_broadcast(c); /* due to altered thread
                                                 library, it returns
                                                 an array of threads
                                                 awakened */

   log_condition_sent(virtual-calling-thread-id, c, threads_awakened);

mutex_unlock(super_c(c));
```

4. **condition_broadcast(c)** A policy decision: a broadcast will only be allowed to execute when all the threads that received the broadcast during the *recording* phase have all executed the appropriate condition_wait() during the *replay* phase. This is done because if a logical broadcast is broken-up into logical condition_signal()'s during replay, this can/will confuse the user debugging in that threads that have not yet reached a condition_wait() statement may appear to be catching signals. Therefore, a logical broadcast will only be executed when all threads that caught this signal during the *recording* phase are ready to execute condition_wait() (whenever CC gives them permission).

   A macro during the *replay* phase will expand this call to:

```
mutex_lock(super_c(c));  /* a condition specific lock,
                            no other thread can signal "c" */

   while (!(do_I_have_permission_to_send_c(virtual-thread-id, c)))
       /* if I don't have permission, then let go of mutex
          super_c(c) so that thread that does have permission
          can get inside the super_c(c) mutex */
       condition_wait(cc_c(c), super_c(c));

   /* at this point, no other thread has permission to send signal "c",
      so CC knows which thread is making this request.  Also at
      this point, all the threads which received this broadcast
```

```
           are ready to execute condition_wait().  The CC coordinates
           activating various condition_wait()'s (one at a time) and
           then activating various condition_signal() (one at a time)
           to send the signal to the activated condition_wait()'s. */

       while (!(are_all_threads_ready_to_receive_(c))
           /* CC signals this condition when true */
           condition_wait(cc_c(c), super_c(c));


       /* get # of times to signal */
       list = number_of_times_to_signal(c, cthread_self());

       for (sig_num =1; sig_num <= list; sig_num++)  {

           /* confirm each condition_wait() has permission to
              execute and that it is has been executed so that it is
              prepared to ''catch'' the signal c */
           while (!(is_thread_ready_to_receive(c, cthread_self())
               condition_wait(cc_c(c), super_c_lock(c);

           condition_signal(c);
           tell_cc_that_c_was_just_sent(c);  /* so CC can advance the log
                                                for signal ''c'' */

           /* wait for confirmation that the mutex was entered */
           while (!(confirm_mutex_entry(c))
               condition_wait(cc_c(c), super_c_lock(c));

       }

   mutex_unlock(super_c(c));
```

## Servers and Data Structures to Support Replay of Threads and Synchronization

Various auxiliary servers are required to support the macro extensions (calls to specific debugger library routines) to the various C-Thread primitives mainly during the *replay* and to a limited extent during the *recording* phase.

**Maintainer of lock holder** For each virtual_mutex_id in existence, it stores the virtual_thread_id that currently holds the virtual_mutex_id. If no virtual_thread_id holds the virtual_mutex_id, then the value of virtual_thread_id is NULL. There is one maintainer of "lock-holder" per task.

Operations of this server include:

- receive a report about a mutex_unlock() and adjust the database accordingly.
- receive a report about a mutex_lock() and adjust database accordingly.
- on lock_holder_lookup(virtual_mutex_id) request, RETURN the owner of the lock (virtual_thread_id), or NULL if no thread currently owns the lock.

N.B. If it receives a report for a mutex it has no record of, it creates an "entry" for the new mutex.

**Mutex_unlock coordinator and flagger (MUCF)** On receiving notification of any logical mutex_unlock(mutex), this server thread running within the application task advances the log for the "mutex" and determines from the log the next thread that can enter the mutex. It sets the mutex specific (replay internal) condition variable so that when the thread that has permission to enter attempts to enter by checking the condition variable, it will recognize itself and subsequently take the lock. Then this controller does an internal (invisible to application program) condition_broadcast(cc_c(mutex)) to wake-up every thread that is waiting to be given permission to enter that mutex (threads within "replay" macros and the "signal-coordinator" (N.B. setting the condition must be protected by a mutex (lock-specific))). There is one MUCF per application task.

**Note:** The CC must be involved when a thread gained entry to a mutex by receiving a signal while waiting in a "condition_wait()" primitive. If the execution history indicates that some thread $t$ entered some mutex $m$ via a condition_wait() and it's presently thread $t$'s turn to enter mutex $m$:

1. allow thread *t* to execute its condition_wait() primitive
2. check execution history to determine which thread sent the signal to thread *t* and allow this thread to send the signal when it reaches that point in its code (if it hasn't reached that point already)

Operations of the Mutex_unlock coordinator and flagger involve:

- receiving a report that a mutex has just been unlocked from an internal replay macro extension. The coordinator will advance the log, set the internal condition flag and broadcast the fact that the log has advanced. All this must be protected by a mutex-specific macro.

Note: There is no reason to look-out for new mutexes, since this thread runs in the same task and has access to all global mutexes and conditions.

**Condition Coordinator (CC)** In general, the condition_coordinator (CC) knows which thread-id is at the head of the list and thus sets the "do_I_have_permission_to_send_c" according (and independently). The thread attempting to signal will recognize that it has a right to do so, but will only signal when the condition coordinator (CC) sets another signal (and sends an internal broadcast signal so that threads waiting on the condition can check whether it is their turn to signal) that it can really signal the thread that should catch the signal and the recipient of the signal during the *recording* phase is guaranteed to catch the signal during the *replay* phase. This is guaranteed by the fact that only one thread at a time is allowed into the mutex, and that there is never more than one thread in the mutex-wait queue.

If the signal only went to one thread, then the controller will indicate to the thread that should do a condition_wait() and to do so *only* after it knows that it has a legal right to enter that mutex when condition_wait is executed (an unlocked mutex or empty condition-specific wait queue for the mutex is guaranteed when the condition_wait() is given permission to execute). The condition-coordinator will have to check if that thread is *next* in the execution history for the send sequence of a particular condition-signal. Then the CC checks to see if appropriate thread waiting for that signal (pending condition_wait()'s). After the CC receives confirmation of the successful condition_wait(), it releases the associated condition_signal(), by means of setting an internal variable and an internal broadcast, variable and and waits for confirmation that the signal was delivered. Once the entire operation is finished, it will advance the signal-specific logs.

**Else**, the signal went to more than one thread, broadcast during the *recording* phase. Therefore, this coordinator must give permission to the condition_broadcast() macro when all threads are ready to receive. Then, at this point, the macro will still have to wait for various permissions to condition_signal() (number of times equals the number of threads that originally caught the signal) so that when it does signal, it is guaranteed that the same thread will catch the signal by guaranteeing that the mutex is unlocked or that the wait queue (mutex-specific) is empty and will stay empty until the signal is delivered.

Only when all original number of signals are *confirmed* caught will the signal-specific queue be moved up by one.

# Chapter 6

# Conclusion and Suggestions for Future Work

## 6.1 Summary

The importance of using a cyclic debugging methodology on distributed programs is paramount. The challenge lies in developing a reliable monitoring/system that faithfully renders a monitored non-deterministic program functional on repeated reexecutions without distracting the user. The non-deterministic choices must be monitored in as unobtrusive a manner as possible durirg the *recording* phase so as not to bias the execution in any specific direction.

The monitoring/replay mechanism is designed at a low-level *without* actually altering the operating system kernel. This is not an unreasonable goal when using a microkernel such as Mach [1], since microkernels are designed to be extensible and allow most "traditional" kernel activity to be done in user-space. This approach enables the monitoring/replay subsystem to be easily portable to a variety of hardware platforms, due to the fact a microkernel's interface, for the most part, hides the peculiarities of the hardware it is running on while still allowing the user to use low-level operating system primitives. Since the monitoring/replay systems runs at a microkernel's system call level, and since a microkernel provides basic services so that more specialized services can be built on top of it, once all potential non-determistic events of a micro-kernel are monitored during the *recording* phase and enforced during the *replay* phase, all composite services that built from the micro-kernel components can use the monitoring/replay service.

An attempt was made to support the monitoring/replay of all (or as much as time would allow) the non-deterministic services of a typical message-based micro-kernel operating system that supports multi-threaded application programs at the kernel level (as opposed to "user-level" threads, in which the operating system is unaware of the existance of threads) in which the kernel schedules threads.

In addition, several debugging tools that take advantage of the guaranteed deterministic execution have been described.

In short, the design of a debugger sufficiently endowed with features and capabilities to be effective has been proposed. The base of the debugger, the underlying monitoring/replay system, and facilities proposed (such as pessimistic causal breakpoints) to take advantage of deterministic reexecution can be easily modified or expanded to provide more effective tools to debug distributed programs.

## 6.2 Suggestions

- The four debugger servers (LOGSERV, FORKSERV, GPNAMESERV, AND CONTROLSERV) can be combined into one multi-threaded server during the *recording* phase (and the counterpart servers for the *replay* phase) in order to minimize the interactions amongst them (on the same node) which now involves messages. A message operation results in a kernel trap, which may increase the probe effect.

- Currently, there is no strategy to deal with multi-threaded, shared-memory tasks, which have some portions of shared memory not properly protected by "mutex" constructs. There is no satifactory solution to this problem, it seems, in the literature.

- LOGSERV process the data it collects (during the *recording* phase) "on-the-fly". This processing activity can be done *between* the *recording* and *replay* phases, thereby reducing the probe effect somewhat during the *recording* phase.

- While there has been some progress on defining and implementing breakpoints in "process-based" distributed systems, a method of setting a breakpoint based on a composed predicate that is powerful yet relatively easy to use still needs to be developed. Furthermore, since more and more message-based distributed systems consist of a collection of *multi-threaded* cooperating servers, there has been little effort expended to date in defining debugging tools to work in such a hybrid system (threaded *and* message-based), based on replay (*e.g.*, breakpoint and stepping facilities).

- In general, more well-defined and useful tools that can assist the user in debugging a distributed system that consists of multi-threaded servers, each possessing their own protected address space and communicating with each other via messages, must be developed. There is currently no widely accepted multi-threaded debugger, let alone a multi-threaded debugger that is specifically designed to work within a distributed system.

- Implementation turned out to be a more time-consuming endeavor than originally anticipated, thus only a fraction of the features were fully implemented. It should be suggested that a project of this nature requires considerable manpower, as well as a sufficient time for the programmers to get acquainted with systems programming on a multi-threaded message-based micro-kernel.

## 6.3 Implemented Modules

Due to severe time constraints, the implementation of the design of the debugger is incomplete. None of the cyclic debugging tools as described in chapters 3 and 4 were implemented, although feasibility tests were performed to judge whether the design was reasonable.

The monitoring/replay mechanism, as described in chapter 5, was partially implemented. Support for multi-threaded application programs was not implemented but the portion of the monitoring/replay system that was implemented was designed with the multi-threaded support in mind. Monitoring/replay support for passing "receive" port capabilities was not implemented. Limited support for passing send rights in a message (as related to the monitoring/replay mechanism) was implemented in that a send right a synchronous RPC passes in order to grant the RPC server send-rights to the reply port is supported only. Passing send-rights via the *netname server* is fully supported.

# Appendix A

# Mach Related Definitions

**task** an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as port capabilities).

**thread** the basic unit of execution. It consists of all processor state (*e.g.*, CPU registers) necessary for independent execution. A thread executes in the virtual memory and port rights context of a single task. The conventional notion of **process** is, in Mach, represented by a task with a single thread of control.

**netmsgserver** The network server is responsible for extending the local Mach Inter-Process Communication abstraction over the network which interconnects Mach hosts.

**netname server** This is a network name service, which is used to acquire send rights to remote ports designated by a string name and an IP address (may be the broadcast address). It is implemented as a module on the netmsgserver. It operates by sending a netname message to the desired address, and waiting for a similar message in response.

**port** is a simplex communication channel, implemented as a message queue managed and protected by the kernel.

**port set** is a group of ports, implemented as a queue combining the message queues of the constituent ports. A thread may use a port set to receive a message sent to any of several ports.

**notify port** on which the task should attempt to receive notification of such kernel events as the destruction of a port to which it has send rights. Each task has receive rights to this port.

**kernel port** handle that is used in the task kernel calls to identify to the kernel which task is to be affected by the call.

**message** is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.

**mutex_lock(m)** attempts to lock the mutex *m* and blocks until it succeeds. If several threads attempt to lock the same mutex concurrently, one will succeed, and the others will block until *m* is unlocked. The case of a thread attempting to lock a mutex it has already locked is not treated specially; deadlock will result.

**mutex_try_lock(m)** The `mutex_try_lock()` function attempts to lock the mutex *m*, like `mutex_lock()`, and returns `TRUE` if it succeeds. If *m* is already locked, however, `mutex_try_lock()` immediately returns `FALSE` rather than blocking.

**mutex_unlock(m)** unlocks the mutex *m*, giving other threads a chance to lock it.

**condition_signal(c)** is called when one thread wishes to indicate that the condition represented by the condition variable is now true. If any other threads are waiting (via `condition_wait()`), then at least one of them will be awakened. If no threads are waiting, then nothing happens.

**condition_broadcast(c)** is similar to `condition_signal(c)`, except that it awakens *all* threads waiting for the condition, not just one of them.

**condition_wait(c, m)** unlocks *m*, suspends the calling thread until the specified condition is *likely* to be true, and locks *m* again when the thread resumes (after receiving signal $c$ and then acquiring the lock $m$. Since there is no guarantee that the condition will be true when the thread resumes, use of this procedure is usually of the form:

```
mutex_lock(m);
...
while (/* condition is not true */)
        condition_wait(c, m);
...
mutex_unlock(m);
```

# Appendix B

# Sources of Non-Determinism for Supported Application Platform

**Disclaimer:** The "race" conditions mentioned in this section do not refer to races that corrupt system data structures; rather it refers to potential non-determinism that may affect the outcome of an application program.

## B.1 Message sending

Several multi-threaded tasks can possess send rights to a particular port $p$. The order in which messages, that task-thread combinations send, *enqueue* at port $p$ is subject to a race condition.

The order in which the kernel sends **notify** messages (see appendix A) to a task's notify port (*i.e.*, their enqueuing order) is subject to a race condition. This is because the kernel sends **notify** messages in response to events that are often initiated by events in application tasks, which are subject to random delays.

A message send operation with a timeout specified may or may not "time-out" on subsequent reexecutions. A "send timeout" specifies that if a message was not able to enqueue at the target port within a specified amount of time from the sending of the message because the target port was full, the operation thus "timed-out".

Any thread within a task can send away a send right to some port $p$ to another task (it can also pass *copies* of a send right it has in its address space), thus taking away from the other threads in the task the right to send messages to port $p$. In a pre-emptive threaded environment, this is a potential race condition, since during subsequent reexecutions, certain threads may be able to send greater or fewer messages to port $p$, depending when the send rights were passed away in a message (or the send rights were simply deallocated by a thread within the task). A send right is shared by *all* threads within a task.

## B.2 Message receiving

A task's receive rights to a port is shared by *all* threads within the task. Consequently, there is a race amongst the threads to dequeue messages from the port in that the same threads may not dequeue the same message(s) on subsequent reexecutions.

A message receive operation with a timeout specified may or may not recur on subsequent reexecutions. A "receive timeout" specifies that if a thread executing a msg_receive() operation was not able to dequeue any message at the target port within a specified amount of time from the time the msg_receive() began executing, the operation thus "timed-out".

Since receive rights to a port can be transferred to another task (cannot be copied), the race condition described in the previous paragraph is compounded in that any thread can transfer receive rights to a port. On subsequent reexecutions, greater or fewer messages will be dequeued from the same port by the threads in the same *task*, depending if the thread(s) which intend to transfer (send in a message) the receive rights to some other task are delayed or not.

Furthermore, any thread within a task can destroy a port the task has send rights to. Therefore, there exists a potential race condition between threads that want to dequeue from some port $p$ and thread(s) that want to destroy port $p$.

## B.3 Port Sets

A port set (see appendix A) can be the site of several potential race conditions:

- Messages being sent to any port in a port set can be delayed. If all ports in a port set are empty except one, then the thread dequeuing from the port set must dequeue from the non-empty port.

- If two or more ports in a port set are non-empty, then the thread dequeuing from the port-set will randomly dequeue (not the choice of the dequeuing thread) from one of the non-empty ports in the port set.

- Any thread within a task that has receive rights to some port set can at any time add or remove a port (it has receive rights to) from a port set, thus potentially affecting the messages a thread dequeuing from a port set will receive.

## B.4 Kernel Primitives

In general, in a multi-threaded environment, calls to the kernel are potential races, since depending on timing and circumstance, a kernel call may or may not fail. For example, if two threads want to deallocate an existing port $p$, the first one to execute the kernel call will succeed, and the second thread will fail because port $p$ had already been deallocated.

Furthermore, certain kernel primitives can return different resources on different reexecutions. For example, asking the kernel to allocate a specific amount of memory an any location may return a different starting memory location on each reexecution.

## B.5 Netname Server

**netname check-in** A race condition can ensue if several attempts by various task/thread combinations to check-in the same name on the same netname server with different "signatures" are made. Once the name is checked in, all other attempts of checking in the same name (with different signatures) will result in an error.

**netname check-out** If more than two different task/thread entities wish to attempt to check-out the same netname entry from a netname server on the same node, only one will succeed.

In addition, there is a possibility that a task/thread will check-out a name from a netname server before everyone that had wanted to look it up had a chance to. On subsequent reexecutions, various task/thread's netname lookup attempts may or may not succeed, depending on the relative time a particular name is checked out from the netname server.

**netname look-up** The look-up call depends on the relative timings of the check-in and check-out primitives. Furthermore, if the name $n$ is looked for via a broadcast look-up and $n$ exists on *more* than one node, the node the caller finds the name on may differ on subsequent reexecutions. Identical netnames on different netname servers (situated on different nodes) may or may not represent send rights to the same port.

## B.6 Thread Synchronization

In general, all threads within a task *contend* to enter mutual exclusion constructs (mutex) in three possible ways:

1. the non-blocking primitive `mutex_try_lock(m)` (returns immediately if lock not acquired)

2. the blocking primitive `mutex_lock(m)` (blocks thread until mutex $m$ acquired).

3. the `condition_wait(c,m)` primitive blocks the thread until it first receives signal $c$, then it continues to block until the thread acquires mutex $m$ when it becomes free.

### B.6.1 Condition Variables

A signal $c$ will (arrive at) be caught by *one* thread that has executed `condition_wait(c, m)`. If more than one thread is at `condition_wait(c, m)`, only one thread will catch the signal. If no thread is waiting for signal $c$, then it will be lost and nothing happens. On subsequent reexecutions, the same thread may not catch the same signal either due to luck or the thread didn't execute `condition_wait()` in time.

If a signal is broadcast, then *all* threads *waiting* for $c$ will receive it. The set of threads *waiting* for a particular broadcast signal may vary on subsequent reexecutions. The order in which the group of threads catch a broadcast signal $c$ may not be in the same order in which the threads gain access to mutex $m$.

## B.7 Task Termination

In a multi-threaded environment, several threads may be in a race to terminate the task normally (*e.g.*, calling `exit()`) or abnormally (raising an exception, *e.g.*, divide by 0). The implication of this possibility is that other threads in the task will execute to a different extent on each reexecution, possibly creating a different final state for its task on each reexecution. In addition, the race condition as to when and which thread effectively kills its task can affect the states of other tasks in that random delays in a task's death may give it time to send more messages to other tasks on certain reexecutions. Furthermore, when a task "dies", it destroys all ports $pp$ it had receive rights on. If a task's death is premature, other tasks who had successfully sent messages to ports $pp$ will not send those same messages in time to be received by the dead task's threads. While the act of sending a message should not itself change the state of the sending task (since a message send operation can be considered a glorified "read"), since the sending thread will be informed of whether the sending operation was a success or failure, the thread may take a different path of execution depending on the success or failure of the send operation. Since different paths may be taken, each path may change the state of the task in a unique manner.

# Appendix C

# BNF for Database Activation

```
record  <global events>

record <specific global events>

record <specific local events>

<specific global events> ::=  <task-id><global-events> |
                              <thread-id><global-events> |
                              <task-id><global-sub-events> |
                              <thread-id><global-sub-events>

<specific local events> ::=  <task-id><local-events> |
                             <thread-id><local-events> |
                             <task-id><local-sub-events> |
                             <thread-id><local-sub-events>

<global-events> ::=   create-task | destroy-task |
                      create-thread | destroy-thread |
                      task-done |  task-aborted |
                      thread-done | thread-aborted |
                    port-creation |  port-destruction |
                    port-set-create | port-set-delete |
                    port-set-add-port | port-set-remove-port |
                    user-event-ended |  user-event-begun |
                    message-transmission | message-reception

<local-events> ::=   assignment-to-variables |
                     assignment-to-pointers |
                     reaching-a-label

<global-sub-events> ::=  <port-name>  port-creation |
                         <port-name>  port-destruction |
                         <port-set-name> message-reception |
                         <port-name>  message-reception |
                         <port-name>  message-transmission |
                         <port-name>  port-set-create |
                         <port-name>  port-set-delete |
                         <port-name>  port-set-add-port |
                         <port-name>  port-set-remove-port |
                         <user-event-name>  user-event-end |
                         <user-event-name>  user-event-begun

<local-sub-events> ::=  <variable-name> assignment-to-variables |
                        <pointer-name>  assignment-to-pointers |
                        <label-name>    reaching-a-label


<user-event-name> ::=  user-event1 | user-event2 | userevent3 | ...

<port-set-name> ::=  portset1 | portset2 | portset3 | ...
```

```
<port-name> ::=  port1 | port2 |  port3 | ...

<task-id> ::= task1 | task2 | task3 | ...

<thread-id> ::= thread1 | thread2 | thread3 | ...

<variable-name> ::=  var1 | var2 | var3 | ...

<pointer-name> ::=  ptr1 | ptr2 | ptr3 | ...

<label-name> ::= label1 | label2 | label3 | ...
```

# Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *Proceedings USENIX Summer Conference*, pages 93-112, Atlanta, GA, 1986.

[2] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avedis Tevanian, and Michael Wayne Young. *MACH Kernel Interface Manual.* Department of Computer Science, Carnegie-Mellon University, January 1990.

[3] Richard H. Carver and Kuo-Chung Tai. Reproducible testing of concurrent programs based on shared variables. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 428–433, 1986.

[4] Deborah Casewell and David Black. Implementing a Mach debugger for multithreaded applications. Technical Report CMU-CS-89-154, Carnegie-Mellon University, Department of Computer Science, November 1989.

[5] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comp. Sys.*, 3:63–75, February 1985.

[6] Eric C. Cooper and Richard P. Draves. *C Threads.* Department of Computer Science, Carnegie-Mellon University, July 1987.

[7] Bao Minh Dang. Methodology and tools for distributed debugging. M.Comp.Sci. Thesis, Concordia University, 1989.

[8] Saumya K. Debray, David Scott Warren, Suzanne Dietrich, and Fernando Pereira. *The SB-Prolog System, Version 2.5.* Department of Computer Science, University of Arizona, September 1988.

[9] Richard P. Draves, Michael B. Jones, and Mary R. Thompson. *MIG-The MACH Interface Generator.* Department of Computer Science, Carnegie-Mellon University, November 1989.

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

[11] C. J. Fidge. Reproducible tests in CSP. *Australian Computer Journal*, 19(2):92–98, May 1987.

[12] Alessandro Forin. Debugging of heterogeneous parallel systems. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):130–140, January 1989.

[13] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *Proceedings of Tenth International Conference on Distributed Computing Systems*, pages 134–141, Paris, France, May 1990.

[14] Haim Gaifman, Michael J. Maher, and Ehud Shapiro. Replay, recovery, replication, and snapshots of nondeterministic concurrent programs. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 241–255, 1991.

[15] Jason Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[16] C.A.R. Hoare. Communicating sequencial processes. *Commun. ACM*, 21(8):666—677, 1978.

[17] Avadis Tevanian Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach threads and the Unix kernel: The battle for control. Technical Report CMU-CS-87-149, Carnegie-Mellon University, School of Computer Science, August 1987.

[18] Dan Julin. Network server design. Technical report, Carnegie-Mellon University, Department of Computer Science, MACH Networking Group, August 1989.

[19] V. Krawczuk, H.F. Li, and T. Radhakrishnan. cdb: A toolkit for debugging distributed programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 242–244, Santa Cruz, California, May 1991. [Extended abstract].

[20] Victor Krawczuk. cdb record/replay facility and debugging tools: v 1.0.0. Technical report, Concordia University, Department of Computer Science, March 1991.

[21] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

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

[23] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. *Proceed'.gs of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):124–129, January 1989.

[24] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[25] Richard M. Stallman. *Emacs Versi^ 18 for Unix Users.* Free Software Foundation, Inc., October 1986.

[26] Richard M. Stallman. *GDB Manual, The GNU Source-Level Debugger.* Free Software Foundation, Inc., October 1989.

[27] Richard M. Stallman. *The C Preprocessor.* Free Software Foundation, Inc., July 1990.

[28] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., June 1991.

[29] R. E. Strom and S. A Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204-226, August 1985.

[30] Mary R. Thompson. Mach environment manager. Unpublished manuscript, Carnegie-Mellon University, School of Computer Science, July 1988.

[31] Jeffrey Tsai, Kwang-Ya Fang, Horng-Yuan Chen, and Yao-Dong Bi. A noninterference monitoring and replay machanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[32] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.