# Checkpointing and Rollback Recovery in a Non-FIFO Distributed Environment

Alain Jules Sarraf

A Thesis

in

The Department

of

Computer Science

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

June 1993

ISBN 0-315-87283-7

## ABSTRACT

Checkpointing and Rollback Recovery in a
Non-FIFO Distributed Environment

Alain Jules Sarraf

The saving of program states into checkpoints for subsequent rollback and recovery has applications in fault tolerance and distributed debugging. In distributed systems checkpoint creation and rollback are non-trivial due to the inherent characteristics of distributed systems such as the lack of global time and inter-dependencies among processes. Algorithms for checkpointing and rollback must ensure that checkpoints are created in such a way as to allow recovery to a consistent state of the distributed system. The conceptual model which has been assumed by many distributed checkpointing and rollback algorithms is based on point to point FIFO channels between communicating processes. This thesis is concerned primarily with one algorithm for checkpoint and rollback recovery referred to as RLV. In RLV, the dependency on FIFO-ness is used primarily for the detection of *pre-rollback* messages. i.e. messages which originated from the previous execution of a process and are no longer valid. In operating systems such as Mach which adopt a port-based communication model, the presence of multiple ports and multiple threads of control makes it impossible to guarantee FIFO ordering of messages between processes, or *tasks*, and therefore existing algorithms cannot fit well onto such a model. Based on the RLV algorithm, a modified algorithm called MRLV is developed which supports the rollback and recovery of tasks executing in Mach-like environments. It makes provisions for detecting pre-rollback messages without depending on FIFO channels. An architectural model, called "Body and Soul", which preserves resources across successive "incarnations" of a task, is designed to facilitate the implementation of MRLV in the Mach environment. The MRLV algorithm has been implemented in the context of a general purpose X-window based distributed debugger called XCDB and the implementation details are presented.

# Dedication

This thesis is dedicated

to

My Parents

Raouf

and

Marie

# Acknowledgement

Over the past two years, my life has revolved around the work presented in this thesis. During this time there have been moments of desperation, frustration, and also great inspiration. I would like to take this opportunity to thank the many people who have helped me get through the tough times and who have made this thesis possible.

First and foremost, I would like to thank my supervisor Dr. T. Radhakrishnan for both his academic and financial support. He is the person who initially convinced me that doing a Master's degree would be a worthwhile endeavour. Although I was hesitant at first, in retrospect I realize that this was a wise move. His down-to-earth approach to solving complex problems made our brain-storming sessions very inspiring and I am very thankful to have had the opportunity to work with him on such an interesting problem. I consider Dr. Radhakrishnan to be much more than just a supervisor. He is a good friend and father figure who cares very deeply about the welfare of his students. I am very thankful for the wise advice and new perspectives he has given me during this time on both scientific, and personal levels. Thanks to him, I have learned much more than can be found in the pages of any text book or technical paper.

During my time in the research lab, I have seen many people come and go but there are a few who deserve special mention. I met Christy Yep at Concordia during my undergraduate years where we became good friends. Christy is the person who has been most influential in the development of my work since we collaborated very closely in the debugging project. We shared both our disappointments and triumphs and when it all was over we had forged a lasting friendship. His sense of humor and late night conversations were always a welcomed break in the monotony. I introduced him to the blues , and he made me laugh in the Paris metro; a fair exchange. I also want to thank Juliette D'Almeida who I could always count on for being a good friend. She is one of the nicest people I have ever met. Although her printer jamming

and phone calls/visits from unexpected sources were a bit of a humorous nuisance, she always managed to make life in the lab eclectic and fun. Her presence was deeply missed when she left for bigger and better things.

I want to thank all of the system analysts at Concordia who helped me overcome problems with the "deathnet" and the dreaded `netmsgserver`. In particular Michael Assels, Paul Gill, and especially Francois Lambert who kept me from being bored when I became the proverbial "Last of the Mohicans" in the lab. I truly appreciate his presence and good sense of humor.

To the best rock n' roll band around, The Rhythm Kings, I want to send out my deepest gratitude for making weekends away from my work infinitely more enjoyable than reading boring technical journals. In particular, I want to thank Dino DeLuca, Peter Pugliese, and David Julien who have not only been my best bandmates, but also my closest friends for the past twelve years. We have been through alot together and I would be very surprised to meet anyone with as much good humor, talent, and kind-heartedness as these three characters. They are like brothers to me and I deeply appreciate their support.

Also, Sandra Bode who I met in Munich a few years ago deserves acknowledgement. Her letters, and our time together gave me the motivation to work harder, and our many treks across Europe have been an unforgettable experience which I will always remember when I think back on my years as a graduate student.

Last, but certainly not least, I want to thank my family: Mom, Dad, Nona ,and Chris, for being my true support system. Thank you Mom and Dad for putting up with my grouchiness after the numerous "bad days" and for always being there with your love for me. My Nona, whose beautiful smile and charm cheered me up every single morning, has been a true inspiration. I love them all very deeply and am eternally grateful for everything they have done for me.

# Contents

# List of Figures

# Chapter 1

# Introduction

*Dear Sir or Madam, will you read my book?*
*It took me years to write, will you take a look?*
*- John Lennon and Paul McCartney*

In recent years the demand for increased processing power has led computer scientists into new directions of computing. One such direction, is the area of distributed computing. Distributed systems achieve increases in throughput by means of distributing the load of a computation among several processes which run on independent processors. Although an increase in processing power is gained by doing so, the inherent characteristics of such a system introduce new problems which are not present in a traditional single-process computation. These problems render distributed systems more difficult to understand, than traditional systems and therefore, methods of solving them are needed in order to make effective use of a distributed system's power.

### Distributed computation vs. Sequential computation

Whereas "traditional" computation involves a single process executing a single task, a distributed computation involves a set of processes working together to accomplish a possibly large task. Each process involved is able to execute concurrently with the other processes, occasionally synchronizing its execution with that of the others. This synchronization takes the form of passing messages between processes via some communication network. Messages are sent between processes along communication

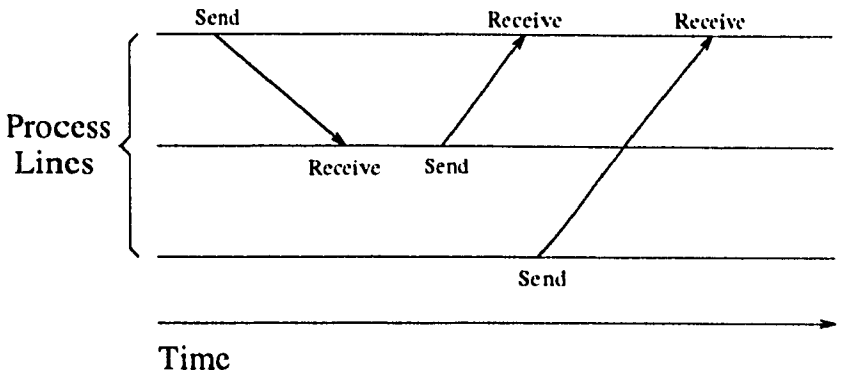


Figure 1.1: A space-time diagram representing a distributed computation

links called *channels*. An example of the use of message passing is to enforce mutual exclusion. Messages can be used to synchronize the execution of tasks ensuring that no two processes enter their critical regions at the same time.

The distributed program executions in this thesis are represented by space-time diagrams. In a space-time diagram, process executions are depicted by a line along the time axis and messages are depicted by directed arrows between process lines (see Figure 1.1). The diagram is a two-dimensional representation of process execution in a distributed system. The X-axis represents time while the discrete Y-axis represents the processes. The source of a directed message line represents the sending event of the message from the originating process , and its destination represents the receiving event at the destination process.

## 1.1 Introduction to Checkpointing

The ability to restart the execution of a program is very important for program debugging and for applications which depend on the progress of the system in the presence of failures. Re-execution from the start of a program is generally straightforward However, sometimes we would like to restart the execution from a pre-defined point in the program , rather than from the beginning. this pre-defined point is called a *checkpoint*.

**Definition 1** *A* **checkpoint** *is the saved state of a single process stored in a form such that the process can restart its execution from the point in time when the checkpoint was created.*

**Definition 2 Checkpointing** *is the process of saving process states into checkpoints.*

Checkpoints contain any information needed to restart the process in which the checkpoint was created. When the process is restarted, its current state is discarded and the state saved in the checkpoint is restored.

**Definition 3** *The restoring of the state of one or more processes to their state previously stored in a checkpoint is called* **Rollback**.

Systems which incorporate checkpoints and rollback are called *checkpoint and rollback recovery systems* since they *recover* a previous state of a system.

### 1.1.1 Checkpointing of independent processes

Checkpointing of processes which do not interact is relatively straightforward. A process wishing to checkpoint its state simply blocks while saving its state to a secondary storage. Since there is no interaction between processes, the blocking only affects the process which is creating the checkpoint. A process does not need to worry about the state of other processes at any time.

When an independent process wishes to rollback, it simply discards its current state and reinstates the state which was previously stored in one of its checkpoints. The rollback of a process has no effect whatsoever on the state of other processes in the system since they never communicate with each other.

### 1.1.2 Checkpointing of interacting processes

**Definition 4** *An* **interacting set** *of processes is defined as those processes which have communicated with each other either directly, or indirectly since their last checkpoints were created.*

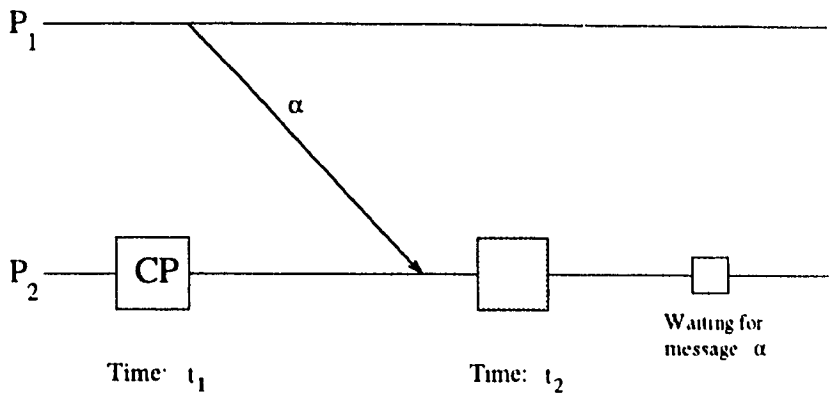


Figure 1.2: Inconsistent rollback of interacting processes

When processes interact with each other in a distributed system, checkpointing becomes more involved . Since any given process is now dependent on its interaction with other processes, the interaction may affect part of the process state which must be saved in the checkpoint. Therefore, when a process wishes to checkpoint its state, it must be sure that if it decides to rollback to this checkpoint at a later time, it must be in a *consistent* state with respect to the other processes in the system. Consistency upon rollback is ensured by guaranteeing that any dependencies on messages are reflected in the checkpoint along with the actual state of the process. In other words, a checkpoint in a distributed system with interacting processes consists of not only the process state but also the *channel* state of the *interacting set.*

Similarly, the rollback of a single process may be affected by the execution of other processes in the distributed system. The rollback to a checkpoint will "undo" any communication which may have occurred since the checkpoint was created. Consider the execution of two interacting processes shown in Figure 1.2. Suppose that the process $P_2$ creates a checkpoint by saving its state to disk at time $t_1$ and afterwards receives a message $\alpha$ from $P_1$. At time $t_2$, $P_2$ performs a rollback to the previously saved checkpoint and repeats the execution starting from that state. $P_2$ will be waiting for message $\alpha$ which does not arrive if $P_1$ has *not* rolled back its state to a point in its execution prior to the sending of $\alpha$. Therefore, $\alpha$ is a lost message.

An equally serious consequence of interaction on checkpointing is depicted in Fig-

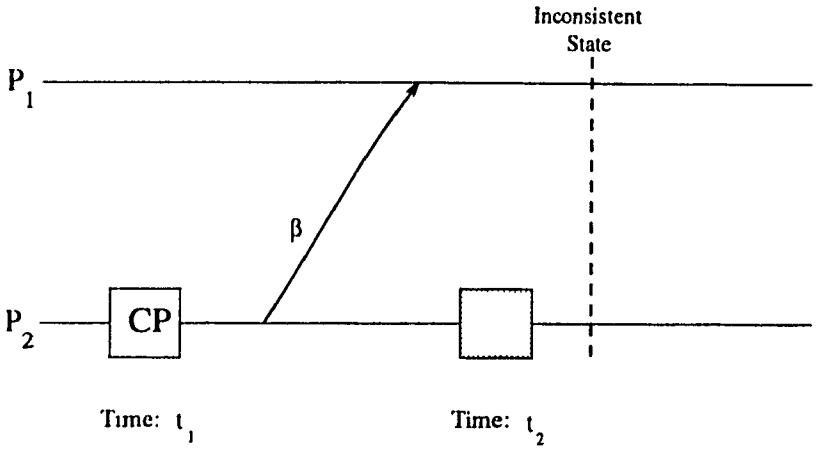


Figure 1.3: Inconsistent state due to incomplete rollback

ure 1.3. When $P_2$ rolls back its state to the checkpoint $CP$, it is in an inconsistent state since at that time, $P_1$ is in a state where it has already received message $\beta$ but $P_2$ is in a state where it has not yet sent $\beta$. Upon continuing execution, message $\beta$ will be sent again by $P_2$. Therefore $\beta$ will be a duplicated message.

The communication between interacting sets of processes produces such *message dependencies* which must be dealt with in order for rollback to a correct and valid system state to be possible. Hence, processes must be rolled back in such a way as to insure that any two processes are in a *consistent* state with respect to each other. i.e. that they agree on which messages have been sent and which ones have not [CHAN85].

**Definition 5** *A set of checkpoints (exactly one checkpoint per process in the interacting set) which forms a global consistent state of a distributed system is called a* **Recovery Line**. *The term global consistent state implies that if a receive event is included , then its corresponding send event is also included.*

All events that are part of the global state formed by the recovery line are said to be *reflected* in the recovery line. Recovery lines are represented by lines joining checkpoints on the space time diagram. Figure 1.4 depicts the execution of four processes. In this figure, two distinct recovery lines are depicted which represent valid

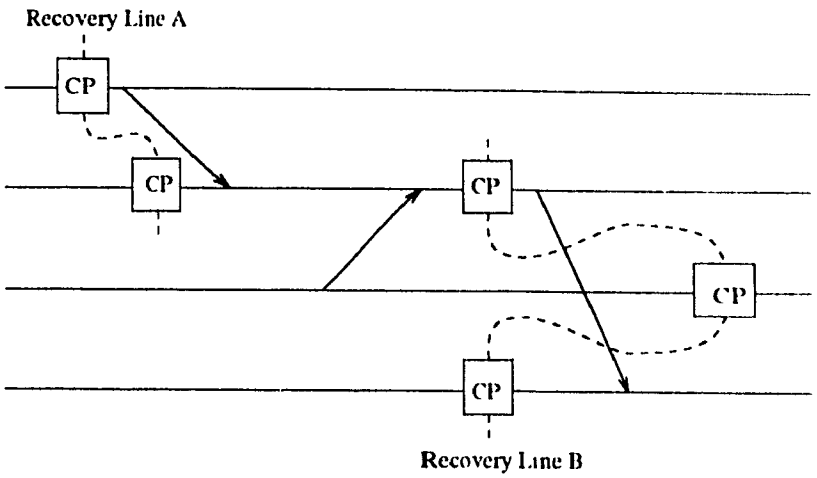


Figure 1.4: Representation of recovery lines on a space time diagram.

states to which the processes owning checkpoints on the recovery line may collectively rollback at a later time.

## 1.2 Software Fault tolerance and Checkpointing

If an error occurs in a distributed computation, very often we would like to correct the effects of the error and continue processing as if the error had never occurred. This is not a trivial task and involves careful checkpointing, rollback and recovery algorithms. As mentioned earlier, checkpointing is needed in many distributed applications. Two applications in particular are fault tolerance discussed in this section, and distributed debugging discussed in the next section.

Software fault tolerance deals with strategies for dealing with design faults in software which may cause errors in a program's execution [WOOD85]. In those cases, checkpointing can be used to provide a method of restarting an application from a point in time prior to the occurrence of a fault thereby rendering the program more resilient to faults.

An example of a software fault tolerance application which makes use of checkpointing is in distributed database systems [SOAG85]. Checkpointing can be used to save database states for recovery of data after failures.

One approach to achieving software fault tolerance is the recovery block scheme originally introduced by Horning in [HORN74] and discussed further in [KELL91]. It is based on collections of code components called *recovery blocks*. For a recovery block, the programmer specifies a number of possible versions of code to achieve the same computing action of that recovery block. Before entry into a recovery block, a *recovery point* which contains the program's state information is established. Upon entry into the recovery block, a primary version is executed and upon completion, an *acceptance test* is performed to determine whether or not the result is correct. If it is deemed incorrect by the acceptance test, the effects of the version's execution are undone and an alternate version specified for that recovery block is executed. This is repeated until the execution is found correct, or all the alternate versions are exhausted.

Although the recovery block approach is completely transparent to the programmer/designer of the system, its underlying mechanism is quite complex. In this case, a backward error recovery mechanism is used to recover from faults. The basic architecture of this mechanism is depicted in Figure 1.5. The recovery cache contains the checkpoints to be used in the rollback mechanism.

Checkpointing , rollback and recovery algorithms used for this approach to fault tolerance must be well designed to minimize the overhead of recovery from faults. Overhead must be minimized since applications requiring such fault tolerance tend to be real-time systems with critical time constraints and they require a guaranteed rate of progress.

Recovery blocks are intended for single-process environments. That is , the different versions specified are for the same program. However, the concept can be extended to a distributed environment. i.e. A recovery block scheme which supports several *interacting* processes.

The concept of *conversations* [GREG85, RAND75] is an extension of the recovery block scheme for this purpose. In this scheme, individual recovery blocks can be looked upon as a single process whose recovery block is treated as a black box. In effect, it is a nesting of recovery blocks between several interacting processes. Processes

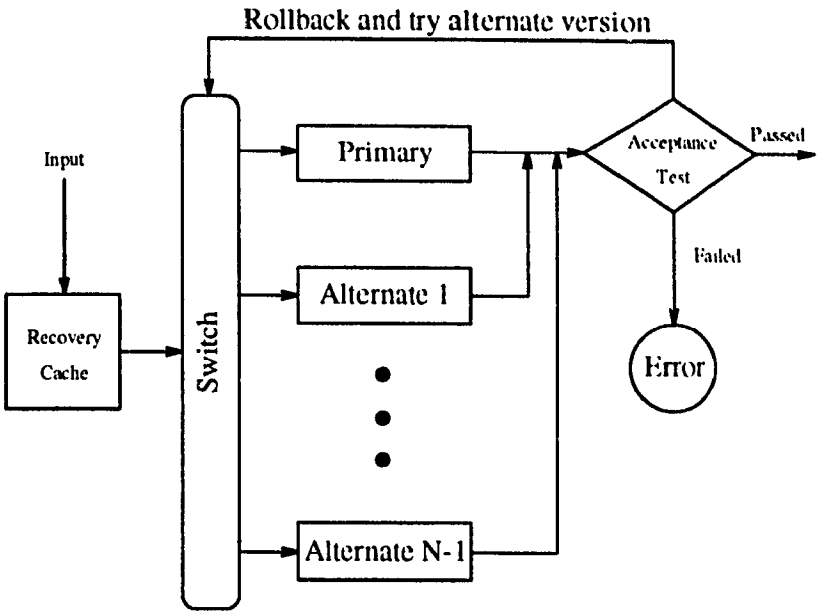


Figure 1.5: Recovery Block Architecture

enter a conversation at different times, separately establishing checkpoints prior to entrance. While in a conversation, processes can interact with each other freely but not with processes which are not part of the conversation. If an error is detected, all processes which form the conversation can collectively rollback their execution and restart the conversation each using an alternate module. Conversations can be nested thereby allowing several levels of recovery within the system.

### 1.2.1 Fault Tolerant checkpointing scheme requirements

Checkpointing schemes which are geared towards applications to fault tolerance must meet the following requirements [FRAZ89]:

- **No single point of failure**: The scheme must be distributed in order to avoid deadlock and other anomalies in the event of failure of a single entity.
- **N-fault tolerant**: The scheme must be able to support the concurrent failure of multiple tasks at any time and still be able to recover computation.

- **Minimal restrictions on Application**: The checkpointing scheme should not dictate the behaviour of the application programs.
- **Suitability for a large class of applications**: The scheme should be efficient for both communication and computation intensive applications.
- **Low overhead**: The checkpointing scheme should minimize the time lost during checkpointing, disk space for storing checkpoints, memory usage, processing during recovery, computation time lost due to rollback etc.

## 1.3 Distributed Debugging

Another application of checkpointing and rollback recovery is in the area of *distributed debugging*. It should be noted that the context of the work presented in this thesis is in the area of distributed debugging rather than fault tolerance and therefore more emphasis is placed on debugging.

Traditional debuggers which have been developed for the sequential world of uniprocessors have proven inadequate for the domain of distributed computing. In sequential debuggers, programs are monitored for the occurrence of events which are of interest to the user. This monitoring enables the user to better understand the behaviour of the program being debugged. Sequential programs are relatively easy to debug due to the fact that information is centralized. In distributed debuggers, such information must be gathered by the system before it can be presented.

A distributed debugger is a tool which aids programmers in detecting errors in their distributed programs. One distributed debugging model is the breakpoint-based debugger. As in sequential debuggers, processes are halted when a user-defined breakpoint is reached. A programmer makes use of a distributed debugger's breakpoint facility by specifying predicates composed of events which may or may not occur in the system. For example, a predicate can consist of message send or receive events. When the debugger detects the occurrence of events which render the predicate true, it halts the system in a consistent state and informs the user of its detection. At

this point, the programmer can view states of the different processes. This type of monitoring of events and halting in the distributed system is very important for distributed debugging.

Before discussing the importance of checkpointing and rollback within the context of distributed debuggers, the inherent characteristics of a distributed environment which make debugging difficult are discussed.

### 1.3.1 Characteristics of a Distributed environment

Debugging a distributed program is complicated by the following characteristics of the environment [JLSU87, MCDO89, MOGE84, CHWO89]

1. Multiple threads of control
2. Non-determinism
3. Communication Delay
4. Probe effect
5. Lack of precise global time
6. Difficulty of meaningful User Interface

#### Multiple threads of control

Unlike a traditional program, a distributed program will have many foci, or *threads* of control in the form of multiple asynchronous processes. This adds a new dimension of complexity which is not present in traditional programs. Specifically:

- Because of the many threads of control, a program's execution is much more difficult to follow. Furthermore, even if it can be followed, the execution is very difficult to understand.

- Distributed programs are prone to new types of bugs which are not present in a traditional computation. These bugs are usually caused by race conditions and synchronization errors, neither of which can ever arise in a sequential computation.

### Non-determinism

The synchronization between processes involved in a distributed computation relies on message passing. Message passing may introduce non-determinism in the execution of distributed programs. The results of a computation depend on both the system inputs, and the relative speed of the processes imposed by the ordering of messages during a computation. Because this ordering may not be unique, the behaviour observed in a system may or may not be reproducible. The (non)anomalous behavior of a program exhibited in one execution of the system may or may not be exhibited in a subsequent execution due to changes in the ordering of messages. This non-determinism adds to the complexity of understanding the behavior of a distributed program, and collecting information about it. The debugging strategies which have been proposed to solve the non-determinism problem, have generally taken one of two basic approaches

- **Live Detection**: In the case of live detection (eg. [SPEZ88]), events are detected "on the fly". That is, during a single execution of the system. Live detection solutions have eliminated the non- determinism problem by looking at only one particular execution of the system. Therefore, there is no need to worry about the different ordering of events which may occur upon subsequent re-executions of the program being debugged. However, one anomaly of this method is that the erroneous event may or may not occur in that particular execution. Therefore, this method is only suitable for recurring bugs.

- **Record and Replay** Record and replay methods deal with recording of system behavior and replay of the execution by means of event histories. These event histories contain all necessary information needed to deterministically replay an

execution [MCDO89, LECR87]. In order for these types of monitoring strategies to be effective , they must force a determinism in the computation so that upon repeated executions of the program, the same behavior will be displayed. LeBlanc's *instant replay* approach [LECR87] uses a central monitor to reroute messages according to previously recorded event histories.

### Communication Delay

The problem of communication delay is closely related to the non-determinism problem. There is an inherent delay in any distributed system between the time that a message is sent, and the time at which it is received. Communication delay increases the complexity of ascertaining the state of a process at any given time. Since the state of a process cannot be known instantly at any given time, valuable state information may be lost. This can happen if the process state has changed between the time at which the state information was requested, and the time at which the message requesting this information is received at the process. Therefore, any debugging strategy must take into account this delay when gathering state information about the processes in the system for predicate detection.

### Probe Effect

Any attempt to monitor the behavior of a distributed system will actually alter the behavior of the system. This is referred to as the probe effect [MCDO89]. Debugging strategies must minimize the interference that they introduce in the normal behavior of the system. Some means of avoiding interference during debugging is needed. There are three ways to avoid the probe effect in distributed debugging.

- **Debugging as an ongoing process:** One way to avoid the probe effect is to view debugging as an ongoing process. In this scheme, the monitoring process is made an integral part of the system which cannot be "switched off". This approach has been investigated by Chang and Wong [CHWO89] and independently by Haban [HAWE88]. The basic approach is to integrate monitoring

hardware into the system which is constantly activated. This way, the monitoring process *is* part of the normal execution rather than an interfering process executing alongside the "normal" execution of the system. The problem with this approach is obviously the overhead imposed by the extra hardware . The system would be built specifically with monitoring in mind thereby slowing down processes for which we have no interest in monitoring.

- **Separate communication networks:** The interference is caused by the messages introduced into the communication network by the debugging system. Haban [HAWE88] has avoided this situation by implementing a separate communication network for debugging messages alone. This method will almost completely eliminate the probe effect, however it is a very expensive solution, and its feasibility for large systems is questionable.

- **Static Analysis:** Static Analysis entails defining program states which generally indicate the occurrence of an error, and statically ensuring that the program cannot enter them. Unlike proof of correctness, there is no specification of program behavior used . The problem with this approach is that it can only be used to detect synchronization and data usage errors [MCDO89]. Furthermore, it is difficult to use for small distributed systems and almost impossible to use for very large ones. It also requires the user of the debugger to anticipate errors.

### Lack of Precise Global Time

Events occurring in a traditional sequential program will always be totally ordered by physical time. Unfortunately ,this is not the case in a distributed system and in order to be able to debug programs in such a system , some method of artificially ordering events in different processes is needed.

The events can either be totally ordered or partially ordered. A total ordering is much easier to understand, and can be useful in solving synchronization problems in a distributed system [LAMP78]. However, it implies an ordering between any two events in the system even if they are totally unrelated. Partial orders, on the other

hand, retain the notion of concurrency inherent to distributed systems. They are more useful for monitoring since they reflect the true behavior of distributed systems more accurately than total orders [MCDO89].

### Meaningful User Interfaces

Traditional debuggers display their information in a sequential order. This is meaningful in the context of a sequential computation where all events are totally ordered. However, in a distributed system with no notion of global time, such sequential displays are meaningless since events may only be partially ordered. Also, debugging information is difficult to represent in large scale[1] distributed systems because of the large amount of information which must be processed by the user. Therefore, the user interface must have an ability to abstract this information by either "classifying it", or "collapsing details". The interface should be built on a sound basis which will allow scaleability of the system. Methods for classification of events and collapsing details in an interactive distributed debugger is still an open problem. Three main techniques of representing debugging information which have been presented in the literature are: [MCDO89].

- **Textual representations**: Textual representations of program behaviour usually display program control information and rely on coloring, highlighting, and nesting to emphasize interesting events. this type of information does not give an overall view of the system's communication patterns and therefore, errors which are caused by erroneous patterns, are difficult to detect. This is a serious deficiency of the approach since ,as mentioned previously, distributed programs are prone to synchronization and race condition bugs which involve several processes.

- **Space-Time diagrams**: Space-Time diagrams are useful for displaying these interactions meaningfully however, the large numbers of processes and messages

[1] involving hundreds of processes

tend to clutter the display making it very difficult for users to debug their programs. This method is inappropriate for scaleability.

- **Animation**: Space-Time diagrams provide a good means of viewing the program behaviour over a period of time but they do not give the user a way of observing the passage of time i.e. observing behaviour as it happens. Animation can be used to present the changes occurring in the system as they occur. Animation-based distributed debuggers help the user to find errors by allowing them to notice changes from one frame of the animation to another. The program behaviour is animated by placing objects on a two-dimensional display which represent entities of the system. For example, boxes may represent processes and arrows may represent communication channels.

### 1.3.2 Checkpointing in Distributed Debuggers

Debugging a distributed program is far from being a trivial task. As in traditional programs, it involves repeated examination of states during execution of the program [MCDO89]. However, unlike traditional programs, the instantaneous examination of such states is not possible due to the communication delay involved.

Checkpointing is used in distributed debuggers to *save* process states during the execution of a program for examination upon subsequent replay. This allows a *cyclical* approach to debugging where checkpoints are chosen by the user as a starting point for re-execution. By doing so, the program need not be restarted from the beginning of its execution, and the search space for locating the bug can be successively narrowed.

A debugger which provides a checkpoint and rollback recovery mechanism can aid the user in debugging programs in the following ways:

- **Detecting race conditions**: Race conditions are very common in distributed programs and are caused by non-determinism. Allowing the user the ability to rollback and re-execute a program from the same point in the code several times can aid the user in detecting discrepancies in the message patterns in the subsequent executions. This can aid in detecting race conditions in the

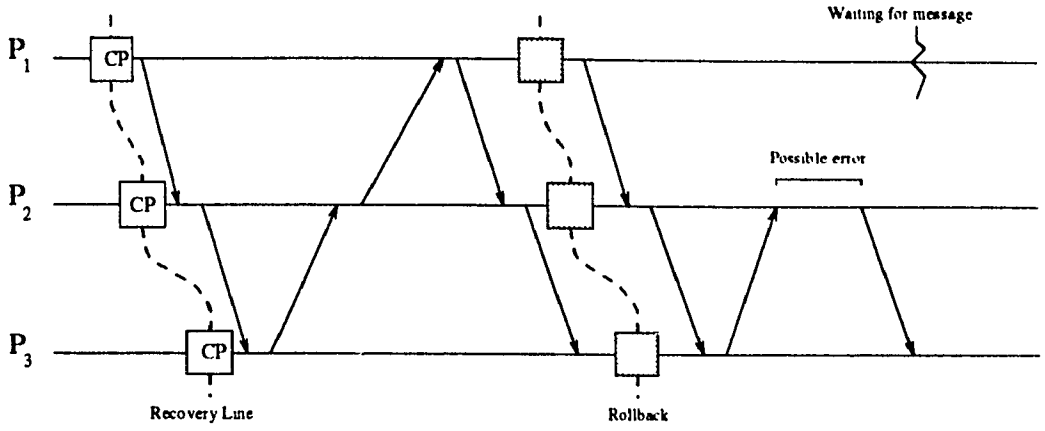


Figure 1.6: Detection of error by message pattern discrepancy.

program. For example, the ability to rollback execution can be used to detect a discrepancy in an expected message pattern such as that of a token ring program. In Figure 1.6 the expected token ring pattern is not repeated upon rollback. Process $P_2$ sends a message to process $P_3$ which should have been sent to $P_1$. This indicates a possible error in $P_2$.

- **Narrowing of program area**: Upon rolling back to a specific checkpoint, the user can specify new predicates to be detected by the debugger. By allowing the user to do so, the debugger can enable the user to narrow the area of code to be inspected for locating the bug.

### 1.3.3 Checkpointing Scheme Requirements for Distributed Debugging

Checkpointing schemes for distributed debugging do not have as strict requirements as those for fault tolerance. They must meet the following requirements:

- **Efficient Rollback**: In order to support a cyclical debugging approach, the debugger must be able to rollback often and efficiently.

- **Efficient Checkpointing**: In order to allow as many recovery points as possible, the checkpointing algorithm should be efficient.

- **Minimal restrictions on Application**: The checkpointing scheme should not dictate the behaviour of the application programs being debugged in any way.
- **Suitability for a large class of applications**: (same as in section 1.2.1)
- **Low overhead**: (same as in section 1.2.1)

Unlike fault tolerance schemes, single points of failure are not as important in distributed debuggers. Failure of the debugging system itself is not important since we are not concerned with progress of the system but rather with observing the behaviour of the application program. Also, since the user of the debugger is only concerned with the rollback of the system to a particular global state, there is no real need to support multiple concurrent rollbacks.

## 1.4 Classification of checkpointing schemes

Distributed checkpoint and rollback recovery algorithms can be classified into two categories according to the method used to create checkpoints with respect to recovery lines [PASS88, KOTO87]:

- **Unplanned:** processes create their checkpoints without regard as to when the other processes are creating theirs. At the time of rollback, it is the responsibility of the rollback and recovery algorithm to determine recovery lines (i.e. to select checkpoints such that they form a consistent recovery line.
- **Pre-planned:** processes creating checkpoints do so in a coordinated manner thereby forming objective and consistent recovery lines. The responsibility of which checkpoints are part of which recovery lines lies with the checkpointing algorithm rather than the rollback and recovery algorithm.

### 1.4.1 Message Logging (Unplanned)

In message logging schemes, messages which have been sent or received by a process are saved with the process state when a checkpoint is created. The checkpoint also

contains the ordering of message send and receive events. Saving the order of message events ensures that upon rollback, the repeated computation will be identical to that performed prior to the rollback. When a process rolls back, the saved messages are *replayed* to it until the message log is depleted. When the log is depleted, the rolled back process is in a consistent state with respect to the other non-failed processes.

### 1.4.2 Coordinated Checkpointing (Pre-planned)

Coordinated checkpointing can be further divided into the following sub-categories:

- **Global Coordinated Checkpointing**: Global checkpointing involves saving the entire state of the system in a global snapshot [TAMI84, FRAZ89]. During recovery, *all* nodes are rolled back to their state as it was at the time of the snapshot. These techniques incur a very high overhead for coordinating the checkpoint creation and saving the state of all nodes.

- **Process-level Coordinated Checkpointing**: Process-level checkpointing involves checkpointing of only a subset of interacting processes within the entire system. i.e. the interacting set.

We will deal only with process-level coordinated checkpointing here since global checkpointing techniques are generally only useful for batch applications such as large numerical computations [FRAZ89].

The idea behind coordinated checkpointing schemes is to ensure that any two process states ,or checkpoints, which have been saved to disk as part of the computation being checkpointed, are consistent with each other. Because of this, algorithms that use a coordinated checkpointing scheme are guaranteed to find recovery lines which will allow the computation to be rolled back to a valid consistent state.

### 1.4.3 Comparison of Schemes (Unplanned vs Pre-planned)

Pre-planned checkpointing schemes are always guaranteed to find recovery lines which are valid since the checkpoints are created in such a way that they are always part of some valid consistent state. In some unplanned checkpointing schemes on the other

hand it cannot be guaranteed that checkpoints form part of any recovery line since these recovery lines must be extrapolated at the time of rollback. Consequently, unplanned strategies which cannot guarantee consistent recovery lines, cannot guarantee that a rollback will not necessitate a re-execution of the entire system (this problem is discussed in more detail in section 1.5.5). This is an important consideration in the context of real-time systems which require a strict rate of progress.

The pre-planned strategies incur a higher overhead during the checkpointing phase since it is during the checkpointing phase that recovery lines are determined. On the other hand, unplanned strategies incur very little overhead during checkpointing but have a high overhead during the rollback phase. Pre-planned strategies are therefore better for applications which anticipate a large number of rollbacks such as debugging and unplanned strategies are better for applications which are not expected to require many rollbacks. Since more than one process may be involved in the creation of checkpoints in the pre-planned schemes, the pre-planned schemes tend to disrupt the system more than in the unplanned strategies where only a single task is involved in creating its own checkpoint.

Since message logs must be kept on disk, the unplanned strategies are not well suited for communication intensive applications. Overhead incurred by the system is directly proportional to the amount of communication involved. Pre-planned strategies, on the other hand, are not required to keep message logs and therefore are well suited for both computation intensive and communication intensive applications.

The biggest advantage that pre-planned strategies have over the unplanned ones is that they can be used for recovery of non-deterministic applications since they are not based on the replaying of logged messages.. The premise behind message logging, is that a process can be recovered by *replaying* messages to it after a rollback has occurred exactly as they appeared during the normal execution. However, in non-deterministic applications, the message ordering may not be guaranteed, so messages may not arrive in the same order as they did during the initial execution.

## 1.5 Problems related to Checkpointing

The inherent characteristics of distributed systems make any operation which requires global state information very difficult. Checkpointing, and rollback recovery are no exception to this rule. This section outlines some problems which make recovering the state of a system very difficult.

### 1.5.1 Lack of global time

As mentioned earlier ,distributed systems suffer from the problem of having no global time scale. Events occurring in a traditional sequential program will always be totally ordered by physical time. Unfortunately ,this is not the case in a distributed system. Therefore , some method of artificially ordering events in different processes is needed in order to discuss the creation of meaningful checkpoints. The relative ordering of events in the system must provide a meaningful notion of *order of occurrence*,and *concurrency.*

In this thesis, and in the checkpoint and rollback recovery algorithms , the notion of *happens before* is as described in [LAMP78] and outlined briefly here.

#### Lamport's happens before relation

The *happens before* relation orders events with respect to other observable events occurring in the system. By doing so, a partial order is imposed on all of the events as follows:

- If $A.a$ and $A.b$ are both events in Process $A$, and $A.a$ occurs before $A.b$ then $A.a$ *happens before* $A.b$ ($A.a \rightarrow A.b$)
- If $A.a$ is an event in process $A$ and $B.b$ is an event in process $B$, and $A.a$ is the send event of a message from $A$, and $B.b$ is the corresponding message receive event in $B$ then $A.a$ *happens before* $B.b$ ($A.a \rightarrow B.b$)
- Consider three events a,b,and c: If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

- Two distinct events $a$ and $b$ are considered *concurrent* If $a \not\rightarrow b$ and $b \not\rightarrow a$. $(a \parallel b)$

### 1.5.2 Communication delay

If the state of a single-process program is checkpointed, it can be done so instantaneously. There is no need to coordinate its checkpoint creation with any other processes since there is no outside interaction. However, in multiple-process applications the fact that there is no global time scale, implies that it is impossible for system wide states to be saved instantaneously. Therefore, in order to save states which are consistent across several processes, some communication protocol must be employed. However, in distributed systems, there is an inherent delay between the time that a message is sent and the time at which it is received at some remote process. Since Checkpoints cannot be created according to a global clock, and messages incur a delay, it is difficult to save and restore states of several processes instantaneously.

### 1.5.3 How often should checkpoints be created?

How often checkpoints should be created really depends on the application. Two things to consider when deciding on the checkpoint frequency are the need to minimize the amount of computation to be rolled back, and the overhead of the actual checkpointing operation. If checkpoints are taken often, the system performance will degrade but recovery time will be decreased. On the other hand, if checkpoints are taken less often, system performance will increase but a penalty will be incurred at the time of rollback and recovery. The designers of checkpoint and rollback recovery algorithms must weigh the advantages and disadvantages of checkpoint frequency on the basis of the likelihood of rollback occurrence. This can be considered as an optimization problem as done in [CHKR88].

### 1.5.4 Pre-rollback messages

The communication delay inherent in distributed systems introduces another obstacle to rollback and recovery. When a process has rolled back its state, some messages

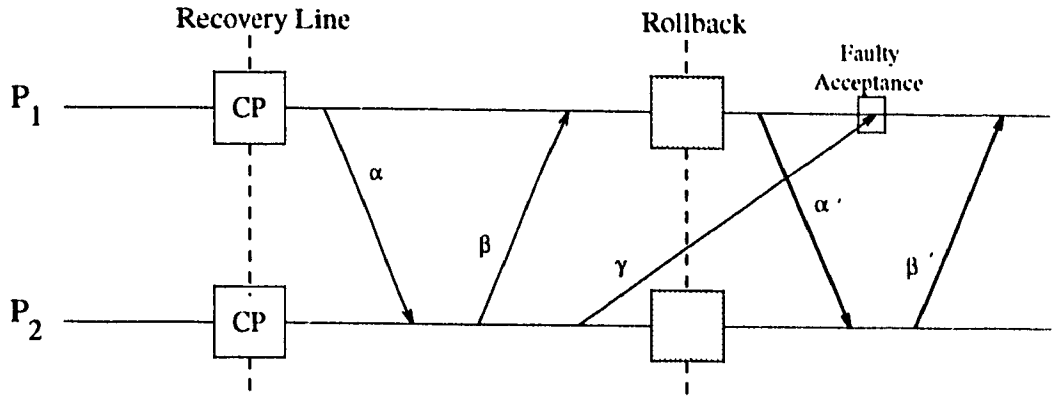


Figure 1.7: Faulty acceptance of a pre-rollback message

which it has sent prior to its rollback may still be in transit (i.e. the messages may not have been received at their destination yet). These messages would no longer be valid since they were sent before the sender had changed its state.

**Definition 6** *Messages which are sent by a process prior to a rollback but are received after that rollback is complete are called* **pre-rollback** *messages.*

It is important for checkpoint and rollback recovery algorithms to make sure that these pre-rollback messages are somehow *flushed out* of the system so that they are not mistaken for valid message re-sends upon rollback. Consider the system execution in Figure 1.7. After the recovery line is created, messages are sent between processes $P_1$ and $P_2$. After the rollback, $P_1$ resends message $\alpha$ as $\alpha'$ and $P_2$ resends message $\beta$ as $\beta'$. However $\gamma$ which was sent from $P_2$ was in transit while the rollback was taking place and arrives at $P_1$ *after* the rollback. $P_1$ mistakenly accepts message $\gamma$ , thinking that it is actually receiving $\beta$. Since it was not detected that $\gamma$ is a pre-rollback message, the whole system execution is now in error.

### 1.5.5 The Domino Effect

In distributed systems, processes are constantly communicating with each other through messages. Because of this, an error which occurs in one process may contaminate the state of another process through the sending of incorrect information in

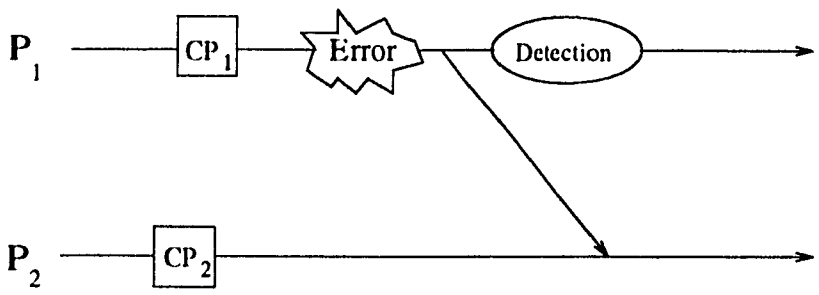


Figure 1.8: Rollback of 2 processes due to contamination

a message. Upon detection of an error, a process' state is rolled back to a previously saved checkpoint whose state is known to be correct. However, this is not sufficient to correct the *global* system state. In order to do so, all processes which may have communicated with the erroneous process must also be rolled back to a point in their execution before (according to [LAMP78] see section 1.5.1) any direct or indirect communication with that process occurred. For example, consider the two processes depicted in Figure 1.8. An error occurs in $P_1$ but before it is detected, $P_1$ sends a message to $P_2$ thereby contaminating it. Once $P_1$ detects the error, it attempts to rollback to checkpoint $CP_1$. However, this is not enough to undo the effects of the error. In order to correct the state of the system, $P_2$ must also be rolled back to a state before any communication with the contaminated process occurred. Thus, $P_2$ must also be rolled back to the state which has been saved in checkpoint $CP_2$.

One of the biggest problems in checkpointing and rollback recovery is the well known *domino effect*[VENK87]. The domino effect refers to the avalanche roll back of processes to their initial state due to rollback of one process. It is caused by the type of contamination described above. Consider the three processes in Figure 1.9. $P_1$ detects an error which causes it to rollback to its most recently saved checkpoint $CP_2$. However , between the time that the error occurred and the time when it was detected, a message $\setminus$ was sent from $P_2$ to $P_1$. When $P_1$ recovers it will expect the receipt of message $\setminus$. One way to achieve this is to have $P_2$ *resend* the message.

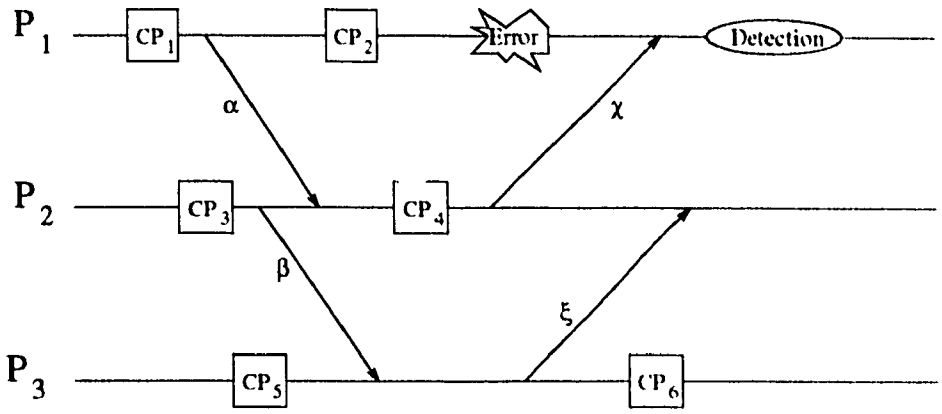


Figure 1.9: The domino effect

Therefore $P_2$ must be rolled back to checkpoint $CP_4$. By the same reasoning,

- message $\xi$ from $P_3$ to $P_2$ must be resent, forcing $P_3$ to rollback to $CP_5$
- message $\beta$ from $P_2$ to $P_3$ must be resent, forcing $P_2$ to rollback to $CP_3$
- message $\alpha$ from $P_1$ to $P_2$ must be resent, forcing $P_1$ to rollback to $CP_1$

This can conceivably continue until the whole computation must be restarted in order to recover from the fault. This is completely unacceptable in a system with real-time constraints [KOTO87] and negates the advantage of performing backward error recovery in the first place.

### What causes the domino effect?

Before discussing the cause of the domino effect, some definitions are needed. These definitions are introduced in [CHAN85] and repeated here briefly. The *local state* of a process $P$ is defined by $P$'s initial state and all events which have occurred in $P$ since that state. The *channel state* is the set of all messages which have been sent but not yet received at a particular point in "instantaneous" time (i.e. real physical time as opposed to the relative time discussed in [LAMP78] see section 1.5.1). The *global state* of a distributed system includes the local state of all processes and the state of all channels in the system.

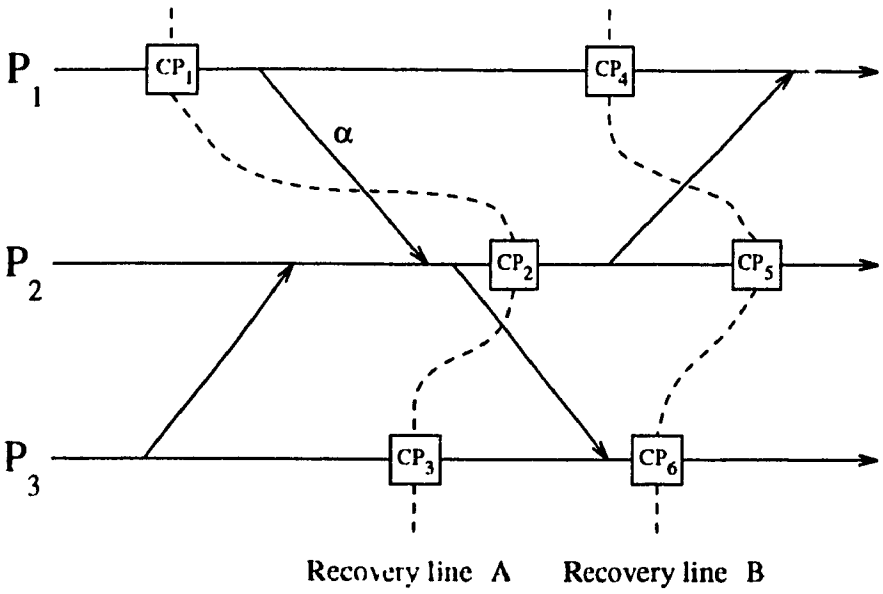


Figure 1.10: Inconsistent and consistent recovery lines

The random generation of checkpoints forming potentially *inconsistent global states* [CHAN85] creates a trap for the domino effect. An inconsistent global state or *cut* is one which contains unaccounted for information. Specifically, it may contain the knowledge of the receipt of a message without that of the corresponding message send. For example, in Figure 1.10 global states are demarcated by the *recovery lines* running through the checkpoints in processes $P_1$, $P_2$, and $P_3$. These recovery lines represent the set of checkpoints which are involved in the rollback and recovery of a correct system state after an error has been detected. It can be seen that the global state demarcated by recovery line $A$ is inconsistent since it contains the receipt of message $\alpha$ by $P_2$ without the corresponding send by $P_1$. Conversely, recovery line $B$ is *consistent* since it does not include any message receive event without the corresponding send event.

The unplanned strategies tend to lead to a domino effect since it cannot be guaranteed that checkpoints will form consistent recovery lines.

### Correct Rollback of global system state

In order to avoid the domino effect , checkpoint creation must be coordinated such that the checkpoints form consistent recovery lines. A planned checkpoint and rollback recovery strategy could guarantee a domino-free rollback. [VENK87, PASS88, VENK88] outline event dependency sets which are used in the determination of domino-free recovery lines (note that for the purposes of this , an event refers to a message send or receive, or a checkpoint creation). Those definitions are restated below:

- **Error dependent set (EDS):** The set EDS($E$) contains the earliest event (if any) in every process which may be affected by error $E$ occurring in the system either directly or through some direct or indirect communication. Any recovery algorithm would obviously have to choose checkpoints which precede events in EDS($E$) upon rollback.

- **Backward dependent set (BDS):** Suppose there are two processes $P_i$ and $P_j$ which contain checkpoints $CP_{ia}$ and $CP_{jb}$ respectively (Let the notation $CP_{ia}$ stand for the *ath* checkpoint created by process $P_i$). Suppose that checkpoints $CP_{ia}$ and $CP_{jb}$ have been chosen for rollback. The set BDS($CP_{ia}$,$CP_{jb}$) is defined as the set of all send events in $P_j$ which precede the creation of $CP_{jb}$ and are received in $P_i$ after the creation of $CP_{ia}$. For example, the send events of messages $\xi$ and $\chi$ from $P_j$ form the set BDS($CP_{ia}$,$CP_{jb}$) in Figure 1.11.

- **Retransmission dependent set (RDS):** Suppose that $P_i$,$P_j$,$C_{ia}$,$CP_{jb}$ are defined as in the BDS set above. The RDS($CP_{ia}$,$CP_{jb}$) is the set of receive events in $P_j$ occurring *before* the creation of $CP_{jb}$ whose messages were sent *after* the creation of $CP_{ia}$ in $P_i$. The RDS in Figure 1.11 includes the receive events of messages $\alpha$ and $\beta$ in $P_j$.

Any recovery algorithm would obviously have to choose checkpoints upon rollback which precede events in EDS($E$). Rollback is said to be *minimal* if there are no intervening local events between the events in EDS($E$) and the checkpoints which

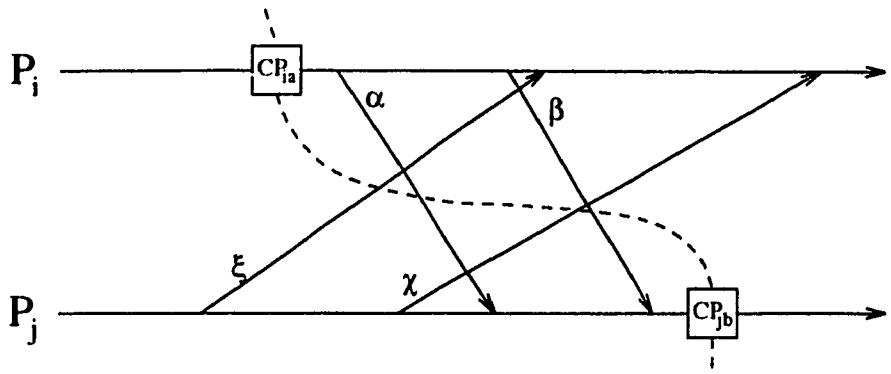


Figure 1.11: Backward and retransmission dependent event sets

constitute the recovery line chosen for rollback. Pre-planned checkpointing schemes are more likely to produce a minimal rollback than unplanned ones because of the consistent way in which recovery lines are formed. Also, it can be easily seen that in order to provide correct recovery, all events in BDS and RDS would have to be re-executed upon rollback to either resend messages or re-consume repeated messages. Otherwise, the re-execution would not be the same as before the error occurred.

[PASS88, VENK88] define consistent recovery lines in terms of the event dependency sets outlined above. Informally, for a recovery line to be consistent, all checkpoint events constituting it should be concurrent (according to [LAMP78]) with no interdependencies. More specifically,

- all checkpoints in the recovery line should "happen before" events in EDS.
- no two checkpoints in the recovery line should be in the same process.
- BDS($CP_{ia}$,$CP_{jb}$)= $\emptyset$ for all checkpoints $CP_{ia}$,$CP_{jb}$
- RDS($CP_{ia}$,$CP_{jb}$)= $\emptyset$ for all checkpoints $CP_{ia}$,$CP_{jb}$

Therefore, the focus of many domino-free checkpoint and rollback recovery algorithms is on removing backward and retransmission event dependencies.

## 1.6 Naive algorithms

Checkpointing and rollback-recovery seem like trivial operations at first glance, however they are quite complex. This section outlines what seem to be obvious algorithms for performing these operations and points out their deficiencies.

### 1.6.1 Naive Checkpointing

Checkpointing itself is nothing but the saving of process states. Therefore, an unplanned strategy which simply saves process states at random times seems to be a simple solution to perform checkpointing. However, although unplanned strategies(see section 1.5.5) for checkpoint creation are simple, they also suffer from the most anomalies. For example, consider the following algorithm: Every process $P_i$ creates a checkpoint immediately after it sends a message to any process $P_j$. Suppose there is no event occurrence between the sending of messages and the checkpoint creation. This will certainly ensure that the most recent set of checkpoints will form a consistent recovery line. However, the storage overhead of saving that many checkpoints would make this solution undesirable.

The naive algorithm can be altered in several ways to reduce the cost mentioned above. However, as will be shown below, while these modifications solve the problem of storage overhead, they introduce new problems of their own.

- To reduce the cost of storing checkpoints after every message send, all processes create a checkpoint after every $n$ message sends. Although this will reduce the number of checkpoints, it may lead to the domino effect described in section 1.5.5 since the checkpoints may not form consistent recovery lines.

- Every process creates a checkpoint at fixed intervals in "instantaneous" physical time. This solution would work for distributed systems with synchronized clocks. Otherwise, it would require the use of vector or logical clocks.

### 1.6.2 Naive Rollback and Recovery

When a process discovers a fault, it initiates its recovery by rolling back and restoring the system state which was saved in its last checkpoint. This may necessitate the rollback of other processes in the system also.

An obvious solution to the restoration of the other process states is to have the initiating process send all other processes a message requesting them to rollback to their most recent checkpoint. However, this may force processes to roll back unnecessarily if they have not been contaminated by the faulty actions of the initiator. Furthermore, in a distributed system it cannot be guaranteed that all processes will recover the state in their checkpoints at the same time. If no synchronization protocol is used for the rollback-recovery, processes may repeatedly rollback and recover indefinitely. Consider the two processes in Figure 1.12. $P_1$ discovers an error and rolls back to checkpoint $CP_i$ after having sent message $\beta$ to $P_2$. Meanwhile, $P_2$ has sent message $\alpha$ which is still in transit when the error in $P_1$ occurs. After $P_1$ recovers, it receives $\alpha$ but it is in a state where it has not yet sent $\beta$. This places the system in an inconsistent state since $P_2$ has received $\beta$ but $P_1$ has not yet sent it. When $P_2$ is finally informed that it should roll back, its recovery *undoes* the sending of $\alpha$. The system is again in an inconsistent state since $P_1$ has received $\alpha$ but $P_2$ has not sent it. This forces $P_1$ to roll back again in order to bring the system into a consistent state. The progress requirement of the system is thus violated. This phenomenon is called *livelock*[KOTO87].

**Definition 7** *The cyclic rollback of a set of inter-related processes which violates the progress requirement of a "distributed system execution" is called* **livelock**.

Livelock is a type of oscillation which can continue indefinitely. The naive algorithm presented above demonstrates the need for careful design of the checkpointing and rollback recovery algorithm to ensure that livelock does not occur.

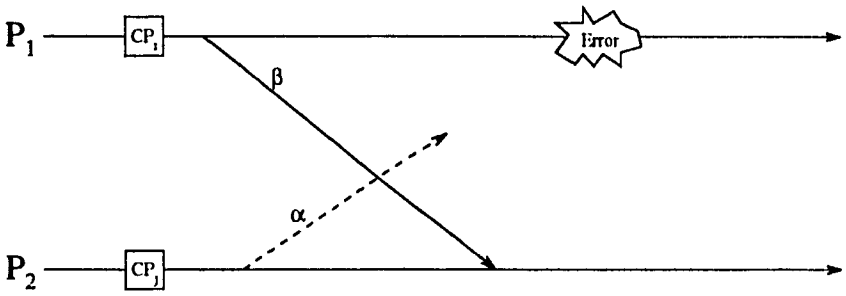


Figure 1.12: Infinite rollbacks due to lack of synchronization

# Chapter 2

# Related Work

*Good artists copy. Great artists steal.*
*- Pablo Picasso*

There has been considerable effort in the research community for the development of Checkpoint and Rollback Recovery algorithms. Most of the algorithms have been designed specifically for fault tolerance applications. However, we would like to use checkpointing and rollback recovery in the context of distributed debugging. This chapter presents a cross-section of the major algorithms for checkpoint and rollback recovery systems which have appeared in the literature over the years and analyzes them with respect to their suitability for various applications. Their deficiencies and strong points are also discussed.

## 2.1 Global Snapshot Algorithm

The algorithm for detecting stable global states of distributed systems presented in [CHAN85] can also be used for checkpointing. It is based on the creation of "global snapshots"[1] of the distributed system state. Snapshots can be looked upon as valid recovery lines to which a process can rollback to.

A global snapshot is obtained using *marker messages.* Marker messages inform processes that a snapshot of the system state is being taken. When a process wishes to initiate a snapshot, it saves its local state in a checkpoint, and transmits a marker

[1]The details of the algorithm are left out of the discussion. Instead a basic overview is given here.

message on all of its outgoing channels to inform the other processes to take appropriate actions. Upon receipt of a marker message, a process will check to see if it is the first marker that it has received. If it is, the process will save its local state, transmit a marker message on all of its outgoing channels, and save all incoming messages. If it is not the first marker message, it will stop the recording of messages on the incoming channel (i.e. the channel that the message was transmitted over). The process therefore stops recording message information when it has received a marker message on all of its incoming channels. When all processes have stopped recording message information, the snapshot is complete, and includes all of the process local states along with any message information that was saved.

### 2.1.1 Analysis of Global Snapshots

The authors of [CHAN85] present a simple method for creating consistent recovery lines. Unfortunately, only the checkpointing algorithm is presented and the recovery of the system is assumed to be trivial. Therefore, no rollback recovery algorithm is given. A recovery algorithm based on global snapshots which uses logical clocks is presented by Morgan in [MORG85].

The global snapshots algorithm is a global coordinated scheme and therefore requires *all* processes to be involved in the creation of recovery lines. Therefore it does not take advantage of causal dependencies to minimize the number of checkpoints which must be saved. It is also computationally expensive due to the large number of marker messages needed to form recovery lines [SPEZ89].

## 2.2 Koo and Toueg algorithm

Koo and Toueg present two algorithms in [KOTO87], one for checkpointing, and one for rollback and recovery, for use in fault tolerant applications. The algorithms are designed to create consistent recovery lines and recover the system to a correct and consistent state. The algorithms follow the pre-planned recovery line strategy. They define two types of checkpoints for the purposes of their algorithm.

- **Permanent checkpoints (PC)** are checkpoints which, once created, are never destroyed. This type of checkpoint guarantees that all computation leading up to the creation of the checkpoint will not have to be repeated should an error occur.

- **Tentative checkpoints (TC)** are checkpoints which can either be destroyed, or converted to permanent checkpoints at a later time.

### 2.2.1 Koo and Toueg checkpointing algorithm

The checkpointing algorithm is based on a two-phase commit protocol. There are $n$ processes involved in the computation.

- **Phase 1:**

  A process $P_i$ initiates the creation of a recovery line by creating a tentative checkpoint $TC_i$ and sending a message $\alpha$ to all other processes $\{P_j\}, j = 1, 2, .., n$ requesting them to do the same. $P_i$ is blocked until it receives a reply from all other $\{P_j\}$. If some process $P_j$ can create a tentative checkpoint $TC_j$, it does so. $P_j$ then sends a message back to $P_i$ informing it of whether or not $TC_j$ was created.

- **Phase 2:**

  If $P_i$ receives replies informing it that all other processes have created a tentative checkpoint, it decides that all tentative checkpoints should be made permanent. Otherwise, it decides that all tentative checkpoints should be discarded. $P_i$ broadcasts its decision to all processes informing them of what to do. Meanwhile, all processes waiting for the decision refrain from sending application messages until they receive a reply from $P_i$.

A process $P_j$ creates $TC_j$ upon receiving $\alpha$ *only* if $TC_i$ has recorded the receive event of any message $\beta$ sent from $P_j$ , and the send event of $\beta$ has not been recorded in the last permanent checkpoint $PC_j$ created by $P_j$. For example, in Figure 2.1 , upon

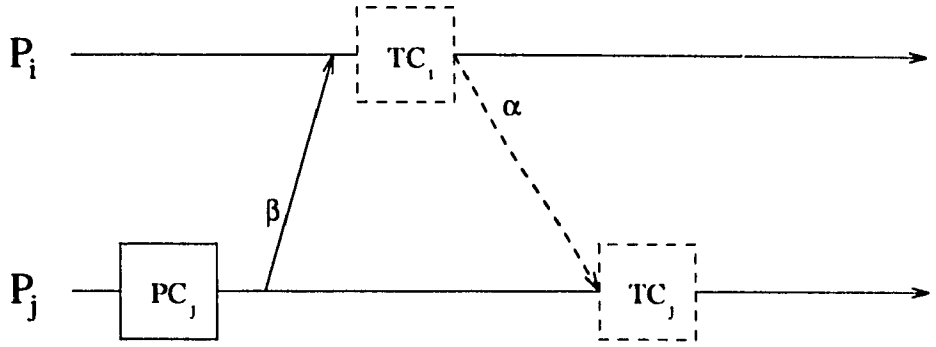


Figure 2.1: Creation of tentative checkpoint

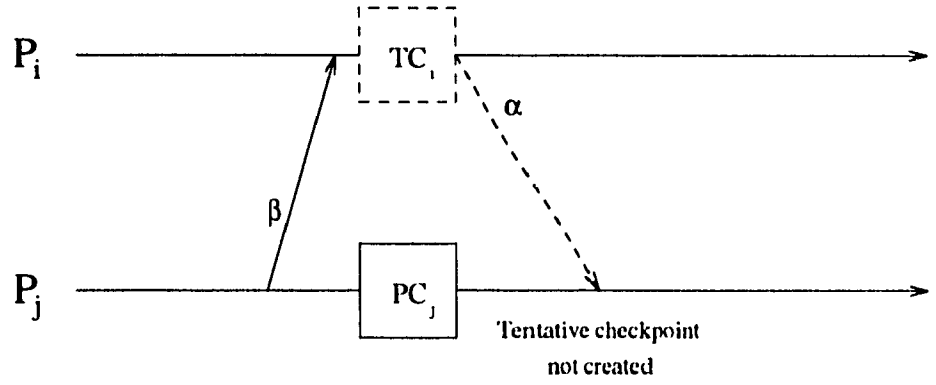


Figure 2.2: Non-creation of tentative checkpoint

receiving message $\alpha$, $P_j$ creates a tentative checkpoint $TC_j$ since $P_i$ has recorded the receipt of message $\beta$ in $TC_i$ and the sending of $\beta$ has not been recorded in $P_j$'s last permanent checkpoint (i.e. $PC_j$). Conversely, in Figure 2.2, $TCj$ is *not* created since the sending of $\beta$ is reflected in $PC_j$. Because of the consistent manner in which the checkpoints are created, retransmission dependencies cannot exist.

This also ensures the creation of only a minimal number of checkpoints. A permanent checkpoint is created only if its non-creation would cause the recovery line to form an inconsistent global state. (i.e. the receipt of the message is reflected in one of the checkpoints of the recovery line, but its sending is not).

The question arises as to how processes can know which events have been saved in the initiating process' tentative checkpoint. This information is exchanged by

appending labels to all messages, which indicate the last message sent or received by that process [2].

What if failures occur during checkpoint creation? The authors have made the assumption that process failures will always be reported so that deadlock cannot occur. If a process fails before it can respond to the initiating process, the initiating process considers the failed one as not having created a tentative checkpoint and continues normally. Processes that are waiting for a decision from the initiator will remain blocked until the initiator restarts.

When the failed process restarts, if its last checkpoint is a permanent one, there is no problem and the rollback and recovery algorithm can be immediately invoked. However, If the last checkpoint is tentative, then the process must decide whether the last checkpoint should be discarded , or made permanent before invoking the rollback-recovery algorithm. Two situations are possible:

- **The failed process is the initiator.** If this is the case, it can simply discard the checkpoint and inform all other processes to do the same. (Note all other processes will have still been blocked waiting for a response from the initiator)
- **The failed process is not the initiator.** In this case, the process contacts either the initiator, or another process to determine the outcome of the initiator's decision and follows it accordingly.

### 2.2.2 Koo and Toueg Rollback-Recovery

This algorithm is also based on a two-phase commit protocol. There are $n$ processes involved in the computation.

- **Phase 1:** The rollback initiator, process $P_i$ ,sends a message $\alpha$ to all processes $\{P_j\}, j = 1,2,..,n$ requesting them to restart from their last checkpoint. It then blocks until it receives a response from all processes $P_j$. Each process $P_j$ responds by sending a message indicating its willingness to restart and then blocks awaiting further instruction from $P_i$.

[2] details have been left out for simplicity but can be found in [KOTO87]

- **Phase 2:** If *all* processes are willing to restart, then $P_i$ sends another message informing them to go ahead and do so.

A process rolls back only if another process' rollback undoes the sending of a message to it.

### 2.2.3 Analysis of Koo and Toueg Algorithms

The checkpointing algorithm guarantees that a minimal number of checkpoints will be created, and that a minimal number of processes will be forced to roll back should an error occur. This ensures that minimal storage requirements will be needed to store the checkpoints. Both the checkpointing and rollback recovery algorithms are tolerant to failures during their execution thereby allowing multiple rollback and checkpointing to take place in the system at any given time.

One of the deficiencies of these algorithms is that *all* processes in the system are involved in the protocols for taking checkpoints and rolling back. The algorithms do not take advantage of causal relationships between processes to reduce this overhead. Furthermore, the algorithms are *intrusive* i.e. they introduce messages into the system which are *solely* for the purpose of checkpointing and rollback. Therefore, there is an increase in communication overhead in the system.

The algorithms guarantee that the system will be in a consistent state upon recovery but they fail to eliminate backward dependencies. Therefore, even if the state is consistent, it may not be correct with respect to the encapsulating application. Failure to eliminate backward dependencies leads to a loss of messages upon recovery. The Koo and Toueg rollback algorithm states in [KOTO87]: "*... the rollback of a process q forces another process p to rollback* **only** *if q's rollback undoes the* **sending** *of a message to p.*"[3]. The algorithm ignores the situation where the rollback of a process q undoes the *receiving* of a message. In Figure 2.3 , the sending of message $\beta$ forces $P_2$ to rollback with $P_1$. However, no provision is made for re-sending $\alpha$ . Therefore, $\alpha$ is lost upon recovery.

[3]The emphasis is the author's.

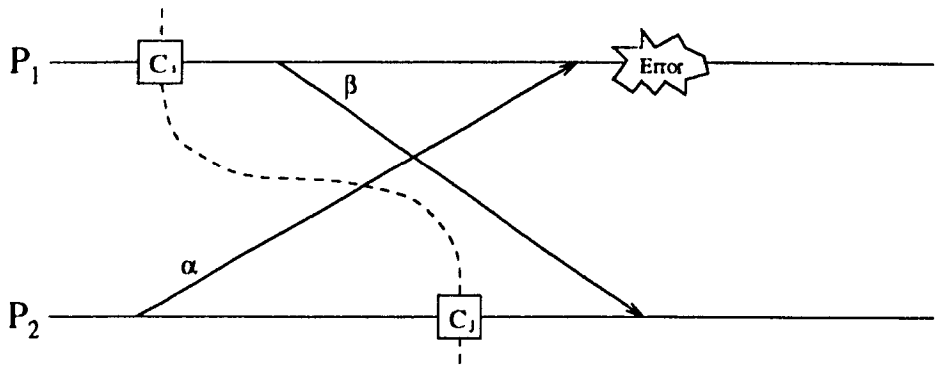


Figure 2.3: Loss of message upon rollback

Another problem with this approach is that the two-phase commit protocol forces processes to block while the algorithms are in action. This is undesirable since it temporarily stifles the progress of the computation. However, the authors acknowledge that this protocol could be replaced by a more expensive, but non-blocking three-phase commit protocol.

## 2.3 RLV Algorithms

The *non-intrusive* algorithms presented in [VENK87] eliminate backward and retransmission event dependencies. They are considered non-intrusive because they do not introduce any extra messages in the system. The algorithms presented also follow the pre-planned strategy of creating checkpoints which form consistent recovery lines.

The authors present several approaches to removing backward and retransmission dependencies in order to eliminate the domino effect.

### 2.3.1 Elimination of backward event dependencies

In order to eliminate backward dependencies, messages sent during execution of the distributed computation need to be saved for re-sending upon rollback and recovery. There are two approaches suggested for selecting the messages to be saved:

- Each process saves *all* messages that it has received during execution in a separate storage area. When the system is rolled back , messages are selected from

the storage area to mimic the original sending.

- Only messages which are included in a BDS set are saved for re-execution. An algorithm is needed in this case to determine which send events are actually part of the BDS.

The RLV algorithm uses the second approach. It selectively identifies the minimal set of messages whose send events are part of the BDS thereby eliminating all backward dependencies.

### 2.3.2 Elimination of Retransmission event dependencies

During re-execution, the messages whose receive events are part of the RDS must be consumed otherwise their re-sending will change the original execution. Two methods are proposed:

- All processes ignore repeated messages. This way when RDS messages are received at a process they will simply be ignored.

- The checkpoint creation is coordinated such that a minimal rollback can always be assumed.

The RLV algorithm plans the establishment of its recovery lines such that retransmission dependencies are eliminated by ensuring a minimal rollback.

### 2.3.3 RLV Checkpointing

As in [KOTO87] , the authors of [VENK87] also distinguish between two different types of checkpoints, although the basis of their classification is different. In [KOTO87] classification is based on the permanent or temporary status of checkpoints whereas in RLV all checkpoints are considered to be permanent and their classification is based on *ownership*. The two types of checkpoints are:

- **Self-induced checkpoints (SIC):** These are checkpoints which are created in response to a request by the application program executing on the distributed system.

- **Response checkpoints (RC):** These are checkpoints which are created as a result of the creation of SIC's in other processes. RC creation is completely transparent to the application program.

The basic idea is that one process in the application program creates a SIC which in turn causes the creation of RC's in other processes as a result of the process to process interactions. This model is adopted to confine the creation of checkpoints forming recovery lines among those processes which actually interact with each other. i.e. those which are causally dependent on each other.

A unique identification label pair called **SICid**, is associated with each SIC created in the system. The pair $(i, n)$ is the SICid for the $nth$ SIC created by process $P_i$ (i.e. $i$ is the unique process id of the process which created the SIC). An analogous label exists for the RC's called **RCid**. However in an RCid, the pair $(i, n)$ corresponds to the process id and checkpoint ordinal number of the *SIC* whose creation led to the creation of the RC. The RCid is used to identify an RC as being part of the recovery line which was initiated by the SIC and the RC is said to be *owned* by the process whose SICid it inherited. Therefore, all checkpoint id's (SICid or RCid) can be viewed as a pair $(i, n)$ where $P_i$ is the owner and $n$ is an ordinal number.

Every process $P_i$ maintains a vector $CCP_i$ of $m$ counters, one for each process in the system. $CCP_i[j]$ represents the number of SIC's that process $P_j$ has created to date as perceived by $P_i$. Whenever $P_i$ sends a message to any other process, it appends its $CCP_i$ vector to it so that the receiving process can update its own $CCP$ vector. For clarity, the appended $CCP$ is termed $RCP$. When a message is received by $P_i$, it inspects the counters in the appended $RCP$ and if $RCP[k] > CCP_i[k]$ for any $k$ ,then $CCP_i[k]$ is replaced by the value of $RCP[k]$. The receiving process then creates an RC which is given an RCid of $(k, RCP[k])$ i.e. the RC *inherits* the SICid of the SIC which led to its creation. However, if $RCP[k] < CCP_i[k]$ then the receive event of this message is stored in all RC checkpoints in $P_i$ *owned* by $P_k$ and having a checkpoint ordinal number greater than $RCP[k]$. i.e. The message is saved in the checkpoint so that it can be re-received by the rolled back process $P_i$ after recovery.

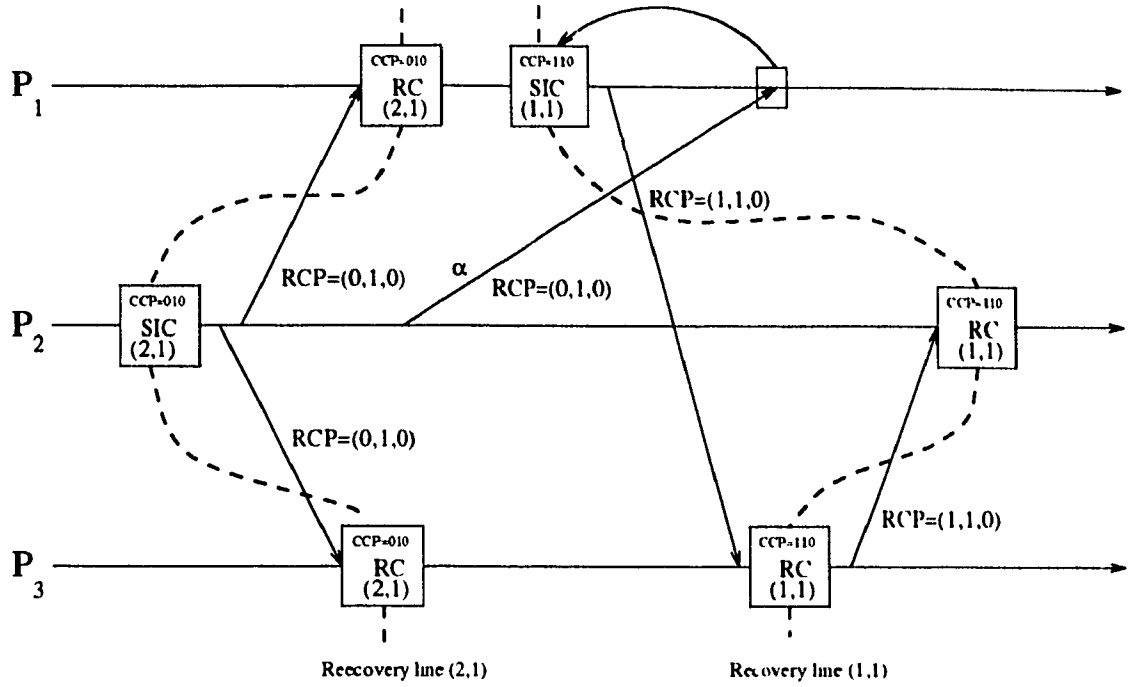


Figure 2.4: Storing of messages in RLV to preserve BDS

The $CCP$ vector idea is used to keep the global knowledge of checkpoint creation consistent across all processes.

The storing of messages into checkpoints described above has the effect of eliminating backward event dependencies by allowing messages to be replayed upon rollback to any of these checkpoints. For example, in Figure 2.4 the receipt of message $\alpha$ by $P_1$ creates a backward dependency since it crosses the recovery line $(1,1)$. Since $RCP[1]$ of $\alpha$ is equal to 0 and , at the point in $P_1$ where $\alpha$ is received, $CCP_1[1] = 1$, the receive event of $\alpha$ is stored in $SIC(1,1)$. Therefore, if a rollback occurs in $P_1$ to $SIC(1,1)$ the knowledge that $\alpha$ should be re-received is not lost.i.e. $P_2$ would not have to resend $\alpha$ for $P_1$ to recover its state correctly since $P_1$ has a copy of the message buffered.

### 2.3.4 RLV Rollback-Recovery

A process detecting a fault will initiate the rollback by restoring its own state to the state saved in its most recent $SIC(i,n)$ preceding the fault, and discarding all

checkpoints which follow it. It then sends out a *rollback control* message containing the identification label of the restored $SIC$ (i.e. $(i,n)$) on all of its outgoing channels. When a process receives a rollback control message, it checks to see if it has any RC's with RCid = $(i,m)$, where $m$ is greater than or equal to $n$. If it does, then this process is involved in the rollback and it must discard its current state and restore its state to that which was saved in $RC(i,m)$. To preserve backward dependencies, the process also inserts any messages that were saved in $RC(i,m)$ into its receive queue so that they will be re-received upon recovery. After doing so, it eliminates all checkpoints after ,and including, $RC(i,m)$ since these checkpoints may be recreated after the rollback.

Once a process has rolled back and restarted, all of its channels except for the one on which it was informed of the rollback (if any) are placed in a *cautious state*. In this state, the process examines the RCP's of all incoming messages to determine whether they are *post* or *pre*-rollback messages. A message is pre-rollback if its $RCP[k]$ entry is greater than or equal to $m$ where $k$ is the owner of the checkpoint which the process rolled back to and $m$ is the checkpoint ordinal number of that checkpoint. If a message is a pre-rollback message, then it is discarded, otherwise it is processed normally like a regular application message. The channel exits the cautious state when it receives another rollback message since this indicates that all pre-rollback messages on that channel must have been flushed out of the system.

### 2.3.5 Analysis of RLV Algorithms

Since the RLV algorithms are non-intrusive, they do not have a high communication overhead. They are inexpensive because the control information is simply piggy-backed onto the system application messages during the checkpointing phase. they do however introduce special control messages during the rollback phase.

The formation of recovery lines using the SIC/RC scheme has the advantage that not all processes are required to create checkpoints in order for the system to be restored to a consistent state after a failure. Furthermore, because the RC checkpoints are created as a result of the receipt of system application messages, the recovery lines

are always consistent and therefore retransmission dependencies cannot exist. The algorithms also succeed in removing backward dependencies and reducing checkpoint storage overhead by saving only those messages which cause backward dependencies in the checkpoints.

In some applications, checkpoints may not be permanently useful and their existence may cause a substantial storage overhead. The RLV algorithms make no provision for discarding checkpoints which are no longer needed.

Lastly, by introducing the notion of a cautious state, these algorithms solve the problem of synchronization of rollback-recovery ensuring that any recovery will place the system in a consistent state. Therefore, livelock cannot occur.

The algorithm places no restriction on the applications behaviour and is transparent to the application. It can also handle multiple concurrent rollbacks .Since the algorithm uses a pre-planned scheme for creating recovery lines, it is suitable for both communication and computation intensive applications (see section 1.4.2).

## 2.4 BCS Algorithms

[BRCI84] presents algorithms for the checkpointing and rollback of distributed applications which are also based on the pre-planned strategy of creating recovery lines.

### 2.4.1 BCS Checkpointing

The authors of [BRCI84] use a notion similar to that of SIC's and RC's as described in section 2.3.3 in their checkpointing and Rollback-recovery algorithms. However, their significance in forming recovery lines is quite different.

Each process in the system maintains a counter called the *K counter* which keeps track of the number of checkpoints within that process. Unlike the CCP counter in [VENK87], this counter counts *all* checkpoints whether they are SIC-types or RC-types[4]. Each SIC-type or RC-type is assigned a *qID* which is the value of the cor-

[4]Note that for clarity the terms SIC and RC have been replaced with the terms SIC-type and RC-type. In the context of the current discussion, the reader should attach the same meaning to these terms as to SIC and RC.

responding process' K counter at the time of its creation. Every checkpoint $CP$ in process $P_i$ contains a list of process ID's called the *PEset*. This list represents the set of processes to be rolled back, as perceived by $P_i$, in the event that $P_i$ should be rolled back to checkpoint $CP$ after an error.

A process' current $K$ counter value is appended to all of its outgoing messages. For clarity, the appended $K$ counter is termed $R$. When a process $P_j$ receives a message from another process $P_i$, it inspects the counter $R$ appended to it. If $R > K$, then $P_j$ creates as many RC-type checkpoints as needed to update its $K$ counter to the same value as $P_i$'s. It also stores $P_i$'s process ID in each of the checkpoints created.

If $K > R$, then the message is stored in all checkpoints whose $qID = n$, where $n > R$. This is done to remove backward dependencies by ensuring that upon rollback to any of these checkpoints, the message will not be lost. Note that if $K > R$, the message crosses a recovery line and therefore it must be a backward dependency.

A recovery line consists of all checkpoints (SIC-type or RC-type) in the system with the same qID. Note that this is different from the method used in [VENK87] wherein checkpoints within a recovery line have the same *SICid*.

### 2.4.2 BCS Rollback-Recovery

The rollback and recovery operation in the BCS algorithm proceeds in two phases. When a process $P_i$ encounters a fault, it rolls back its operation to its most recent checkpoint. It then sends a *rollback offer* message to the first process in the PEset of the checkpoint. This offer contains a list of processes called the *LPIset* (which is actually just the PEset) and the recovery line number. After the offer has been sent, the process "sleeps". When $P_i$ receives a rollback offer, it inspects its checkpoint $CP$ corresponding to the recovery line number contained in the offer. It then appends the PEset contained in $CP$ to the LPIset received in the offer. This new LPIset is then sent by $P_i$ in another rollback offer message to the first process in the new LPIset which has *not* been contacted yet. When the last process $P_j$ in the chain receives the offer, it changes the offer to a *rollback accept* message. This message is then sent to $P_j$'s predecessor in the LPIset. All processes receiving the accept message will do the

same and then enter a second sleep state.

When the rollback initiator $P_i$ receives the accept message, the second phase of the algorithm begins. $P_i$ will resend the rollback offer message in the same manner as before. This time, the receiving process will wait until all processes in transit have arrived, and then restore the state of its checkpoint and channel buffers.

### 2.4.3 Analysis of BCS Algorithms

These algorithms create consistent recovery lines. Furthermore, the recovery is correct since both retransmission and backward dependencies are accounted for in the creation of checkpoints. As in [VENK87], the recovery lines are created such that a minimal number of processes will be forced to roll back should an error occur. This is due to the fact that the checkpointing algorithm creates recovery lines on the basis of causal relationships between processes.

The algorithms are non-intrusive and therefore have minimal communication overhead.

There are however a few loose ends in this algorithm [PASS88]. When the second rollback offer is sent, the processes are supposed to wait for all messages in transit to have been purged. This is needed to allow multiple rollbacks to occur concurrently. However, the authors make no mention of how long a process should actually wait before proceeding with the restoring of the state saved in the checkpoint. If they do not wait long enough, then pre-rollback messages may be mistaken for post-rollback messages. If they wait too long, then post-rollback messages may be mistakenly discarded.

As in the RLV algorithms, no provision is made for discarding checkpoints which are no longer useful.

## 2.5 Optimistic Recovery (OR)

This section introduces the optimistic recovery algorithm [STRO85, GOGO90] which is one of the most important algorithms for checkpointing and rollback recovery based

on an unplanned strategy.

The idea behind the OR algorithm is to allow processes to checkpoint their state asynchronous of process execution. The checkpoints do not form consistent recovery lines since they are created independently of each other. However, messages are saved in such a way that if some process $P_i$ rolls back its state to a previously saved checkpoint $CP$, it can continue its execution from that point without forcing the re-sending of any messages that may have been received by $P_i$ between the creation of $CP$ and the rollback. This is done by *logging* messages so that upon rollback, $P_i$ can consume messages on which it is dependent from its log rather than from a communication channel.

### 2.5.1 OR Algorithm

One of the goals of this algorithm is to restore system states in a consistent way such that for a communication channel between any two processes $P_i$ and $P_j$ ,no messages are lost (i.e. sent by $P_i$ but not received by $P_j$) and no messages are received by $P_j$ but not sent by $P_i$. A message which falls into the latter category is called an *orphan.* In pre-planned strategies, orphans do not occur since the recovery lines are specifically created such that they are consistent. The OR philosophy, on the other hand, is to save time by allowing orphans to occur and take care of them during rollback. This is justified by the assumption that orphans will probably occur infrequently.

During the application's normal operation, processes execute normally, creating checkpoints as they please without regards for any other processes. Therefore, there isn't really any checkpointing algorithm since processes do not have to adhere to any protocol during checkpoint creation.

Each process participating in the computation is required to keep track of its dependency on the state of other processes in a *dependency map.* Dependency maps define the set of unlogged messages on which the current state of the process depends. A process appends its own dependency map on all of its outgoing messages so that it can be inspected by the receiver. This is done so that the global knowledge of which processes are dependent on which messages is known to any processes in the

interacting set. Processes use their dependency maps to detect whether or not they have performed any computations which causally depend on events which have been *lost* by another process. i.e. orphans.

The protocol for maintaining the dependencies in the system uses *half-sessions*. In these half-sessions, a sender process consecutively numbers all outgoing messages and saves a copy of all messages which it knows have not yet been logged by the receiving process. It also attaches its dependency map to all outgoing messages. Upon receiving the message, the receiver updates its own dependency map according to the one sent in the message. Eventually, the receiver will log the message to disk and send a *log message* to all other processes. This log message is the mechanism which is used in OR to update the global knowledge of dependencies. When a process receives a log message, it removes the particular dependency indicated in the message from its dependency map (if it exists).

Upon rollback, a message is considered *lost* if it was already sent at the time of the checkpoint creation and it has a sequence number greater than the last logged message. The sending process assumes that message to be lost because at the time of the checkpoint creation, the sending process had not yet received a log message confirming that it had been logged on the receiving end.

At the time of rollback, all processes are informed about which messages are currently considered lost. If a process is in a state where it has already received a "lost" message, then that message is an orphan and the process is rolled back to an earlier state. Rollback stops when all processes are rolled back to a state in which they do not *depend* on any lost messages. That is, their state is not causally dependent on a lost message. In this algorithm the receiving process is always responsible for detecting duplicate and lost messages.

### 2.5.2 Analysis of Optimistic Recovery

The authors of [STRO85] claim that the domino effect is completely eliminated. Unfortunately, this is not the case since the presence of orphans can cause subsequent rollbacks of the system state. However, the domino effect is somewhat minimized

since only messages which haven't been recorded by a receiving process may cause subsequent rollbacks in the sending process at the time of message sending.

The basic philosophy of OR is to minimize the time needed to create checkpoints so that disruption to the system performance during the normal operation of an application can be minimized. This tradeoff in speed occurs in the rollback and recovery phases since a high overhead is incurred to rollback the state of a process(es) at rollback time. The authors justify this tradeoff by assuming that in most operation environments, faults and errors, which require the rollback of some part of the system, occur infrequently. This algorithm is therefore unsuitable for a distributed debugging environment where frequent rollbacks are expected and desired.

## 2.6 Discussion of Algorithms

In this chapter, we have presented an overview of four important algorithms from the literature for checkpoint and rollback recovery. Discussion of more algorithms for checkpointing and rollback recovery in distributed fault tolerant and real time systems can be found in [SAMK90].

It should be noted that an exhaustive computer search has been performed to locate more recent work on checkpoint and rollback recovery algorithms. Although the papers were ordered several months ago, they are unfortunately not available at the time of writing. Only the basic ideas are presented here for future reference. [CRJA91] presents a set of protocols for checkpointing large distributed computations based on a time-stamp scheme. [CHKE91] present algorithms which do not require the communication subsystem to deliver messages in a FIFO order. In [WOWO90] the authors present algorithms which are deadlock-free and are based on atomic checkpointing of sender and receiver processes. [JOZW91] presents an algorithm suitable for both deterministic and non-deterministic computations.

All of the algorithms presented in this chapter have assumed that errors can always be detected by the recovery system. Therefore, an error which occurs in one process can never contaminate further checkpoints since a rollback to a previous safe

checkpoint would have occurred before the error could have propagated. However, if reliable error detection is not always guaranteed, then rollback to a particular checkpoint may not guarantee correct performance and may require further rollbacks until a real safe state has been reached. In the context of distributed debugging, this is not a unrealistic assumption since the purpose is only to be able to rollback the execution to a previous state regardless of whether or not that state is contaminated by a previous error. Discovering this contamination is part of the debugging process. However , in the context of fault tolerance applications, contamination of checkpoints due to previously undetected errors can have serious implications. Lin and Shin have researched the idea of recovery of applications in environments which do not support reliable error detection. In [LISH89] they present a modified version of the optimistic recovery algorithm which works efficiently in environments where error detection is not guaranteed.

Discarding of unwanted checkpoints is not addressed by any of the algorithms presented. This is an open problem which involves the *identification* of checkpoints which are no longer useful. Such checkpoints must be discarded in such a way that the remaining checkpoints would still form consistent recovery lines.

Although the algorithms in this chapter have been described in the context of distributed computing, it should be noted that much work has been done in the area of checkpointing and rollback recovery of distributed databases. These checkpointing schemes are based on transactions. Important work in this area can be found in [SOAG85, CHLI80, SOAG89, KUMA90].

As mentioned in section 1.5.3, one of the main problems in checkpoint and rollback recovery algorithms in the context of fault tolerant systems is the problem of optimal checkpoint placement. This problem is dealt with in [CHKR88], [KRKA84], and more recently in [FUKA91].

The algorithm with most significance to this thesis presented in this chapter is the RLV method . The RLV algorithms are very well suited for distributed debuggers because of their application transparency and correct maintenance of valid recovery lines. In the next chapter, it will be shown how the RLV algorithms can be modified

to support the checkpointing and rollback recovery of a larger class of applications.

# Chapter 3

# MRLV Checkpointing Algorithm

*The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka!" (I found it!) but "That's funny ..."*
*– Isaac Asimov*

Checkpointing of a multi-threaded task has not been addressed by the algorithms in the previous chapter. Although it might seem that supporting such an environment would involve a trivial extension to already existing algorithms, this is not the case. The presence of multiple threads within a task adds new problems which are not present in a single-threaded task. Allowing multiple threads to exist within a task implies that algorithms which base their system model on environments which support FIFO channels between tasks such as [VENK87, KIYO86, BRCI84, SOAG85, STRO85, KOTO87] cannot be easily implemented in a multi-threaded environment.

The ability to checkpoint and rollback such a task is very important for environments which support multi-threading such as the Mach operating system. We have chosen Mach as the implementation model for the checkpoint and rollback recovery algorithms and therefore, a new algorithm which supports multiple threads within a task has been developed with Mach as the target model. The algorithm is an extension of the RLV algorithms presented in the previous chapter and it is called *MRLV* (or *modified* RLV).

This chapter presents the basic concepts of the Mach operating system model which are relevant to the goal of checkpointing a multi-threaded task. The difficulties in checkpointing such a task using a typical rollback and recovery algorithm (in

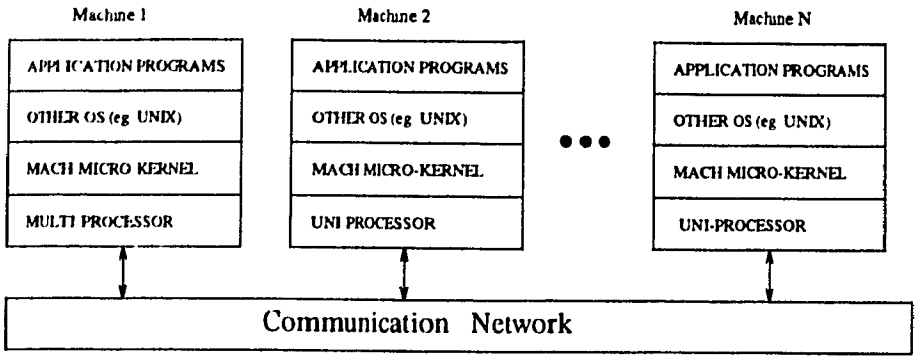


Figure 3.1: Transparency of underlying network provided by Mach micro-kernel

particular RLV) are discussed. Finally, the new MRLV algorithm is described in detail.

## 3.1 Relevant Mach Concepts

The Mach operating system, developed at Carnegie Mellon University, is designed with distributed computation in mind. In order to make the distributed computing environment more readily available, it is designed as a micro-kernel on top of which other operating systems such as Unix can be built. The goal of Mach is to provide an integrated environment consisting of networks of processors (multi or uni) on top of which distributed applications can be built easily. The networks and characteristics of the machines are made transparent to the programmer by the Mach kernel through the provision of efficient memory management and interprocess communication facilities (see Figure 3.1). The main concepts pertaining to the Mach model which are relevant to checkpointing issues are discussed in this section.

### 3.1.1 Multiple threads within a task

The unit of computation in Unix-like operating systems is the *process.* A process is defined as an instance of a program in execution [BACH86]. Each process executes only a single instance of one program at a time. On the other hand, the principle unit of computation in Mach is the *task* and is defined below.

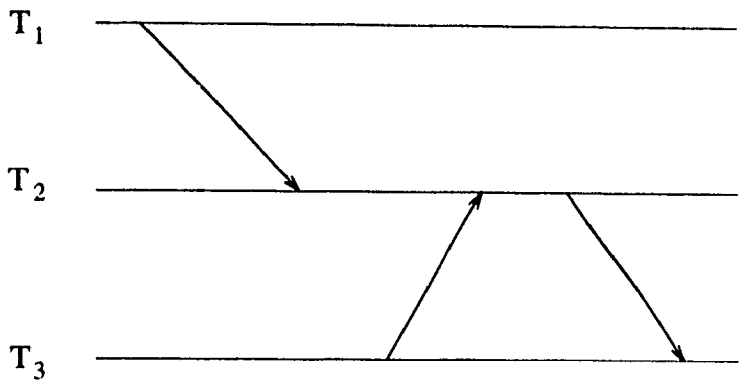


Figure 3.2: Threads of control in a distributed computation

**Definition 8** ***A task*** *is an execution environment within which computations (0 or more) can take place.*

For the current discussion, let us assume that all tasks execute on different host machines. In Figure 3.2, $T_1$, $T_2$, and $T_3$ denote three tasks running on different host machines. However, they are all working toward the same *global* computational goal. In this sense, each of the tasks is a *thread* of the global computation being performed on all machines collectively.

The concept of threads can be taken a step further by allowing each of the tasks themselves to contain several threads of control. i.e. within each task on each of the machines, there are several computations being performed concurrently.

**Definition 9** ***A thread*** *is a basic unit of execution which executes within a single task. A thread shares all resources, such as memory, with other threads executing within the same task. However, each thread has its own processor state, its own execution stack and a "small" amount of static storage which are unique to itself.*

The multi-threaded task model is very useful particularly for server applications where each thread can be put in charge of a particular aspect of a complex server [TEVRA87]. Furthermore, multiple threads in a task allow the possibility for exploiting the parallel processing power of the underlying machine architecture if the application is implemented on a tightly coupled shared memory multi-processor. The

multiple threads per task model adds a second level of parallelism to the distributed computation. The task level is the *inter-machine* parallelism provided by communicating tasks, and the thread level is the *intra-machine* parallelism allowed in a multi-threaded computational model.

The Mach operating system is designed to allow both the inter and intra machine parallelisms described above. Each Mach task can have several threads of control, each of which executes within the same task sharing all of its resources (eg. virtual memory space). A thread can be easily created or destroyed by the application programmer with the assistance of Mach packages designed specifically for these purposes [MACHE89].

A task and its threads can collectively communicate with other tasks and their threads. Figure 3.3 depicts communication between three Mach tasks each of which has multiple threads of control. $T_1$,$T_2$, and $T_3$ each execute on their own host machines communicating via message passing. Within each of these tasks are $N$ threads of control which , within their own task , execute independently. $T_1$, $T_2$, and $T_3$ represent the inter-machine parallelism and within each task, threads $1..N$ represent the intra-machine parallelism.

It should be noted also for completeness that threads within a single Mach task execute concurrently and may need to communicate with each other for synchronization purposes. Mach allows threads to communicate with each other either through message passing or via shared memory [MACHS89].

The concept of a Mach task is quite different from Unix's concept of a process. The Mach task can be viewed as a generalization of the Unix process to support finer grain parallelism . A Mach task with a single thread of control can be considered equivalent to a Unix process with respect to their computational behaviour. It should be noted however, that the presence of multiple threads within a task is not the only characteristic that distinguishes Mach tasks from Unix processes. Another relevant distinction is discussed in the next section.

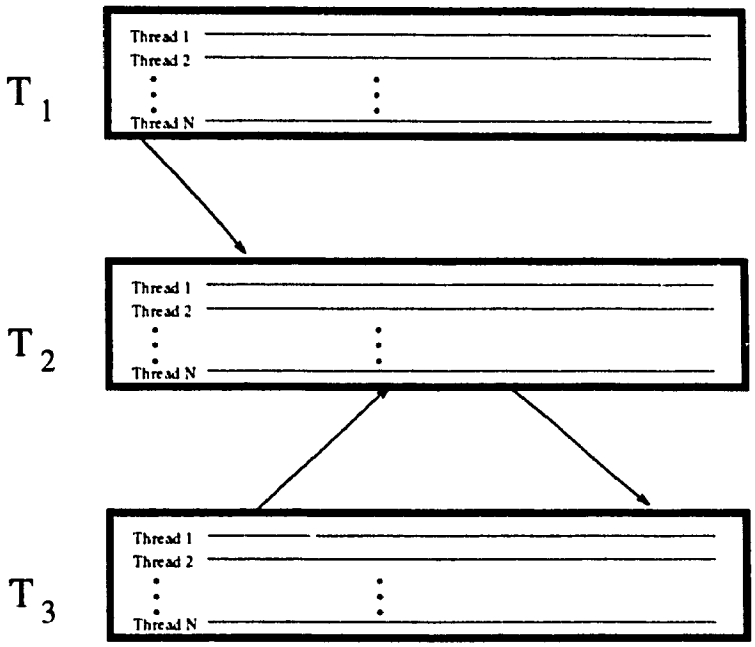


Figure 3.3: Distributed computation involving tasks with multiple threads of control

### 3.1.2 Mach ports

Tasks executing in a distributed computation send messages to each other. Messages may be exchanged for synchronization purposes or, because they include information needed by remote tasks to continue their computation. Nevertheless, there must be some specific semantics defined for the sending and receiving of messages. For sending and receiving messages, Mach adopts the *Port* semantic model. A port is defined as follows:

**Definition 10** *A* **port** *is a logical queue of messages which acts as a communication medium between threads.*

A message $\alpha$ is said to be *queued* on port $p$ if $\alpha$ has been sent from some task $T_i$ to the port $p$ and can now be *received* by some task $T_j$ on $p$ ($i$ may be same as $j$). The message is considered to be *received* once it has been dequeued from the port it was sent to by a receiving task.

All ports have *port rights* associated with them [MACHK90]. The two relevant types of port rights are:

- **Port Receive Rights**: A task which creates a port is considered to have *receive rights* for that port[1]. Receive rights for a port are necessary to dequeue a message from that port. Furthermore, only one task is allowed to own the receive rights to a port at any time.

- **Port Send Rights**: A task $T_i$ is said to have *send rights* to port $p$ if $T_i$ is allowed by the system to enqueue messages on port $p$. In other words, a task can send messages to a port only if it has *send rights* to that port. Unlike receive rights, many tasks may own send rights to a given port. These send rights can be acquired by tasks in different ways which will not be discussed here. It should be noted however that a task which creates a port is automatically given send rights to that port by the operating system.

Basically, messages are sent to *ports* rather than to *tasks*. The task owning the particular port to which a message is being sent is of no concern to the sender. The sender of a message need not know who the receiving task is but is only concerned with the port where messages that it has sent will be enqueued. As a consequence of this model the receiving task does not know who the sender of a particular message is unless it has specifically been made available by the programmer at the application program level.

### Multiple ports per task

Mach allows the creation of multiple ports per task. The number of ports required is determined by the application program. This further reinforces the abstraction that messages are sent to ports rather than tasks since messages destined for several ports may actually be sent to the same task.

[1]These rights may be passed on to another task but this is not relevant to the current discussion, and it will be assumed from now on that port receive rights remain in the task which created them unless specified.

The semantics of multiple ports with respect to the multiple threads are as follows: Any thread within a task $T_i$ can send messages to any port for which the task $T_i$ has send rights. Similarly, any thread executing within $T_i$ can receive messages from any port for which $T_i$ has receive rights.

## 3.2 RLV Conceptual model

The algorithm which has been chosen as the base algorithm for MRLV is RLV. In the RLV model, tasks (or processes) are implicitly *single-threaded*. That is, within any given task there is no more than one computation being performed at any given time. The model is as follows:

- Tasks are interconnected by logical point to point channels.
- The interconnecting channels are reliable i.e. messages are not lost by the communication network.
- There is only one channel connecting any two given tasks.
- Messages are received in a FIFO manner.

The RLV communication model is depicted in Figure 3.4.

## 3.3 Mach model vs. RLV conceptual model

All of the algorithms discussed in the previous chapter have made the assumption (explicitly or implicitly) that the tasks which are being checkpointed contain only a single thread of control. None of them have addressed the problem of checkpointing a task which may have several threads of control executing within a single task's address space. This section discusses the problems with adapting a checkpointing algorithm to an environment which supports the features that the Mach operating system does. In particular, it discusses the problems in matching the RLV conceptual model onto a Mach-like computational model.

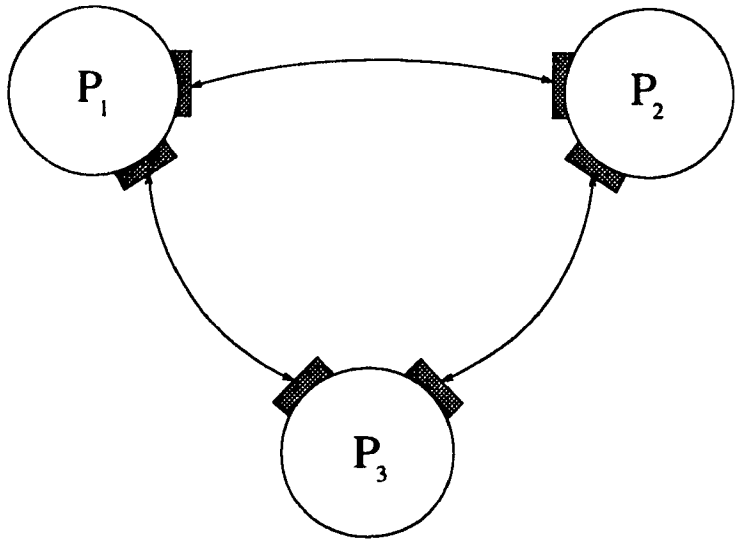


Figure 3.4: RLV communication model.

The main problem to overcome in adapting RLV to a Mach-like model is that Mach does not deal with communication on *channels*. Rather, it deals with communication between pre-defined *ports* which can receive messages from *several* different sources on remote hosts. Essentially, the Mach model is a generalization of the RLV channel concept to allow the definition of several channels between two tasks rather than a single channel between every pair of communicating tasks as in RLV. The Mach communication model is depicted in Figure 3.5. There are three communicating tasks each of which have several ports and several threads. Messages sent from one *task* to another need not be sent to the same port within the receiving task. Compare this with the RLV model depicted in Figure 3.4 which forces messages to be sent to the same "port".

### 3.3.1 Non-FIFO message ordering in Mach

A message sent by Mach thread $t$ (within a task $T_k$) to port $p$ will arrive at port $p$ i.e. enqueue at $p$) in another task $T_j$ in the order sent. Mach relies on the "transport protocol" of the communication sub-system to ensure this happens and TCP/IP carries out Mach's wish. However, since many threads within $T_k$ may send messages

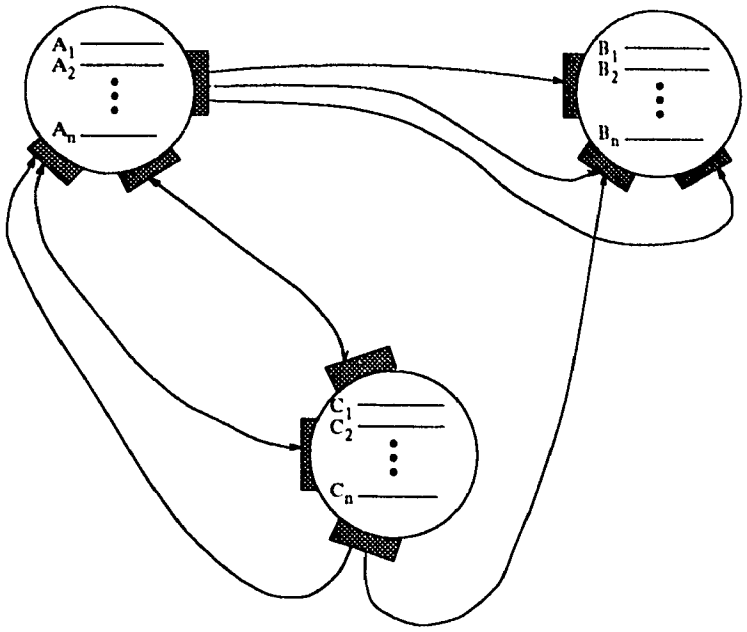


Figure 3.5: Mach communication model.

to many different ports residing in $T_j$, there is no guarantee that messages sent to $T_j$ will be ordered. This is since $T_j$ may have multiple threads executing in parallel which in turn receive messages from several ports.

Suppose that task $T_k$ has two threads $t$ and $u$ executing in parallel and that one of them has created two ports $p_1$ and $p_2$. Let the application that $T_k$ represents be written such that $t$ only receives messages queued on port $p_1$ and $u$ only receives messages queued on port $p_2$. Now suppose that some thread in a remote task $T_j$ sends two messages $\alpha_1$ and $\alpha_2$ to task $T_k$. i.e. $\alpha_1$ is sent to $p_1$ and $\alpha_2$ to $p_2$ in that order. If $t$ receives $\alpha_1$ on port $p_1$ before $u$ has received $\alpha_2$ on port $p_2$ then the message ordering is preserved. i.e. The messages sent by task $T_j$ are received by task $T_k$ in the order that they were sent (see Figure 3.6). The problem however, is that this may not be the case. Because of the different scheduling possibilities of $t$ and $u$, it is quite possible that $u$ receives $\alpha_2$ on $p_2$ *before* $t$ receives $\alpha_1$ on $p_1$. This scenario is depicted in Figure 3.7. This type of non-determinism makes it impossible to directly implement RLV-type algorithms which rely on FIFO ordering of messages on the Mach platform.

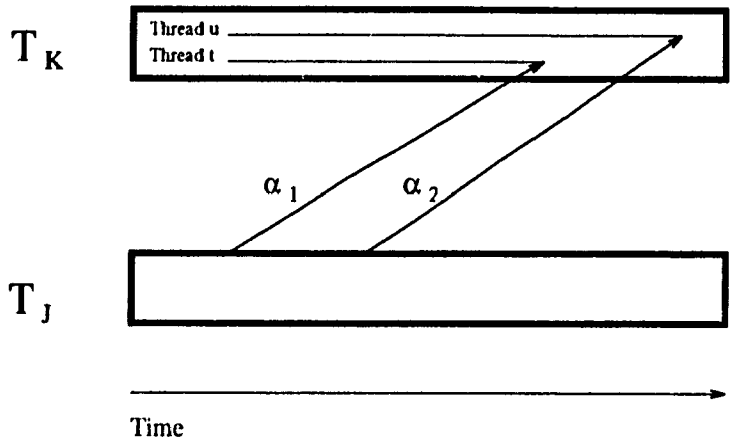


Figure 3.6: Coincidental FIFO message receipt by threads within a task

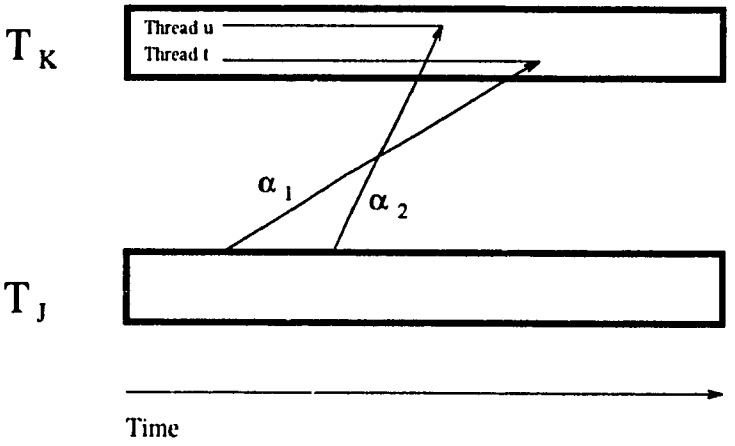


Figure 3.7: Non-FIFO message receipt by threads within a task

In RLV , message ordering is used primarily to detect the presence of pre-rollback messages in the rollback phase of the algorithm. Therefore, the main goal in adapting an RLV-type algorithm to a multi-threaded environment such as Mach is to modify the algorithm to remove this dependency on FIFO ordering of messages for pre-rollback message detection. This thesis proposes a scheme for detecting pre-rollback messages without relying on the FIFO property of channels.

## 3.4 Why RLV?

The MRLV algorithm is designed to operate in the context of a distributed debugger for applications running in the Mach environment. RLV has been chosen as a base platform upon which to design the new algorithm which solves the multiple thread / multiple port problem discussed in section 3.3. The other algorithms discussed in chapter 2 are unsuitable for the distributed debugger application for the following reasons: The Koo and Toueg algorithm is unsuitable since:

- All tasks must be involved in checkpoint creation.
- All tasks must block during checkpointing and rollback.
- Backward dependencies are not eliminated in recovery line creation.
- Algorithm forces blocking of all tasks during both checkpointing and rollback recovery.
- Dependency on FIFO channels.

The BCS Algorithm is not suitable since it is incomplete in terms of the rollback protocol and it is also dependent on FIFO channels. The optimistic recovery algorithm is unsuitable for the following reasons.

- Since it is an unplanned strategy, it cannot support the debugging of non deterministic applications.
- It does not completely eliminate the domino effect.

- Dependency on FIFO channels.
- Since the algorithm is designed to minimize checkpointing overhead in favor of rollback overhead, it is not suitable for a debugging environment which needs an efficient rollback mechanism.

The theoretical advantages of RLV over other algorithms have already been discussed in section 2.3.5 and are briefly reiterated here in the current context. These advantages are the basis for choosing RLV as a solid base algorithm.

- It is a process-level scheme therefore not all tasks are required to take checkpoints when one task wishes to do so.
- Simple coordinated checkpointing scheme. Since it is a pre-planned checkpointing algorithm, it can support the rollback recovery of non-deterministic applications.
- No livelock possible.
- Since recovery lines are built consistently, no domino effect can occur.
- By piggy-backing rollback information onto application messages, an opportunity is provided for design tradeoffs.

Another advantage of using RLV for the proposed modifications is that since the original RLV algorithm makes use of the FIFO channel assumption only in the rollback phase, only that phase of the algorithm needs to be modified and thus receives minimal changes. The checkpointing algorithm is virtually identical to the one described in [VENK87].

## 3.5 MRLV System Model and Assumptions

The MRLV algorithm presented in this chapter assumes the following computational model:

- Tasks may be interconnected by one or more logical channels via ports.
- The underlying communication network is reliable i.e. messages are not lost.
- Messages need not be received by tasks in a FIFO manner.
- Machines on which tasks execute are reliable. This is since the algorithm is to be used in the context of distributed debugging rather than fault tolerance.

## 3.6 MRLV detection of Pre-Rollback Messages

The modification of the RLV algorithm for detecting pre-rollback messages is two-fold. It first involves a different means of determining the pre-rollback status of a message. Secondly,it involves modification of the protocol by which tasks establish valid (i.e. consistent) recovery lines.

### 3.6.1 Pre-rollback status determination

In the "vanilla" version of RLV, pre-rollback messages can only arrive at a task *between* rollback messages. The channel is said to be in a *cautious state* from the time it sends out a rollback message until the time it receives a rollback message from some task on the same channel. This principle is the basis for detection of pre-rollback messages. It is assumed that since messages arrive in FIFO order on a channel, the arrival of the second rollback message indicates that any messages in transit before the rollback would have been *flushed out* of the channel and therefore, no more pre-rollback messages could exist. In the context of a model such as the Mach model which cannot guarantee FIFO channels in the presence of multiple threads, this method of detecting pre-rollback messages is inappropriate. In this section, a new method of detecting pre-rollback messages based on *task incarnations* is proposed.

The concept of a *task incarnation* is basic to the new rollback algorithm.

**Definition 11** *A task is said to enter a new* **incarnation** *every time it rolls back its state to any previously created checkpoint (SIC or RC).*

**Definition 12** *A task is said to be* in *incarnation* 0 *if it has either not started its execution, or it has not performed any rollbacks to any previously created checkpoints (SIC or RC) since the start of its execution.*

**Definition 13** *A task is said to be executing* **in** *incarnation* $\alpha$ *if it has performed* $\alpha$ *rollbacks to any previously created checkpoint (SIC or RC) since the start of its execution.*

A task's incarnation is independent of the incarnation of other tasks. Therefore different tasks may be in different incarnations at any given time. It should be noted that the meaning of incarnations presented here is similar to that presented in [STRO85] where a similar approach to detecting pre-rollback messages is used. However, in [STRO85] incarnation numbers are used along with message numbers and logs to detect pre-rollback messages whereas MRLV uses only the incarnation numbers.

The proposed modification to the algorithm makes use of a data structure called an *incarnation table* which allows a task to determine the current incarnation of a remote task instead of relying on the order of messages along a channel. This solution eliminates the need for a cautious channel state, and allows tasks/threads to communicate through as many channels as they need rather than on a one to one basis. Furthermore, unlike standard RLV, ordering of messages sent from one task to another need not be preserved in order for the algorithm to work. This is important for the implementation of the algorithm on a Mach based system.

### 3.6.2 Incarnation tables

The main purpose of incarnation tables is to allow a task to determine whether or not a message which has been received is an obsolete pre-rollback message.

Each task $T_i$ maintains an incarnation table $inc\_tab_i$ which is an $array[1..N]$ of integers, where $N$ is the maximum number of tasks allowed by the system. $inc\_tab_i[j]$ represents the current incarnation of task $T_j$ as known to $T_i$ [2]. Whenever a task

[2]This global knowledge is made available to all tasks in the rollback phase of the algorithm

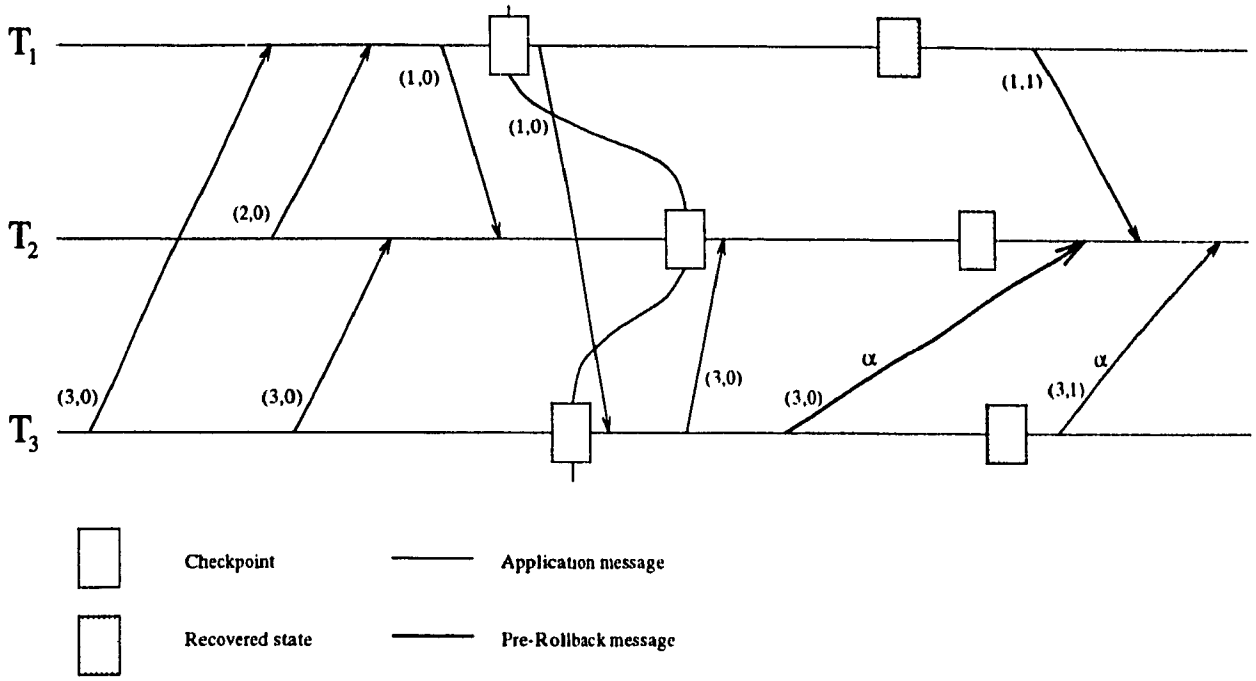


Figure 3.8: Use of incarnation numbers to determine pre-rollback status of messages

$T_j$ sends a message to any task, it appends the pair $(j, current\_inc_j)$ to the message, where $j$ is the unique task id and $current\_inc_j$ is the incarnation which $T_j$ was in when the message was sent. When task $T_i$ receives a message from task $T_j$, it compares the appended $current\_inc_j$ value with $inc\_tab_i[j]$. If $current\_inc_j$ is less than $inc\_tab_i[j]$, then the message is a pre-rollback message and it is discarded. It is discarded because the mismatch in incarnation numbers indicates that the message is from a previous *incarnation* of task $T_j$.

Consider the execution depicted in Figure 3.8[3]. Whenever a message is sent , the sender includes a tuple containing its own unique *task_id* and its current incarnation number. Message $\alpha$ is sent by $T_3$ during its 0 incarnation. However, the message is only received after $T_3$ has rolled back. Since $T_2$ would have the knowledge that $T_3$ is now in incarnation 1 and the incarnation number appended to the message is 0, $T_2$ knows that message $\alpha$ is pre-rollback.

explained in the next section.

[3]Rollback establishment messages are omitted from the figure for simplicity.

```
Algorithm Message_Send(remote_task,msg){
    include sender_id in msg
    include CCP_i vector in msg
    include this task's current_inc_i in msg
    send msg to remote_task
}
```

Figure 3.9: MRLV message send algorithm

### 3.6.3 MRLV message Send and Receive Algorithms

The original algorithms with respect to sending and receiving messages in the RLV scheme must be modified to accommodate the incarnations scheme previously discussed.

#### Modification of Message Send

The algorithm used by a task $T_i$ for sending messages in Figure 3.9 is used during normal execution of the program and during checkpointing. It is identical to the RLV algorithm in every respect except for the addition of the incarnations scheme.

#### Modification of Message Receive

The algorithm for receiving a message during normal execution is identical to that of the RLV algorithm except that provision is made to take into account the incarnations scheme. The algorithm performed by task $T_i$ upon receiving a message is shown in Figure 3.10.

## 3.7 MRLV Rollback protocol

The information contained in the incarnation tables of all tasks must be globally distributed so that it becomes a global knowledge. This is done during rollback since it is the only time that the incarnation information will change. Furthermore, during this time all tasks need to communicate with each other for determining whether or not they are part of the recovery line being rolled back.

```
Algorithm Message_Receive(msg) {
    if (msg.current_inc < inc_tab_i[msg.sender_id])
        discard message /* pre rollback message */
    else{
        /* accept message */
        For all tasks T_j{
            if (msg.RCP[j] > CCP_i[j]){
                create checkpoint < j,msg.RCP[j] >
            }
            else if (msg.RCP[j] < CCP_i[j]){
                store msg in all checkpoints < owner,number >
                where owner = j and
                  number > msg.RCP[j]
            }
        } /* end for */
    }
}
```

Figure 3.10: MRLV message receive algorithm

In RLV, when a task (*rollback initiator*) wishes to initiate a rollback, it broadcasts a rollback message on all of its outgoing channels and then performs a rollback to a previously saved state. It then continues its execution normally . Upon receiving a rollback message, if a task finds that it is part of the recovery line being rolled back, it also performs a rollback to a previous state and broadcasts the rollback message on all of its outgoing channels. Even if it is not part of the recovery line, it performs this broadcast. A task will accept messages *blindly* along a channel only after it has received another rollback message on that channel (The multiple receptions of rollback messages occur because each task broadcasts the rollback message on all of its outgoing channels). Otherwise, the received message may be a pre-rollback message and if so, it is discarded. This is depicted in Figure 3.11. Message $\alpha$ is discarded since $T_2$ has not yet received a rollback message from $T_3$. Upon receiving the rollback message from $T_3$, $T_2$ accepts the current (i.e. for the current incarnation) message $\alpha$.

### 3.7.1 Distribution of incarnation information

The above scenario is the method used by the vanilla version of RLV. In MRLV, the correct detection of pre-rollback messages is dependent on the distribution of incar-

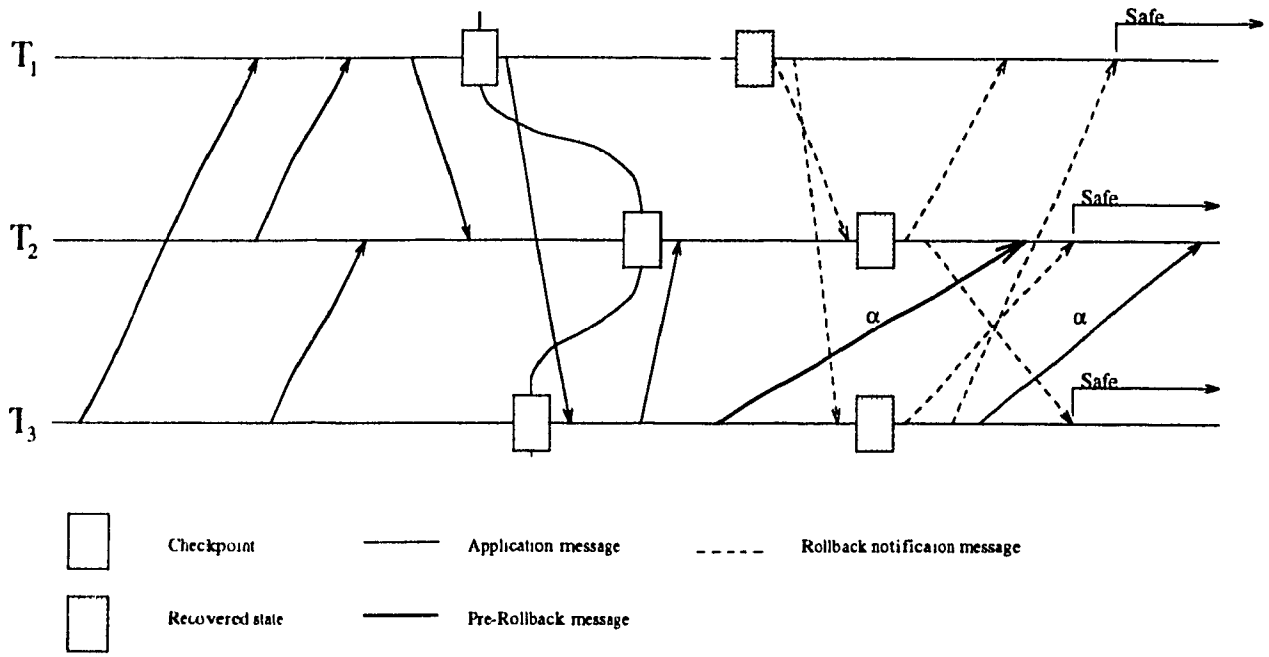


Figure 3.11: Rollback protocol in standard RLV

nation information to all tasks. In order to distribute the incarnation information, the algorithm is modified as follows: When a task $T_i$ wishes to initiate a rollback, it first increments $current\_inc_i$. It then sends a rollback message, which includes the id of the SIC checkpoint to which it is rolling back , to all tasks (it is assumed that $T_i$ has knowledge of all other tasks[4]). In addition to the SIC rollback information, it also includes the tuple $(i, current\_inc_i)$ .This is to inform all other tasks of the new incarnation that $T_i$ wishes to enter. The rollback initiator then performs a rollback to the state saved in its SIC, but it does not continue its execution , nor does it accept the receipt of any application messages. Instead, it waits until all tasks have replied to the rollback message. This reply is a rollback message including the sender's new incarnation number.

When a task $T_j$ receives a rollback message from task $T_i$, it updates $inc\_tab_j[i]$ according to the $current\_inc_i$ received in the rollback message. It then performs a rollback of its state (if it has previously created any checkpoint which forms part of

[4]This information is not specifically made available by Mach but it can be provided as will be shown in the architectural model of section 4.2.2

the recovery line), and broadcast a message which includes the tuple $(j, current\_inc_j)$. It then waits for a reply from all other tasks except $T_i$[5]. When $T_j$ receives a reply from some task $T_k$, it sets $inc\_tab_j[k] = current\_inc_k$.

Only after $T_j$ has updated all of its $inc\_tab_j$ entries, is it allowed to proceed with its execution. This guarantees that no messages are sent until all tasks have had a chance to update their incarnation tables, allowing a task to accurately verify whether or not a received message is a pre-rollback message or not based on *up to date* incarnation information. The execution in Figure 3.12 demonstrates the use of incarnation numbers and the new rollback protocol to discard pre-rollback messages. Message $\alpha$ is discarded when it is first received by $T_2$ since $T_2$ knows (because of the incarnation information received in the rollback messages) that $T_3$ should be in incarnation 1, but the tuple associated with $\alpha$ indicates that it is a message left over from incarnation 0. The restriction that no application messages are accepted[6] while a task is awaiting rollback messages, ensures that message $\alpha$ cannot be received until after $T_2$ has received all of its pending rollback replies.

### Update of incarnation information in saved messages

The incarnation scheme also requires a minor modification of the algorithm with respect to messages saved in checkpoints after a rollback has occurred.

Since the MRLV checkpointing algorithm is the same as that of RLV, messages may have to be stored in checkpoints. When messages arrive, they include the sender's incarnation number, so this incarnation number is automatically stored along with the message. This may cause problems upon rollback since when the saved messages in the checkpoint are restored after a rollback, they would contain the incarnation number of the sending task at the time of the saving of the message. This incarnation number may no longer be up to date at the time of the rollback and upon re-consumption by the user task, they may be mistaken for pre-rollback messages and discarded

[5]This is to ensure that a deadlock cannot occur.

[6]We cannot prevent messages from being received at a given port, however the term *accepted* here refers to the actual dequeuing of the message by the application task through the execution of a Mach message receive system call.

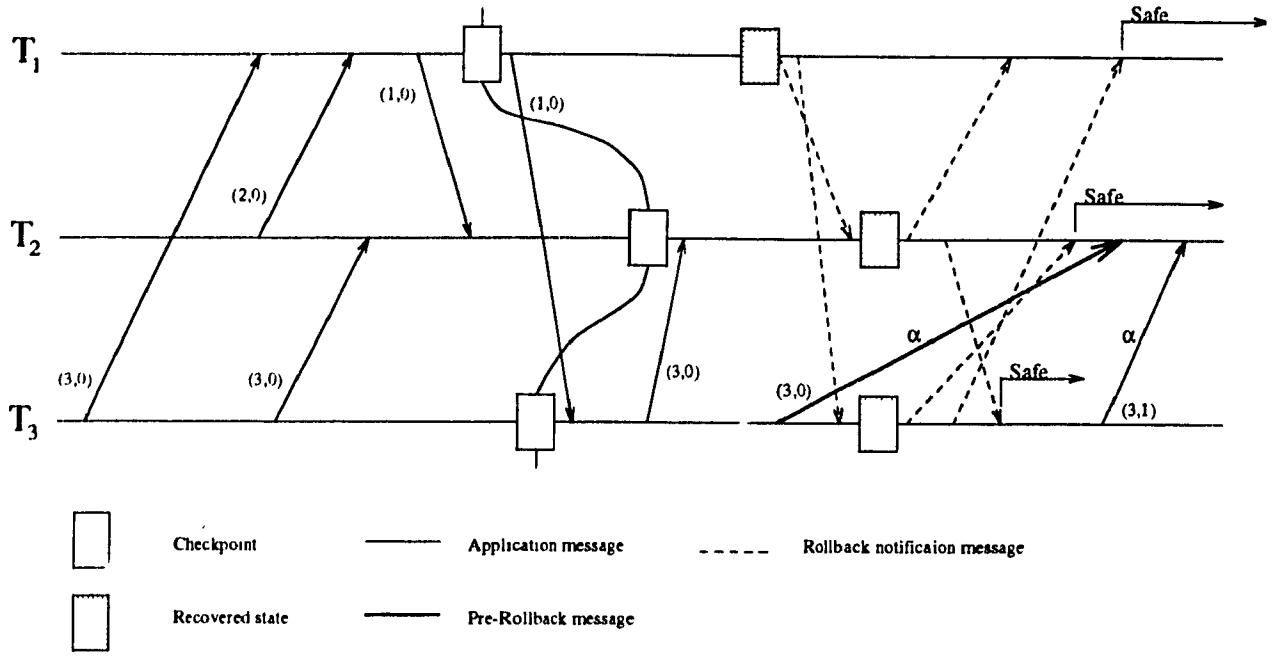


Figure 3.12: Rollback protocol in modified RLV

(since their incarnation numbers may be inconsistent with the current incarnation number value of the task which originally sent the message). Therefore, upon rollback incarnation numbers of all messages must be reset to the current value before being queued for re-consumption by the user task.

### 3.7.2 MRLV Rollback protocol algorithms

There are two basic algorithms[7] needed for the realization of the MRLV protocol discussed in the previous section. Unlike the algorithms in sections 3.6.3, these algorithms are completely different from the original RLV scheme.

#### MRLV Rollback Initiation algorithm

The algorithm shown in Figure 3.13 is used whenever a task $T_i$ wishes to initiate a rollback to a previously created SIC checkpoint $CP$. This causes the incarnation of the task to be incremented and the possible participation in the rollback of other

[7] A more detailed description of the algorithms can be found in Appendix A

```
Algorithm Initiate_Rollback(CP){
    disable application message acceptance
    increment current_inc_i
    rollback state to CP
    discard all messages saved in any checkpoint
        after the creation of CP
    reset value of CCP_i vector to that saved in CP
    purge all checkpoints created after CP was created
    /* broadcast rollback msg */
    for all tasks T_j {
        include current_inc_i in rollback_msg
        include unique_task_id in rollback_msg
        send rollback_msg to task T_j
    }
    /* update incarnation info */
    for all tasks T_j{
        wait for reply r from T_j
        inc_tab_i[j] = r.current_inc
    }
    update incarnation information in messages saved in CP
    restore messages saved in CP
    enable application message acceptance
}
```

Figure 3.13: MRLV rollback initiation algorithm

tasks in the system. The name of the message which is broadcast upon rollback is *rollback_msg*.

### MRLV Rollback Participation algorithm

The algorithm shown in Figure 3.14 is used by any task $T_i$ upon receipt of a rollback message $n$ from another task.

## 3.8 Analysis of MRLV with respect to RLV

### 3.8.1 Message complexity

Since only the rollback phase of MRLV is different from RLV, it's checkpointing phase is non-intrusive in the sense that it requires no extra messages to establish recovery lines. The information needed to establish recovery lines is still piggy-backed onto normal application messages.

```
Algorithm Participate_Rollback{
    disable application message acceptance
    update incarnation of task that sent rollback
        participation msg in inc_tab_i
    if n indicates that task should rollback {
        increment current_inc_i
        rollback state to CP indicated by n (n.CP)
        discard all messages saved in any checkpoint
            after the creation of n.CP
        reset value of CCP_i vector to that saved in n.CP
        purge all checkpoints created after n.CP was created
    }
    for all tasks T_j { /* broadcast rollback msg */
        include current_inc_i in n
        include unique_task_id in n
        send message n to task T_j
    }
    /* update incarnation info */
    for all tasks T_j (T_j ≠ original sender of rollback message n){
        wait for reply r from T_j
        inc_tab_i[j] = r.current_inc
    }
    if rollback occured in this task{
        update incarnation information in messages saved in CP
        restore messages saved in CP
    }
    enable application message acceptance
}
```

Figure 3.14: MRLV rollback participation algorithm

MRLV incurs no *extra* message overhead over the RLV algorithm. The "vanilla" RLV algorithm requires a broadcast send by all tasks during the rollback phase of the algorithm. This implies that there are $O(n^2)$ extra messages introduced into the system solely for the purposes of the rollback. MRLV also requires this broadcast for the same reasons but adds no other messages besides these. Therefore, there is no loss or gain in the number of messages during the rollback phase and the message complexity is still $O(n^2)$.

Although the number of messages needed by MRLV in the rollback phase is the same as in RLV, the size of normal application messages is larger than in RLV. In RLV a task need only include the $CCP$ vector in all of its outgoing application messages. In MRLV, a task must append the $CCP$ vector as well as its current incarnation number and unique identifier. In RLV the message size is dependent on the size of the $CCP$ vector which is in turn dependent on the number of tasks involved in the distributed computation. Let $k_1$ denote the size of each $CCP$ vector entry and $n$ denote the number of tasks involved in the computation. The extra information appended to each application message is $k_1 n$. Therefore, we say that the RLV message size overhead is $O(n)$. The MRLV messages are larger by a constant factor. Let $k_2$ denote the size of the current incarnation number and the unique identifier combined. The MRLV message size is also dependent on the size of the $CCP$ vector so the size of the extra information attached to a message in MRLV is $k_1 + k_2 n$. We can say that $k_1 + k_2 n \approx n$ for $n \gg 0$. Therefore the application message size overhead in MRLV is also $O(n)$.

Rollback messages are larger in MRLV since they must include the unique id of the sending task and its incarnation number whereas in RLV no information is required in the message. However, they are larger by a constant factor and therefore will not increase with the number of tasks.

### 3.8.2 Task Blocking

In RLV, when a task decides to initiate a rollback, it can simply change its state, broadcast the required messages, and continue with its execution without regards to

the state of the other tasks. In other words, there is no need to block any tasks during rollback. Tasks which are not directly involved in the rollback (i.e. those that do not have checkpoints which form a part of the recovery line), are not required to block in any way.

It can be seen in the rollback participation algorithm of MRLV (see section 3.7.2) that regardless of whether a task must actually rollback its state or not, it must wait for a reply from all other tasks[8] before it is allowed to continue its execution. Unfortunately, this means that all tasks must block their execution during *any* rollback regardless of whether or not the task has checkpoints which form part of the recovery line being rolled back. However, in the context of a distributed debugger we are not concerned with the progress of the computation but rather its correct rollback and recovery. Therefore this limitation is not important.

### 3.8.3 Advantages of MRLV

The message complexity of MRLV is not greater than that of RLV. Furthermore, MRLV can be used in environments such as Mach which support multiple threads within tasks and multiple communication channels (ports) between tasks whereas RLV cannot. In fact, the vanilla version of RLV cannot be used in any environment which supports multiple channels between tasks unless all messages on all channels between two tasks are collectively ordered. This is a very difficult thing to guarantee. RLV also cannot be used in any environment which does not guarantee FIFO ordering of messages on channels regardless of whether there are multiple threads involved.

Task blocking is clearly not desirable for a real time system, however in the context of distributed debugging or fault tolerant applications which are more concerned with safety than progress, the gains outweigh the losses. One of the main gains of MRLV is that it allows the algorithm to be implemented in a multi-threaded environment such as Mach which otherwise would be impossible. The elimination of a need for the concept of FIFO channels allows the recovery of more flexible and realistic applications than are allowed by vanilla RLV. In short, the vanilla RLV algorithm cannot be

[8] except for the one which originally sent it the first rollback message

mapped easily to accommodate the Mach model and a new strategy such as the incarnations of MRLV is needed to ensure correct checkpointing and rollback in such an environment.

The Mach model is not an isolated experimental model. Mach is readily available on machines produced by such manufacturers as DEC,IBM,SUN,Intel,NeXT, Hewlett Packard, and others. It has also been adopted as the base operating system for OSF's (open software foundation) operating system development [SILB92]. Therefore, the adaptation of checkpointing algorithms to environments similar to that of Mach is useful and important. Also, it is not the only operating system which supports the multiple port/thread model. For example, the CHORUS [ROAB90] operating system supports the same model.

# Chapter 4

# System Architecture for MRLV (Body and Soul Model)

*Whereas the body perishes, the soul is eternal.*
*- Hindu Philosophy*

This chapter discusses the basic architectural model, called *Body and Soul* , designed to facilitate the implementation of the MRLV algorithm in a distributed computing environment based on Mach. Although the MRLV algorithm has been designed to support multiple threads within a user task, the body and soul model is described in this chapter assuming a single user-defined thread of control per task. This is to simplify the explanation of the architectural design and can easily modified to support multiple threads [1].

## 4.1 Motivation for Body and Soul Model

In the body and soul model, a distinction is made between *system tasks* and *user tasks*.

**Definition 14** *A* **user task** *is any task whose existence is a direct result of the execution of a user-written distributed application program.*

**Definition 15** *A* **system task**, *is any task which is* **not** *a user task.*

[1] It should be noted that the CDB debugger also assumes a single user-defined thread per task.

System tasks include any tasks which are introduced by the checkpoint and rollback recovery system or an encompassing system software. For example, in the context of a distributed debugger , all tasks which form part of the application program being debugged (i.e. that have been specified by the programmer ) are considered user tasks. All tasks which are present for the functioning of the debugger (including all checkpoint and rollback recovery tasks, and the debugger task itself) are considered system tasks.

Several problems arise in the implementation of the MRLV algorithm in a Mach environment. The motivation for adopting the body and soul model is to provide the implementation with functionality that is otherwise not provided by the operating system, and to facilitate operations which must be performed by the algorithm. In order for MRLV to be implemented, the design must provide:

- Transparency of resource deallocation.
- Manipulation of a user task's incoming and outgoing messages.
- Access to sender task information.
- Control over execution of user tasks.
- Transparent Maintenance of incarnations information.

### 4.1.1 Transparency of resource deallocation

The MRLV algorithm is meant to be executed in an environment where communication is based on the concept of *ports* rather than *channels*. In Mach , ports are "perishable resources" which are kept within the kernel and are destroyed when the task which owns them terminates.

**Definition 16** *A user task's resources which are deallocated by the underlying operating system upon termination of the task are said to be* **perishable**.

In general, when a task terminates, the operating system kernel deallocates all of its resources (eg. ports), if the task has not explicitly done so. Deallocation causes the

view of perishable resources with respect to all other tasks to change. Therefore the following problem arises in the re-allocation of ports following the rollback of a task which has been terminated. Prior to a rollback, a task may have terminated causing its ports to be destroyed. Upon rollback, this task would have to restart its execution from the state saved in its checkpoint. However, since the task had terminated, all of the ports which it had previously used for communication with remote tasks no longer exist. New ports can easily be created by the user task upon rollback however if normal communication between all tasks is to be maintained transparently, knowledge (i.e. port rights) of these new ports would also have to be distributed among all other tasks in the system. A problem could arise that a remote user task $T_i$ sends a message to a port $p$ owned by user task $T_j$ after a rollback. If $T_j$ was in a *terminated state* prior to the rollback, port $p$ would have been destroyed. $T_j$ would be required to allocate a new port $p'$ which it would use as a replacement for $p$. Send rights to $p'$ would have to be sent to $T_i$ (and any other task which owned send rights to $p$ prior to the rollback) before any messages could be sent. Otherwise, an error would occur when $T_i$ sends a message to $p$ as this port no longer exists and, the port $p$ being sent to is unknown to the system.

Distributing new port rights after a rollback is very complicated since it entails the management of all port right information (eg. who owns send and receive rights to which port). By controlling a task's resources, and ensuring that its ports are never deallocated regardless of whether the task is alive or terminated, remote tasks can still send messages to a user task's port even after the user task has terminated without incurring an error. Therefore, after a rollback, re-allocation of ports is unnecessary.

In order to keep resources such as ports from being deallocated, there is a need for controlling this deallocation. This is accomplished by the body and soul model.

### 4.1.2 Sender Task Information

Both the RLV and MRLV models assume that all user tasks can communicate with all other user tasks. Furthermore, they assume that the receiver of a message always knows who the sender of that message is. Specifically, this knowledge is used

for broadcasting of rollback messages and identification of pre-rollback application messages. These assumptions do not fit well into many distributed operating system models [2]. In distributed operating systems which support a port-based communication model such as Mach, tasks are not automatically assigned any system-wide "unique id". Thus , there is no way to determine who the sender of a message is. Also, send rights to ports must be acquired and there is no way to perform a general broadcast of messages to "all ports". The body and soul model solves these problems by providing a means of identifying tasks in the system and allowing message broadcasts.

### 4.1.3 Manipulation of Messages

The MRLV algorithm requires the manipulation of messages. It must:

- **Append and inspect vectors:** MRLV uses CCP vectors and incarnation tuples to determine the creation of recovery lines. These structures must be appended to all outgoing messages, and inspected on all incoming messages.
- **Append and inspect incarnations information:** MRLV must append incarnation numbers to all of a task's outgoing messages and must inspect all incoming messages' incarnation numbers to determine their pre-rollback status.
- **Append and inspect sender task information:** The sending task identifier must be appended on all of a task's outgoing messages.
- **Save received messages:** MRLV requires some messages to be stored in checkpoints for re-use upon rollback. All saving of messages must be transparent to the user task.

The body and soul model allows the above operations to be performed transparently without affecting the user task's execution in any way. It allows the manipulation of the in-coming and out-going messages of any task.

[2] In particular, they do not fit into the Mach operating system model which is the chosen implementation platform

### 4.1.4 Task Execution Control

The body and soul model allows complete manipulation of a user task's resources and execution. The execution of user tasks must be controlled in such a way they can be suspended, resumed, or terminated whenever necessary for checkpoint creation or rollback.

### 4.1.5 Maintenance of Incarnations Information

Incarnation information of user tasks must be maintained transparently. The body and soul model provides a mechanism by which incarna' 'on information can be updated easily without disrupting the user task's execution.

## 4.2 The Body and Soul Model

The body and soul model is named as such because of the way in which entities within it can be viewed. A user task can be viewed as the *Body* of a computation. When a rollback occurs, the body enters a new incarnation and is unaware of its previous activity, or *previous life*. When the body dies, its perishable resources are kept alive by a *Soul* system task to allow them to be inherited by subsequent incarnations of the body. The soul can never die. Therefore, no matter what state the body is in, certain necessary details can be retained continually across several incarnations. The soul's presence is always transparent to the body yet it can control every aspect of the body's execution such as when it will die, when it will be reincarnated, when it will be suspended and when it will be resumed. This basic conceptual model is implemented through the use of the following entities:

- Controller task (Soul)
- Central Name Server (CNS)
- User task daemons

### 4.2.1 Controller Task (Soul)

The "control" of a user task's resources and execution can be achieved by adding another level of indirection in the communication between user tasks. This indirection is the fundamental philosophy behind the body and soul model. In a distributed computation, user tasks executing on remote processors communicate by message passing via a communication network. This is depicted in Figure 4.1. Referring to Figure 4.2 we see that a *controller* task is attached to every user task at the initialization time. The controller redirects all messages entering and leaving the user task transparently. As far as user tasks are concerned, they are sending and receiving messages to and from other user tasks. In fact, user tasks send and receive messages to and from their respective controller task and the actual remote communication is performed between controller tasks only. The controller acts as the user task's "soul". The redirection allows the controller to manipulate incoming and outgoing messages as outlined in section 4.1.3:

- **Append and inspect vectors:** The controller task allows information to be updated, inspected, and appended to all messages transparently since all manipulations are performed outside of the user task.

- **Append and inspect sender task information:** Each controller knows the unique id of all other controllers. Whenever a message is sent out, the controller appends its own unique id to the message. Furthermore, because it is aware of all other controller id's, a controller can infer who the sender of a particular message is. This knowledge is otherwise unavailable under Mach.

- **Maintaining incarnation information:** The controller appends incarnation information to all messages and inspects it upon receipt of messages from remote tasks before forwarding the message to its real destination.

- **Saving received messages:** The presence of the controller makes the storing of messages simple. When a message is received, if it is deemed by the MRLV algorithm that it should be saved, the controller stores a copy of the message in

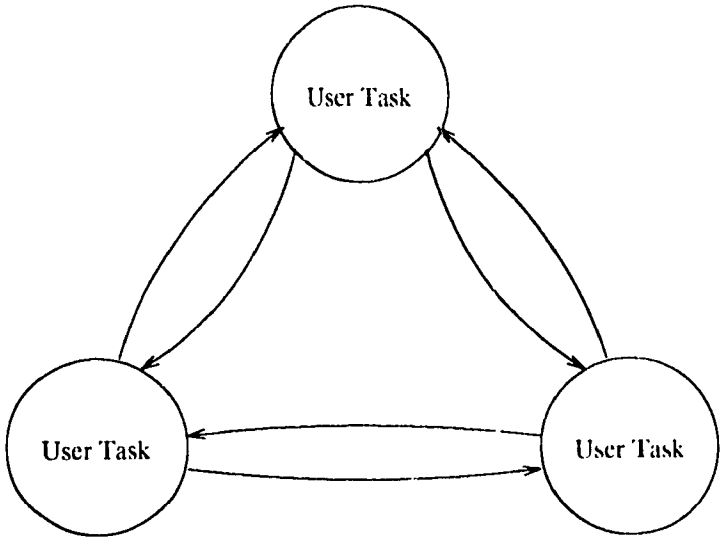


Figure 4.1: Message passing between user tasks

a checkpoint message buffer before redirecting the message to the user task. All saving of messages is transparent since it takes place outside of the user tasl

The controller task always remains active even if the user task which it is control ling has terminated or rolled back its execution to a previous state. This allows the "outside" view of the user task to remain consistent, and avoids the need for distri bution of new port rights upon rollback. The user task is "fooled" into thinking that it owns these resources so that its execution can proceed normally. The controller allocates perishable resources such as ports in its *own* address space. The controller task can be looked upon as the *Soul* of the computation since it never dies, and it contains the important resources which should not be lost upon a task's death. The controller also affects the user task's execution without the user task being aware of any higher level interference.

Because the user task does not contain any perishable resources, it is a dispensable entity which can be terminated at any time without affecting the overall computation. Thus, the user task is viewed as the *Body* of the computation.

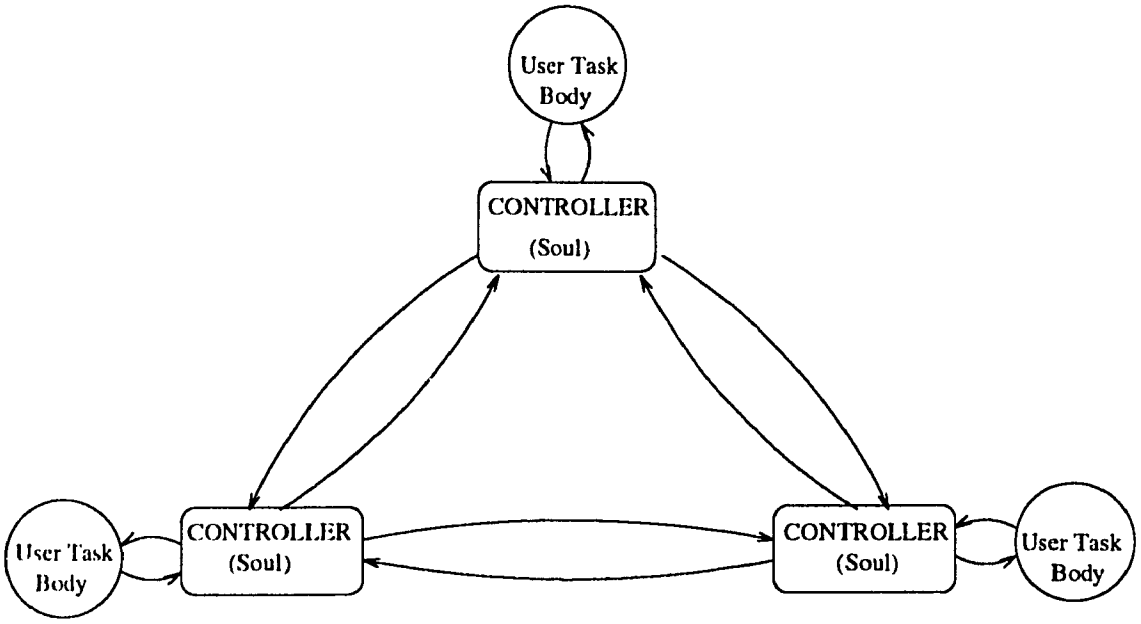


Figure 4.2: Controllers attached to user tasks

### 4.2.2 Central Name Server (CNS)

To allow global knowledge of sender task information such as controller port send rights and unique user task identifiers , a central name server called the CNS is used. Controllers collaborate with the CNS at the start of the user application program to gain knowledge about all other controllers.

The purpose of the CNS is to generate system-wide unique task id's and distribute them among all controllers in the system along with send rights to all controller ports. This relationship is depicted in Figure 4.3. Each controller is responsible for providing send rights of its own port to the CNS. The CNS is responsible for distributing this knowledge and assigning unique system-wide identifiers for each controller. The distribution of send right information and the assignment of unique task id's by the CNS allows controllers to identify the source of messages and to broadcast messages during rollback.

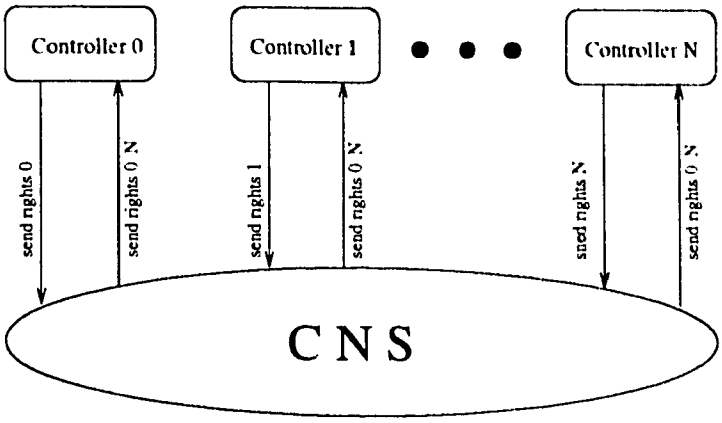


Figure 4.3: Distribution of send rights by CNS

### 4.2.3 User Task Daemons

The body and soul model uses the concept of user task daemons to aid the controller (Soul) in controlling the execution of the user task (Body). With the use of these daemons, the controller can dictate when the user task should be suspended, resumed or terminated. It can also rely on the daemons for saving and restoring the state information of the user task for checkpointing and rollback recovery. These daemons are actually Mach threads which are spawned in each user task at the initialization time of the application. They are:

- **Checkpoint Daemon**: This daemon is responsible for performing checkpoint ing duties on behalf of the controller task. There is one checkpoint daemon for every user task.

- **Rollback Daemon**: This daemon is responsible for performing rollback and recovery duties on behalf of the controller. There is one rollback daemon for every user task.

**Definition 17** *A* **user thread** *is any thread executing within the user task created by the application program* [3].

[3]For simplicity, a single user thread per user task has been assumed.

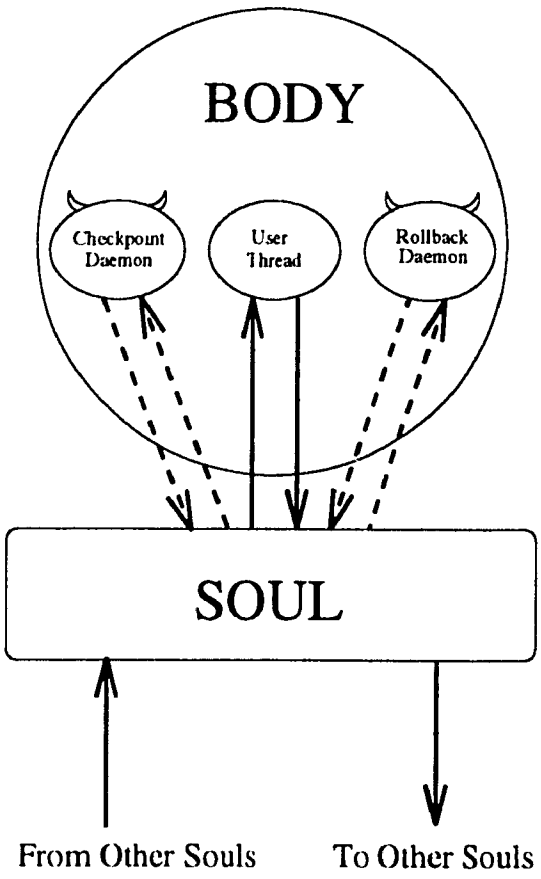


Figure 4.4: Checkpoint and rollback daemons

The checkpoint and rollback daemon threads execute concurrently with the *user thread* and are transparent to the application program. Both daemons remain suspended, or *dormant*, in the user task, and do not affect the user thread's execution until they are invoked by the controller (Soul). The controller invokes these daemons by sending them control messages which instruct them to perform predetermined functions on behalf of the controller.

When a daemon is invoked by the soul, it suspends all other threads and takes over the body's execution. The daemon *possesses* the user task until it has achieved its goals. In the case of the checkpoint daemon, the possession is temporary and control is always returned to the user thread eventually. During possession by the checkpoint daemon, the user task's virtual memory state and the register state of all

its threads are saved onto secondary storage in a checkpoint. Once this is done, the checkpoint daemon informs the soul that it has completed its possession by sending a control message, and lets the user thread regain control of the body. The daemon then becomes dormant awaiting further messages from the soul.

Unlike the checkpoint daemon, possession by the rollback daemon is "fatal" in the sense that it leads to a termination of the current execution of the user task and causes a new incarnation of the body (user task). Once invoked, the rollback daemon kills the user thread(s) and loads a previously saved checkpoint from secondary storage. This new checkpoint includes all threads of the task as they were when they were saved in a previous execution , or *previous life*, of the body. This includes the virtual memory and register state of the daemon thread's from the old life. Once the rollback daemon has loaded all necessary information into the task, it lets the new user thread take control and informs the soul (by way of a message ) that it should now control the *new* user thread. When the rollback thread completes its duties, all old daemon's are said to have been *exorcised* since they are forced to leave the body in order to allow new rollback and checkpoint daemons within the reincarnated body. The following sections describe the above entities which make up the body and soul model in a greater detail.

## 4.3 CNS Architecture

The main role of the CNS is to keep track of and/or distribute information which is needed by all controller tasks in the system. Send rights and unique user task id's must be made available somehow to all controllers by the CNS. We have two options.

- **Centralized scheme**: The CNS is involved throughout the execution of the application program and keeps track of the send rights of all tasks.
- **Distributed scheme**: The CNS is involved only at the initialization phase of the application program being executed and distributes all send rights amongst the tasks.

The distributed scheme is the one that has been adopted. The reason for this design decision are outlined in the following sections.

The design of the CNS simply involves saving information for later distribution to all controllers. Both the CNS and all controllers follow a simple protocol to allow the distribution of send rights to take place. Upon checking itself in to the CNS, a controller blocks waiting for a response from the CNS. The response is given when all the controllers have checked themselves in. It includes all information needed by the controllers to communicate with each other. The CNS is then no longer needed since controllers can now communicate directly with each other.

### 4.3.1 Role of CNS

Upon initialization of a user application program, the CNS task is created and it is notified [4], of how many controllers it should expect to be part of the application. It then remains idle awaiting messages from the controllers. These messages can be one of two types:

- **Controller Check-in message**: Each controller must check itself in to the CNS by way of a msg_rpc. i.e. a message send which is immediately followed by a blocking message receive. This forces the controller to block awaiting a response from the CNS. Once the CNS has received a check-in message from *all* controllers, it broadcasts a reply to all controllers. Included in the reply are send rights to all controllers' ports involved in the application along with their unique task id.

- **Die! message**: This is a notification message to the CNS which informs it that the application is terminated or that the CNS is no longer needed and that it should now exit.

When a controller is created, it is automatically given send rights which will allow it to establish communication with the CNS once the application starts. The

[4] This information is provided by the the application that encompasses the checkpoint and rollback recovery subsystem (eg the XCDB debugger)

```
Algorithm CNS{
    While TRUE{
        Receive message CNSmsg from any controller /* block */
        if CNSmsg.messageType is CONTROLLER_CHECKIN{
            save send rights of controller
            assign controller a unique id
            if all controllers have now been checked in{
            for all controllers C_i{
                send reply to C_i which includes its unique id and
                  send rights to all controllers
                }
            }
        }
        else if CNSmsg.messageType is DIE {
            exit loop
        }
    } /* end while */
}/* end Algorithm CNS */
```

Figure 4.5: Central Name Server (CNS) algorithm

algorithm performed by the CNS upon its startup is given in a pseudo code format in Figure 4.5.

### 4.3.2 Justification for Distributed scheme

This section discusses the reasons for choosing a distributed scheme over a centralized one in the design of the CNS.

#### Advantages of Distribution

The main advantages of distributing send rights are as follows:

- **Simple rollback**: The centralized CNS scheme complicates the rollback mechanism because of the fact that all information is kept in the CNS and therefore in order for a task to get information about another task, it must first ask the CNS. Furthermore, since a task must always know who the sender of a particular message is, a high overhead would be incurred during normal execution of the application by controllers requesting such information from the CNS. The

distributed scheme simplifies this and reduces all overhead since all necessary information is distributed among all controllers at the start of the application and from that point onwards, information need not be requested from an outside entity such as the CNS. This way, overhead is not incurred during the actual execution of the application as in the centralized scheme. This is the main reason why the distributed approach was adopted.

- **No single point of Failure**: In a fault-tolerant application context, the distributed scheme is more robust since the Central Name Server (which is the only single point of failure) is needed only during startup of all tasks. Therefore, during the actual execution of the application, failure of the CNS has no effect on the progress of the application.

### Disadvantages of Distribution

The main disadvantages of the distribution of send rights approach are outlined here:

- **Non-Transparency**: The biggest disadvantage with this approach is the fact that in order for send rights to be distributed among all controllers, the number of controllers must be known to the CNS. This requires the application programmer to specify the number of tasks beforehand. Furthermore, because of this, dynamic creation of tasks is difficult but not unobtainable. It involves a complex protocol to re-distribute send rights of the new task's ports to all other controllers transparently. If a centralized scheme had been used, this level of transparency could have been maintained since there would be no need to keep track of how many tasks are currently executing within the application. Tasks could check themselves in to the CNS and carry on with their execution without caring about send rights of other tasks' ports in the system. If a task needed send rights , it could access them indirectly through the CNS. Therefore, it would be easier for tasks to be spawned dynamically since it would not involve any redistribution of send rights.

- **Simultaneous startup of all tasks:**

  The need for synchronizing the start of a task is highly dependent on the design of the CNS. If the CNS is used throughout the execution as in the centralized scheme, then the task execution need not be synchronized. Once a task has been checked in, it can continue with its execution oblivious of the other task's status. If the CNS is used only at the beginning of the execution of the application program as in the adopted distributed scheme, then all tasks must be started at the same time. This is necessary since they must all be given information about each other prior to the start of the application in order to allow communication during the rollback of a task.

## 4.4 Checkpoint Daemon Architecture

The checkpoint daemon's sole responsibility is the creation of checkpoints. At the startup of an application program, a checkpoint daemon is spawned in every user task and it establishes communication with the controller of its respective user task. From then on, the checkpoint daemon does nothing other than wait for messages from the controller. Only when a message from the controller arrives, does the checkpoint daemon "possess" the user task.

The controller dictates *when* a checkpoint should be created. The controller is oblivious to the mechanics of *how* the checkpoint is actually created and it leaves this responsibility to the daemon. In this sense, there is a master/slave relationship between the controller and the checkpoint daemon in the user task.

If the controller decides that a checkpoint is required, it asks the checkpoint daemon to create it and store it on secondary storage. This request is in the form of a message containing the information which the daemon needs to know about the checkpoint to be created. In particular, the checkpoint daemon needs to know the ordinal number of the checkpoint to be created, and who the owner task is (i.e. that task's unique id). Once it has created and stored the checkpoint, the daemon becomes dormant until the controller decides that another checkpoint is required. The controller

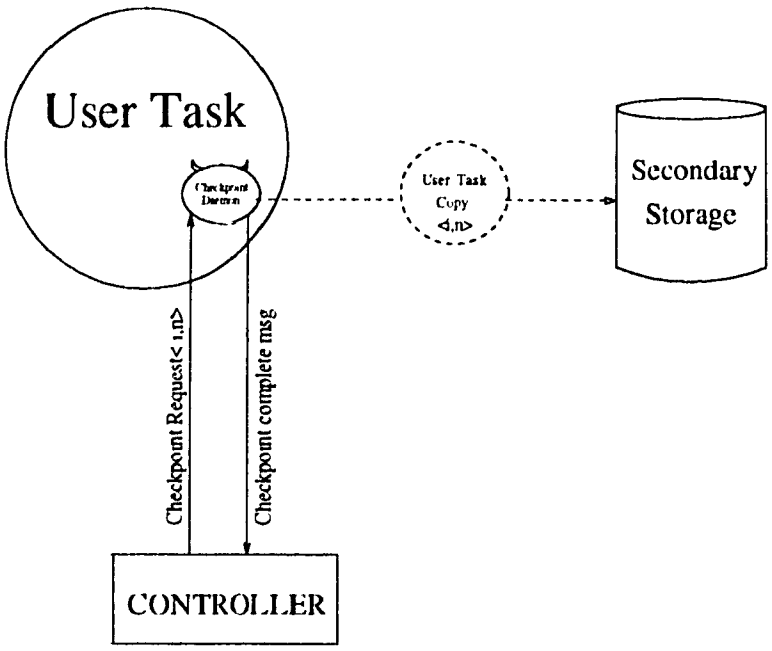


Figure 4.6: Relationship between controller task and checkpoint daemon

is responsible for keeping track of all necessary information about the checkpoint (eg. its identifier, its owner etc).

The relationship between the controller task and the checkpoint daemon is depicted in Figure 4.6. The controller sends a checkpoint creation request message to the checkpoint daemon asking it to save the current state of the user task with the id $< i, n >$. i.e. the current state will be saved in a checkpoint *owned* by task $T_i$ with ordinal number $n$. The controller then waits for a reply. When the daemon receives the message, it suspends all threads executing within the task (user thread and rollback daemon) and copies the current state of the user task to secondary storage making sure to label the checkpoint as requested by the controller (i.e. checkpoint $< i, n >$). Once the checkpoint creation is complete, the daemon informs the controller of the creation, returns control to the user thread, and becomes dormant again. The user thread is of course oblivious to the fact that it has been suspended during checkpoint creation and carries on as if it had never been interrupted.

### 4.4.1 Checkpoint creation issues

When the checkpoint daemon creates a checkpoint, it must be sure that the other entities within the task (i.e. the user thread and the rollback daemon) are not performing any functions which may bring them into an undefined state upon rollback. In particular, this could happen if either of the other entities are performing operations which are handled by the operating system at the time of the checkpoint creation.

The rollback daemon , like the checkpoint daemon, waits for messages from the controller while it is dormant. This message receive operation is one such operation that is taken care of by the underlying operating system. If a checkpoint is created by the checkpoint daemon while the rollback daemon is waiting for a message, then the outcome of the rollback daemon's message receive operation may be undefined when the task's state is rolled back to this checkpoint at a later time. This is because the actual *control* of the rollback daemon may be in the operating system kernel at the time of the checkpoint creation. That is, the kernel may be executing instructions on behalf of the rollback daemon and so it is executing outside of the user task space (see Figure 4.7). Therefore , the state of the user task at the time of checkpoint creation would not include the correct state of the rollback daemon. The solution to this problem is to send a special class of messages called *checkpoint warning messages* which inform the rollback daemon of an impending checkpoint creation. When a checkpoint warning message is received by the rollback daemon, it exits its message receive instruction (which will ensure that it is not executing kernel level instructions) and performs a null operation which will not affect the state of the task upon rollback (see Figure 4.8). When the checkpoint creation is finished, the rollback daemon restarts its message receive as if nothing had happened.

For the user thread, there are two cases to be considered:

- **SIC checkpoint creation**: The user task knows when a SIC is being created and therefore can easily be forced to perform "busy waits" or some other operation which would not affect the rollback to this checkpoint.

- **RC checkpoint creation**: As in the case of the rollback daemon, the only

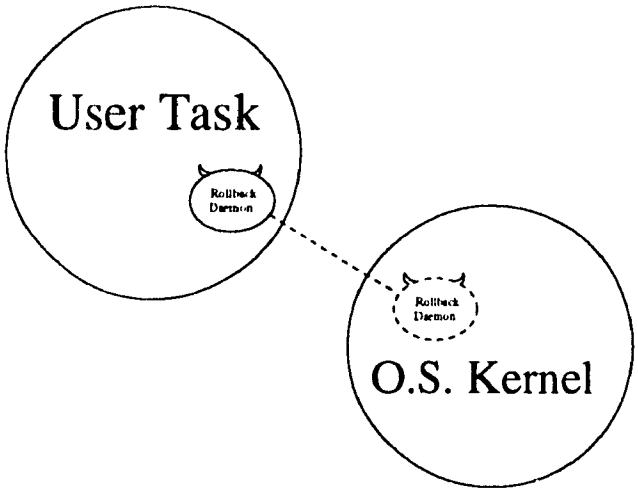


Figure 4.7: OS kernel executing instructions on behalf of the rollback daemon

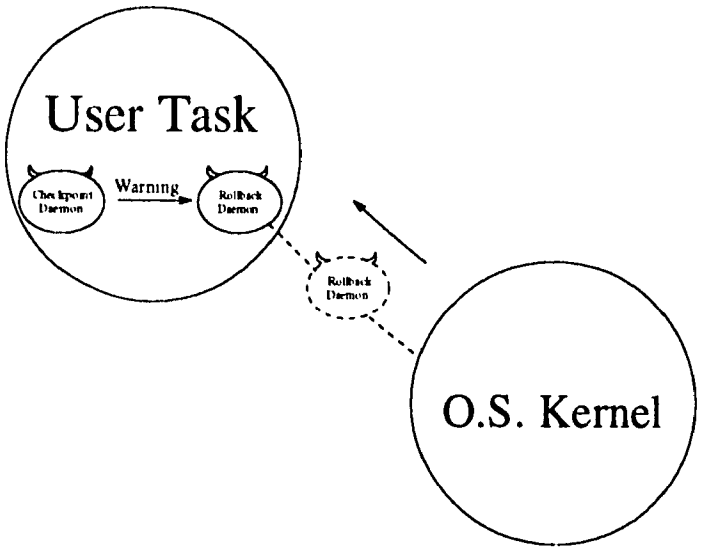


Figure 4.8: Warning message causes Rollback daemon to exit message receive

```
Algorithm Checkpoint_Daemon{
    Establish communication with controller task
    While TRUE{
        Wait for message CONTRmsg from controller
        Send Warning message to Rollback Daemon
        Wait for Warning to be received/processed by Rollback Daemon
        suspend Rollback Daemon
        suspend user thread
        save virtual memory of user task in
            checkpoint < CONTRmsg.owner, CONTRmsg.number >
        save register state of user task in
            checkpoint < CONTRmsg.owner, CONTRmsg.number >
        resume Rollback Daemon
        resume user thread
        reply to controller with checkpoint completion message
    } /* end while */
}/* end Algorithm Checkpoint_Daemon */
```

Figure 4.9: Checkpoint Daemon algorithm

important operation that could be in progress during response checkpoint creation is a message receive. This is so, because the only time that a response checkpoint is created is in response to a message receive operation. That is the only time, according to the MRLV algorithm ,that CCP vectors are checked to see whether or not a checkpoint should be created. In this case, the solution is to simply create the checkpoint while the user thread's message receive is being performed, and to kill and restart the operation transparently upon rollback.

The checkpoint daemon algorithm is given in Figure 4.9. Note that *CONTRmsg* is the checkpoint creation message sent by the controller when it wishes to inform the checkpoint daemon that a checkpoint should be created. Also, the user thread and rollback daemon are suspended before the state is saved to ensure that the state of the user task being saved in the checkpoint is not a "moving target" i.e. that the state is not changing while the checkpoint is being created.

## 4.5 Rollback Daemon architecture

The rollback daemon is the third entity which executes within the user task's space (the other two being the checkpoint daemon and the user thread itself). Its responsibilities are to restore the state of a task from a saved checkpoint and to change the user task's incarnation number upon rollback. Like the checkpoint daemon, it does not keep track of any checkpoint information, but rather acts as a slave to the controller which is responsible for the accounting of this information.

The rollback daemon's presence does not affect the normal progress of the user thread , as it simply lies dormant within the user task awaiting messages from either the controller ,or from the checkpoint daemon (warning messages).

When the controller decides that a rollback should occur in the user task which it is controlling, it sends a *rollback initiation* message to the rollback daemon. This message includes the $SICid$ (i.e. owner and ordinal number) of the checkpoint which it wishes the task to rollback to. The daemon upon receiving this message kills all threads running in the task, deallocates all virtual memory space , clears all registers, and then loads the state of the task from the checkpoint which was requested by the controller. The new state includes the memory state of all threads and their register contents. Once the new state is ready to be started, the daemon sends a *rollback completion* message to the controller and then terminates itself. Its termination is necessary since the state of the task which was loaded from the checkpoint also includes the state of the old rollback daemon as it was when the checkpoint was created. The rollback daemon algorithm is given in Figure 4.10. Note that $CONTRmsg$ is a control message sent either from the controller or the checkpoint daemon to the rollback daemon. This message instructs the rollback daemon to perform different actions depending on the source of the message.

The "new rollback daemon" is unaware of the rollback that has taken place, because the checkpoint to which the rollback has occurred was created while the "old rollback daemon" was performing a busy wait. Therefore, upon restart, the new rollback daemon is in a state where it has just completed its busy wait, and will restart

```
Algorithm Rollback_Daemon{
    Establish communication with controller task
    While TRUE{
        Wait for message CONTRmsg from controller
            or Checkpoint Daemon
        if (CONTRmsg is warning from Checkpoint daemon){
            busy wait until checkpoint creation complete
        }
        else if (CONTRmsg is rollback from controller){
            suspend Checkpoint Daemon
            suspend user thread
            deallocate all of user task's virtual memory
            clear all registers
            restore virtual memory of user task from
                checkpoint < CONTRmsg.owner, CONTRmsg.number >
            restore register states of user task's threads from
                checkpoint < CONTRmsg.owner, CONTRmsg.number >
            increment user task incarnation
            restart Checkpoint Daemon
            restart User thread
            reply to controller with rollback completion message
            restart Rollback Daemon
            Die.
        }
    } /* end while */
}/* end Algorithm Rollback_Daemon */
```

Figure 4.10: Rollback Daemon algorithm

the main loop without even knowing that it had been rolled back.

## 4.6 Controller (Soul) architecture

The controller is the main entity in the Body and Soul model. It controls every aspect of the execution of the user task . In particular, it is responsible for the following:

- It decides when a checkpoint should be created, and when a rollback should take place by following the MRLV algorithm.
- It is responsible for keeping track of any information which is pertinent to all checkpoints which have been created for the user task which it controls.

- It decides when a task should be suspended, resumed, or terminated for checkpoint and rollback purposes.

- It controls all incoming and outgoing messages of the user task which it controls.

Unlike the checkpoint and rollback daemons that execute as threads within the application program's user tasks, the controller task is a system task which executes independent of the user task. It controls the user task by monitoring its messages and communicating with the checkpoint and rollback daemons. The controller is structured as a *server* which performs actions depending on the receipt of different classes of messages. These message classes are based on the source of the message. The classes are:

- **User application message class**: These messages are simply application messages which are redirected through the controller from the user tasks. This sort of redirection is one of the main responsibilities of the controller.

- **Checkpoint daemon control message class (CDM)**: Once checkpoints have been created, the checkpoint daemon responds with a checkpoint daemon control message which the controller interprets as an acknowledgment that a checkpoint has been created.

- **Rollback daemon control message class (RDM)**: These messages are similar to the checkpoint daemon control messages. They are sent by the rollback to the controller to acknowledge that a requested rollback has been completed.

- **User control message class (UCM)**: User control messages are messages which have been sent to the controller by the user task transparently. They are sent in response to important events which must take place within the user task such as system calls.

- **Controller control message class (CCM)**: These are messages which have been sent from remote controllers.

All messages , with the exception of user application messages, are sent to the controller by the various entities through the use of a *controller notification* port. The controller notification port is a port owned by the controller on which it receives the various control messages which dictate its behaviour. This is the port whose send rights are made available to other controllers through the use of the central name server (CNS).

The controller is the engine which actually executes the MRLV algorithms. The rollback and checkpoint daemons only aid in the mechanics of checkpoint creation and rollback.

### 4.6.1 Establishment of communication by controller

Upon startup of a distributed user application program, all user tasks are suspended until the controller has had time to set up. When the controller is started, it establishes communication between the checkpoint daemon and the rollback daemon which have been spawned transparently in the user task. It then checks itself into the CNS and waits for a reply. The reply contains the unique system-wide id for the user task which is being controlled, and send rights to all controller notification ports.

Once all needed communication links have been set up, the controller allows the user task to be resumed. Communication between the user task and the controller then occurs transparently through the sending of UCM's. Special communication between controllers will occur through the sending of CCM's. The different types of UCM's and CCM's and their function are now described in more detail.

### 4.6.2 User control messages (UCM's)

UCM's are messages which request the controller to perform important functions on behalf of the user task. The user task sends these messages transparently. For the most part, these UCM's are sent from the user task as a result of a modified system call. The different types of UCM's are:

- **Port Allocation UCM**: sent as a result of a port allocation system call.

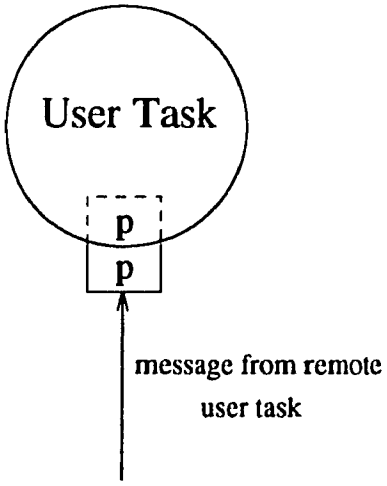


Figure 4.11: Normal port allocation by user task

- **Port Lookup UCM**: sent as a result of a send rights lookup system call.
- **Message Send UCM**: sent as a result of the user task performing a message send system call.
- **Message Receive UCM**: sent as a result of the user task performing a message receive system call.
- **SIC Creation UCM**: sent as a result of the user task initiating the creation of a MRLV SIC checkpoint.

### Port allocation UCM

Under normal execution, when a user task wishes to allocate a port $p$, it performs a Mach port allocation system call. The user is then allocated a new port by the operating system on which it can receive messages which have been sent from remote user tasks. The user knows the port by the name $p$ (see Figure 4.11).

The controller keeps the user task's view of the port allocation system call consistent with that described above. However, in reality ports are allocated in such a way as to allow redirection of messages which were originally directed to the user task from some remote task.

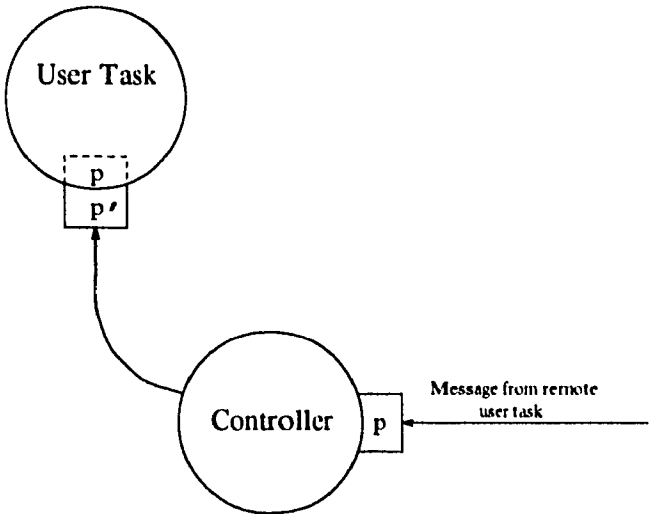


Figure 4.12: Redirection of messages by controller via port stealing

When the user task performs a port allocation system call, it actually allocates a port through a normal system call, and then sends a port allocation UCM message to the controller. This message contains the identifier $p$ of the port which it has allocated. Upon receiving the message, the controller "steals" port $p$ from the user task for itself. This is done by allocating a new port $p'$ in the user task, which the user task knows under the name $p$, and extracting the receive rights of the real $p$ into the controller task. From this point on when the user task performs a message receive on what it knows as port $p$, it is actually performed on port $p'$. This way, the controller can perform a message receive on $p$ and forward the message after it has finished processing it to port $p'$ where it is received by the user task (see Figure 4.12). The algorithm performed by the controller whenever it receives a port allocation UCM is given in Figure 4.13.

### Port Lookup UCM

User tasks send messages to each other by sending messages to ports to which the remote tasks have receive rights. In order to do so, the sending task must own send rights to that port. In the Mach model, one way of acquiring send rights is through the use of a server called the `netmsgserver` [SAJA86].

```
Algorithm Controller_Port_Allocation{
    create new port p' to be inserted into user task
    steal port UCM.allocated_port from user task
    insert port p' into user task with name UCM.allocated_port
} /* end Algorithm Controller_Port_Allocation */
```

Figure 4.13: Controller port allocation algorithm

Like port allocation, obtaining send rights for a port through the **netmsgserver** is done using a system call. The user task specifies the name of the port for which it wishes to acquire send rights and the system call returns send rights to that port.

In order for the controller to intercept and forward messages sent from the user task to remote tasks, the controller must own send rights to all ports which the user task may want to send messages to. Since send rights to those remote ports are acquired by the user task with a port lookup system call, the controller must "steal" the send rights when they perform this system call.

When the user task performs a port lookup system call, it first calls the port lookup system call and then transparently sends a port lookup UCM to the controller. This UCM contains the name of the port $p$ whose send rights have been looked up by the user task through the **netmsgserver**. Upon receiving the UCM, the controller allocates a new port $p'$ in itself. It then "steals" the send rights to port $p$ from the user task and inserts send rights to port $p'$ owned by the controller in the user task. The controller makes sure that the user task knows port $p'$ by the name $p$. When the user task sends messages to what it knows as port $p$, the messages are actually sent to port $p'$ which is owned by the controller. The controller can then process the message as it pleases and forward the message to the real port $p$. The algorithm performed by the controller when its user task wishes to perform a port send rights lookup is given in Figure 4.14

```
Algorithm Controller_Port_Lookup{
    allocate new port p' in controller
    extract send rights to port UCM.port_looked_up from user task
    insert send rights to port p' with the name
    UCM.port_looked_up into the user task
} /* end Controller_Port_Lookup */
```

Figure 4.14: Controller port send right lookup algorithm

### Message Send UCM

In order to perform necessary functions needed by the MRLV algorithm, all messages that are sent from the user task are intercepted by the controller. These message interceptions follow the algorithm described in section 3.6.3.

When a user task wishes to perform a message send system call, it actually performs two message sends. The first message send is the normal application message which it sends to what it perceives as the remote port. Since the port send rights are stolen upon lookup, the user task actually sends the message transparently to a corresponding port in the controller instead of the remote port it intends the message for. The second message is a message send UCM. This UCM informs the controller that the user task has sent an application message and that it is now waiting on the port whose send rights had been "stolen" by the controller during the port lookup. The message is received on this port and then manipulated as dictated by the MRLV algorithm. Once the controller has finished processing the message, it forwards it to the "real" remote port. This interception of message sends is depicted in Figure 4.15. The user task is "fooled" into thinking that it is sending message $\alpha_1$ to the remote port $p$. In fact, it sends $\alpha_1$ to port $p'$ which is in the controller. The user task then sends a UCM which , among other things, includes the name of the port that $\alpha_1$ was intended for. When the controller receives the UCM, it knows that a message is waiting on port $p'$ since a logical mapping between $p$ and $p'$ has previously been established at the time of the port lookup. The controller receives $\alpha_1$, augments it as needed by MRLV and sends the new message $\alpha_1'$ to the real remote port $p$ to which

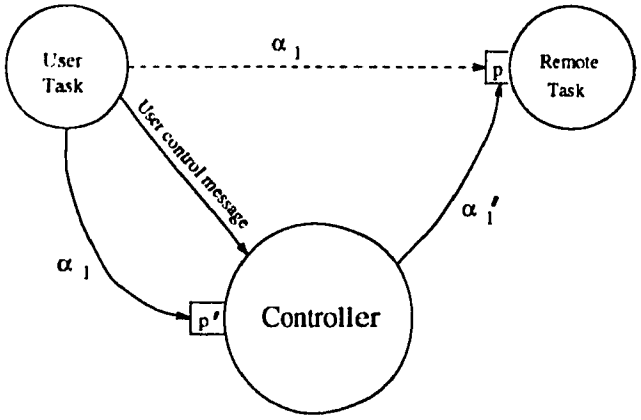


Figure 4.15: Interception of message sends by controller.

```
Algorithm Controller_Message_Send{
    find controller port p' corresponding to UCM.remote_port
    /* message interception */
    receive application message α from user task on port p'
    perform any needed MRLV functions related to message send
    find remote task port p corresponding to UCM.remote_port
    forward message α to port p
}/* end Algorithm Controller_Message_Send */
```

Figure 4.16: Controller message send algorithm

the controller owns send rights. The algorithm performed by the controller upon receipt of a message send UCM is given in Figure 4.16.

### Message Receive UCM

Messages between user tasks must be intercepted not only by the sending user task's controller, but also at the receiving end by the receiving user task's controller. The method by which received messages are intercepted is similar to that used at the sending end. When a user task wishes to perform a message receive on one of its ports $p$, it actually sends a message receive UCM to its controller and then performs a message receive on the port which it believes to be $p$. In fact, since send rights to $p$ have been stolen at the time of its allocation and replaced with another port $p'$, the user task actually performs a message receive on port $p'$. The UCM which the

```
Algorithm Controller_Message_Receive{
    find controller port p corresponding to UCM.receive_port
    /* message interception */
    receive application message α from remote task on port p
    perform any needed MRLV functions
    find user task port p' corresponding to UCM.receive_port
    forward message α to user task port p'
} /* end Algorithm Controller_Message_Receive */
```

Figure 4.17: Controller message receive algorithm

user task sends to the controller contains, among other things, the name of the port which it wishes to receive a message on, namely $p$. Upon receiving the UCM, the controller performs a message receive on the real port $p$, manipulates the received message as dictated by the MRLV receive algorithm given in section 3.6.3, and then forwards the application message to the user task by sending it to port $p'$ on which the user task has been suspended awaiting a message. The algorithm performed by the controller upon receiving a message receive UCM from the user task is given 4.17 Note that *UCM.receive_port* represents the name of the port which the user task wishes to receive a message on. It should also be noted that if, upon receipt of an application message, the controller sees that an RC checkpoint creation is pending by its user task, it will communicate with the checkpoint daemon in a fashion similar to that described in the next section on SIC creation.

### SIC creation UCM

As in the RLV algorithm, SIC checkpoints in MRLV are initiated by the user task. When a user task wishes to create a SIC, it initiates the creation by sending a SIC creation UCM[5] to its controller. While waiting for the checkpoint to be created, the user thread of the user task goes into a busy wait until further notice. This message is simply a flag that indicates to the controller that the user task wishes to save its

[5]SIC creation notification messages are classified under UCM's for simplicity. However, these messages could also be treated as CCM's since they could easily be sent from the system's encapsulating application rather than the user task itself.

```
Algorithm Controller_Create_SIC{
    send checkpoint creation message < user_task_id, CCP[user_task_id] >
        to checkpoint daemon
    block waiting for a response CDM
    update any necessary information as dictated by MRLV
    allow user task to resume normal operation
}/* end Algorithm Controller_Create_SIC */
```

Figure 4.18: Controller SIC creation algorithm

state immediately.

Upon receipt of a SIC creation UCM, the controller communicates with the checkpoint daemon requesting that a checkpoint be created as described in section 4.4. The controller then waits for an acknowledgement CDM message from the checkpoint daemon indicating that the checkpoint has been completed. Before resuming normal operation, the controller updates all necessary information which it is responsible for keeping track of such as updating the CCP vector entry for the user task and keeping track of the checkpoint etc. When everything is updated, the controller allows the user task to exit its busy wait and resume normal operation. The algorithm performed by the controller task upon receipt of a SIC creation UCM is is given in Figure 4.18.

### 4.6.3 Controller Control Messages (CCM's)

Controller control messages (CCM's) are always sent from one controller to another controller and always pertain to some aspect of the rollback of the system state. The two types of CCM messages are:

- **Rollback initiation CCM's**
- **Rollback participation CCM's**

#### Rollback initiation CCM

In the MRLV rollback algorithm, a user task initiates a rollback of the system state by rolling back to one of its self-induced checkpoints (SIC's). In the body and soul

```
Algorithm Controller_Rollback_Initiation{
    disable application message acceptance
    increment current_inc_i
    send message to rollback daemon informing it that it should
        rollback to checkpoint < user_task_id, CCM.ordinal_number >
    wait for rollback completion RDM from rollback daemon
    for all controller tasks C_j {
        include current_inc_i in rollback message n
        send rollback message n to task C_j
    }
    for all controller tasks C_j {
        wait for a reply r from C_j
        inc_tab_i[j] = r.current_inc
    }
    reset information (CCP update,incarnation in messages etc.)
    enable application message acceptance
} /* end Algorithm Controller_Rollback_Initiation */
```

Figure 4.19: Controller rollback initiation algorithm

model, the rollback is initiated by the sending of a Rollback initiation CCM to the controller. In the current implementation, this CCM is sent by the encapsulating application (i.e. the debugger). However, in some other implementation, it could be sent from the user task itself. The CCM includes the ordinal number of the SIC checkpoint created by the user task which should be rolled back to.

Upon receipt of a rollback initiation CCM, the controller performs the rollback initiation algorithm described in section 3.7.2 i.e. application messages are disabled, rollback participation messages are broadcast, etc. However, it relies on the rollback daemon for the actual mechanics of the rollback. this is done by sending a message to the rollback daemon informing it that a rollback should occur to the requested checkpoint. It then waits for a reply in the form of a rollback completion RDM from the rollback daemon indicating that the new state is ready to execute. The algorithm performed by the controller of a task $T_i$ upon receipt of a rollback initiation message is given in Figure 4.19 [6].

[6]Note that the complete algorithm is given since the algorithm in section 3.7.2 must be split up in order to present it properly.

```
Algorithm Controller_Rollback_Participation{
    disable application message acceptance
    update incarnation in inc_tab_i for sender of n
    if n indicates that task should rollback {
        increment current_inc_i
        send message to rollback daemon informing it that it should
            rollback to checkpoint < n.owner, n.number >
        wait for rollback completion RDM from rollback daemon
    }
    /* broadcast rollback msg */
    for all controllers C_j {
        include current_inc_i in message n
        send message n to controller C_i
    }
    /* update incarnation info */
    for all controllers C_j (C_j ≠ original sender of rollback message n){
        wait for reply r from C_j
        inc_tab_i[j] = r.current_inc
    }
    if rollback occurred in this task{
        reset necessary information (eg. restoring messages, CCP etc.)
    }
    enable application message acceptance
} /* end Algorithm Controller_Rollback_Participation */
```

Figure 4.20: Controller rollback participation algorithm

### Rollback Participation CCM

The second type of CCM which the controller can receive is a *rollback participation* CCM. This CCM is the same as that described in the rollback protocol of MRLV (see section 3.12). Upon receipt of such a CCM, a controller performs the MRLV rollback participation algorithm as described in section 3.12. If, according to the algorithm, the controller must rollback the state of its user task, it sends a rollback initiation message to the rollback daemon. It then waits for the rollback daemon to complete the rollback and send a rollback completion RDM. Thus the algorithm performed by the controller of a task $T_i$ upon receipt of a rollback participation message $n$ is given in Figure 4.20.

## 4.7 Problems with Body and Soul

The Body and Soul model for the architecture of an MRLV-based checkpoint and rollback recovery system described in this chapter introduces new problems which are not present when the MRLV algorithm is viewed at the conceptual level. This section describes these problems and solutions for them in the context of the body and soul model.

### 4.7.1 Second Level Pre-rollback Messages

Both the RLV and MRLV algorithms take specific measures to detect the presence of *pre-rollback* messages sent between user tasks. Unfortunately, the indirection imposed by the presence of the controller task with respect to messages , adds another level of pre-rollback messages which would otherwise not be present. These messages are termed *second level pre-rollback messages*[7].

If tasks are "uncontrolled", then pre-rollback messages can only occur between user tasks, and both the vanilla RLV and the MRLV algorithms will detect this scenario by using the *cautious state* and *incarnations* schemes respectively. However, because of the presence of the controller, pre-rollback messages must now also be considered between the user task and its controller as well. i.e. a message received by the controller from its user task may be from a previous incarnation and has reached the controller's port in the time between the user task's death and its reincarnation into a previous state. This scenario is depicted in Figure 4.21. Message $p$ reaches the soul after the old body has died.

Similarly, messages sent from the controller to the user task may be second level pre-rollback messages also. The controller task is always forwarding messages from remote controllers to the user task. The situation may arise where the controller task has forwarded a message to the user task which only arrives *after* the user task has been rolled back and has entered a new incarnation. Therefore, the controller must

[7]Note that there cannot be any more levels of pre-rollback messages since there is only one level of indirection.

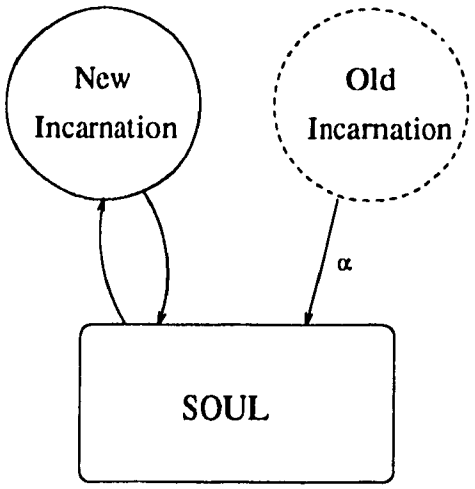


Figure 4.21: Second level pre-rollback messages between Body and Soul

have some method of determining whether or not messages it receives from the user task which it is controlling are pre-rollback and vice versa.

The idea of incarnations can be easily extended to solve the second level pre-rollback message problem. In this solution, both user tasks and controllers keep track of which incarnation they are in. In the user task, the incarnation value is called *body_incarn* and in the controller task it is called *soul_incarn*. After any rollback , the incarnation numbers are incremented in both the body and soul.

Whenever a message is sent from controller to user task (eg. port allocation UCM message), the body transparently appends *body_incarn* to it. Upon receipt of the message, the controller checks to see that *body_incarn* is equal to its current *soul_incarn*. If these two values are not equal, then the message is a second level pre-rollback message and is discarded. Similarly, controllers always append their *soul_incarn* value to all application messages which are sent to its user task. When the user task receives an application message which has been forwarded by its controller, it transparently checks to see that its *body_incarn* value matches that of the *soul_incarn* value in the forwarded message. Again, if these values do not match, then the application message is discarded.

### 4.7.2 Deadlock Between Controllers

According to MRLV, during a rollback, user tasks are forced to block until their controllers have received incarnation information from all other controllers. The actual waiting for incarnation messages may cause a deadlock to occur.

In general, when any (controller or user) task performs a message receive operation, it blocks until a message has been queued on the port on which it is attempting to receive a message. Because of the fact that the controller notification port is separate from the user ports (i.e. the ports used by the controller to receive messages from other user tasks through their controllers), and Mach message receives will block a task on a port until a message arrives, the following situation may occur. Referring to Figure 4.22, at time $t_1$ task $T_2$ initiates a rollback and sends rollback messages $\alpha_1$ and $\alpha_2$ to tasks $T_1$ and $T_3$ respectively. However, at time $t_2$ task $T_3$ is waiting for a message on one of its user ports (shown in bottom part of figure). Because it is blocked waiting for a message on a user port, it will not receive message $\alpha_2$ even though it has been queued on its controller notification port. Therefore, it will not be informed of the rollback until it is unblocked by the receipt of a message on its user port. Meanwhile $T_1$ and $T_2$ are both awaiting a user incarnation update (rollback) message from $T_3$. However, this message will never arrive since $C$ is blocked on its user port and is not aware of the rollback which has taken place while it was waiting for a user application message. Therefore, the whole system is deadlocked.

This problem would not arise in the vanilla RLV algorithm since upon a rollback, the controllers would not have to wait for a response from all other controllers before continuing.

The solution to this problem is quite simple. When a user task requests that a message receive be performed by the controller, instead of blocking on the user port awaiting a message from a remote task, the controller will loop inspecting both its controller notification port *and* the user port. A blocking message receive will only occur when there is actually a message queued on the port. This way the controller will never be blocked waiting for a message. Both ports are inspected and the first

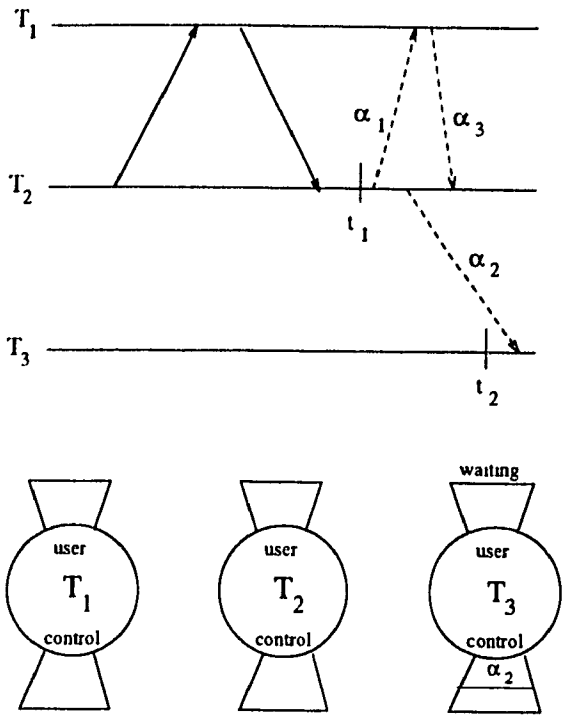


Figure 4.22: Deadlock caused by incarnation update waiting

message to arrive on either of the ports is processed. If a message arrives on the user port first, then it is processed normally. If a message arrives on the controller port first, then it must be a rollback message (since the user task is blocked waiting for a response from the controller, it cannot send any more control messages therefore the control message must have come from a remote task). This way deadlock cannot occur because rollback messages will always be processed regardless of the state of the receiving controller.

## 4.8 Discussion of Body and Soul model

### 4.8.1 Reusability of Body and Soul entities

The design can be used to simplify :

- implementation of new checkpoint and rollback recovery algorithms

- use of system as a black box to be inserted into different applications (debuggers etc.).

The following is a brief list of what each entity in the Body and Soul architecture can be used for:

- **Controller (Soul)**: reusable as a general purpose task monitoring device.
- **Central Name Server**: reusable as a general information distributor in any distributed application.
- **Checkpoint Daemon**: can be reused by new checkpoint and rollback and recovery algorithms as a black box to save the virtual memory and register state of any task to disk.
- **Rollback Daemon**: Given any task, this module can be used as a black box to load the state of a previously saved task into he space of a currently executing task. Note both the checkpoint daemon and the rollback daemon are extremely useful for the implementation of other checkpoint and rollback recovery algorithms.

### 4.8.2 Advantages of the model

Although the body and soul architectural model is designed in the context of Mach, it is generic in the sense that it can be applied to the implementation of MRLV on any distributed platform which is based on a port communication model. Furthermore, the design is not restricted to the MRLV algorithm, but can be adapted to fit other checkpoint and rollback recovery algorithms.

The body and soul architecture has been designed specifically for the implementation of the checkpoint and rollback recovery subsystem of the CDB distributed debugging system which will be discussed in more detail in the next chapter. However, the model's entities can be collectively looked upon as an independent checkpoint and rollback recovery subsystem.

The advantages of using the Body and Soul architecture:

- Total control over user tasks
- Preservation of resources upon termination of tasks
- Hiding of Mach resource allocation/deallocation
- Re-usability for different checkpointing algorithms.
- Less time involved during rollback due to simplicity re-allocation of resources

By adopting the *Body and Soul* model to the MRLV checkpoint , rollback and recovery algorithm, it is possible to control tasks executing in the Mach environment more easily. One drawback of this model is the fact that the redirection of messages created by the presence of the controller task necessarily increases message traffic almost two-fold. Unfortunately, this is the price that must be paid in order to implement the algorithm in the given environment. However, if the controller task always executes on the same host machine as the user task that it is controlling, then message traffic across the network is not increased. Rather, only traffic within the host is increased. The presence of the soul task introduces the possibility of a second level of pre-rollback messages. However, this was shown to be an easily solvable problem which does not entail much overhead in terms of verification of messages. Using the simple incarnation scheme , second level pre-rollback messages can be easily detected and eliminated. Finally, the *Body and Soul* model is constructed in such a way as to hide the details of Mach resource allocation/deallocation. The consequence of this is that new checkpointing and rollback algorithms can be built on top of the model relatively easily.

# Chapter 5

# Implementation of MRLV

*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*

The MRLV algorithm has been implemented in the context of a distributed debugger called **CDB**(Concordia Distributed deBugger)[LRAK90] which is part of an ongoing project at Concordia University. CDB is a tool which can be used to debug distributed programs written in the *C* or *C++* programming languages under the Mach operating system. Related work on this project can be found in [CYEP92, VKRA92, HSEG93, BDAN89, VENK88, PASS88, IIAM88]

MRLV is the algorithm which is used in CDB to support checkpoint and rollback recovery of distributed programs during debugging. It is implemented as a separate set of modules using the body and soul architectural model discussed in chapter 4. This chapter introduces the CDB tool and discusses its checkpoint and rollback facility along with some implementation details.

## 5.1 CDB Distributed Debugger

The current implementation of CDB , called **XCDB**, runs under the *X-Window system* and the *Athena Widget set* [NYOR90]. XCDB allows the debugging of programs

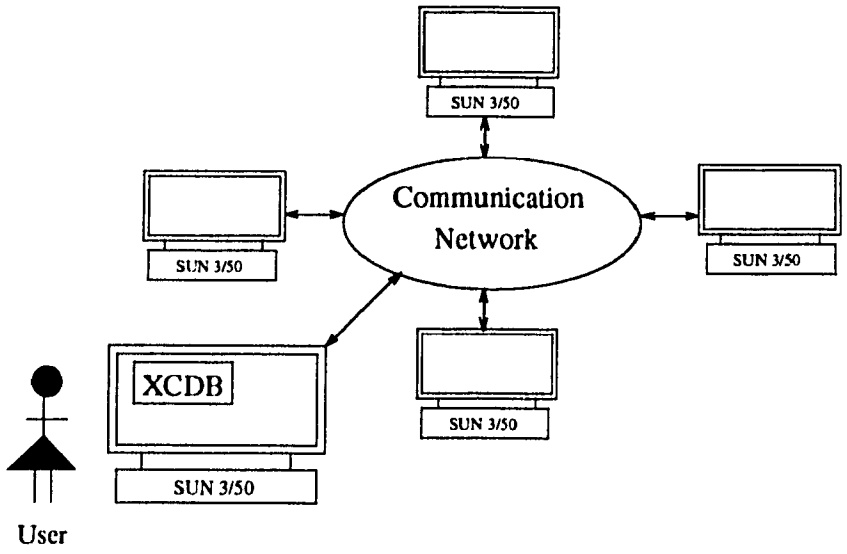


Figure 5.1: User debugs distributed program from a central site

written specifically for a network of SUN 3/50 workstations running the Mach operating system.

The main XCDB program, which can be executed on any of the five machines, acts as a central debugging site for monitoring programs running on remote processors. The user monitors program execution from the central site where he/she is able to interactively debug the program with the help of XCDB's debugging facilities. (see Figure 5.1).

### 5.1.1 Compiling an XCDB program

In order for a program to be able to run under XCDB, the user must modify his/her code before compilation so that information necessary for checkpointing and rollback may be made available to the debugger. These changes are:

- The user is required to include a constant called `DEBUG_MSG` in all of a program's message structures.
- The `cdb.h` header file must be included in all programs.
- The first instruction to be executed in the code must be the `INIT_CDB` macro.

The user must also include the **ddebug** library when compiling the program. These changes allow the user's program to interact with XCDB's checkpoint and rollback recovery subsystem transparently. More detail about preparing programs for execution under XCDB can be found in [CYEP92].

### 5.1.2 XCDB Facilities

At this time, XCDB provides two basic facilities:

- Breakpoint Detection and Halting
- Checkpoint Creation and Rollback

The breakpointing facility allows users to monitor the behaviour of a distributed program. A user specifies distributed breakpoints to be detected by XCDB on their program using a predicate definition language called **PDL** [CYEP92]. PDL predicates are defined as trees whose leaf nodes are *primitive events* and whose non-leaf nodes are *operators*. A primitive event consists of any event which can be detected by a local debugger attached to each task eg. assignment to variables, task termination, etc. This local debugger is currently GNU's **GDB** debugger [STAL89][1].

There are currently two PDL predicate operators. They are Lamport's "happens before"($\ll$) and "concurrent"(&&) relations as defined in section 1.5.1. Using these operators and primitive events, the user can build predicate definitions which span several tasks in the distributed program. In Figure 5.2 a PDL predicate definition is depicted with its corresponding execution. The primitive events $e_1$, $e_2$, and $e_3$ represented by the leaf nodes occur in several tasks which are not executing on the same host machine.

Upon execution of the program, if events have occurred such that the pre-defined PDL predicate is satisfied , the system is halted in a consistent state. At that point, state information of the various tasks can be extracted. The extracted state infor-

[1]It should be noted that one of the main reasons that GDB was chosen was because of the fact that it can support multiple threads as described in [CABL89].

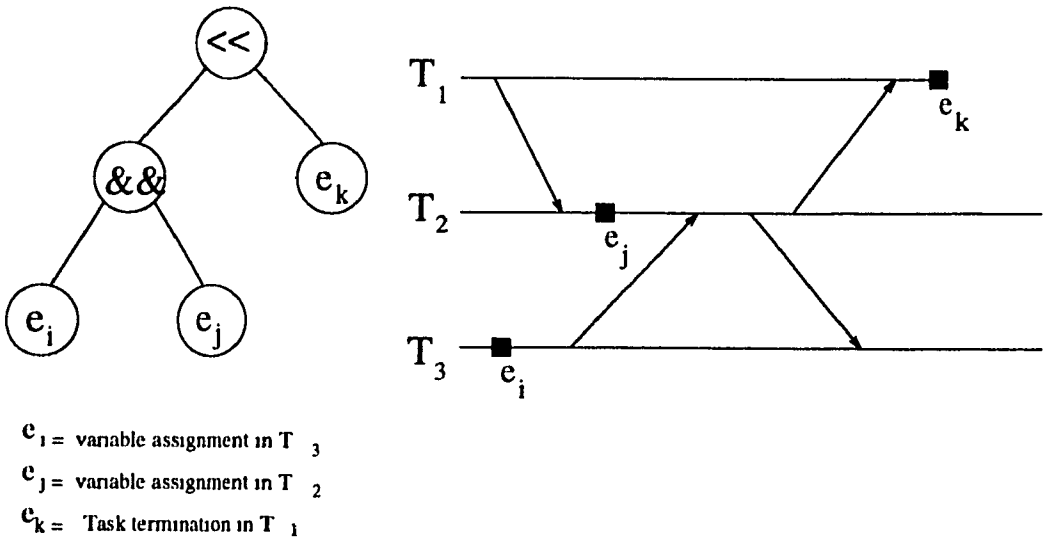


Figure 5.2: PDL predicate definition and corresponding execution.

mation can be any information which could be returned by the GDB debugger eg. values of variables,registers, program counter etc.

The XCDB checkpointing facility allows a user to rollback the execution of any task to a pre-defined point in its execution without being concerned about the need to rollback other tasks due to causal dependencies. The user is responsible for defining where in the source code checkpoints should be created. Upon completion of the distributed program, or detection of a predicate, the user may selectively rollback the execution to any of the defined checkpoints using the XCDB interface and restart the execution from that point.

## 5.2 XCDB Checkpoint and Rollback Facility

XCDB's checkpoint and rollback facility is based on user defined checkpoints and interactive rollback through XCDB's user interface.

### 5.2.1 Checkpoint Creation Flags

The user indicates places in the code to which he/she may wish to rollback at a later time using *checkpoint creation flags*. Checkpoint creation flags can be placed anywhere in the application program code. The user need not worry about the creation of *recovery lines* since these will be created by XCDB according to causal dependencies (i.e. sending and receiving of messages) according to the MRLV checkpointing algorithm. Therefore, the user is only concerned with the rollback of one particular task at any given time. However, the rollback of that task may necessitate further rollbacks of other tasks. The C code fragment in Figure 5.3 shows how a programmer would use checkpoint creation flags in the distributed program code. In this fragment , `checkpoint()` is the checkpoint creation flag, `task_is()` is a function which returns the name of the current task, `setup_send()` sends a dummy message using a Mach `msg_send()` system call to its parameter port, and `receive_data()` receives a message on the calling task's port using a Mach `msg_receive()` system call. There are two tasks $A$ and $B$ which simply send messages to each other[2]. The user places the `checkpoint()` flags in the code at points in the code which he/she believes to be "before" the bug. The flags in this code fragment results in the transparent creation of the recovery lines depicted in Figure 5.4. In effect, the checkpoints created by the placing of checkpoint creation flags are MRLV SIC checkpoints and the unspecified checkpoints created by XCDB transparently are MRLV RC checkpoints. Note that the message received by task $B$ right after the creation of its SIC will be saved in the checkpoint so that the recovery line $< B, 1 >$ is consistent.

### 5.2.2 Interactive rollback using XCDB user interface

After the user has defined all checkpoints by placing checkpoint creation flags in his/her code, the distributed program is compiled according to the procedure detailed in section 5.1.1.

When XCDB is started on one of the host machines, the main XCDB window in

[2]the actual contents of the message is unimportant.

```
if (task_is("A")){
   printf("this is task A\n");
   sprintf(other_port_name,"B");

   /* lookup of send rights for task B's port here */
   netname_look_up(name_server_port,"*",other_port_name,&other_port);

   checkpoint();                /* creates SIC <A,1>
   setup_send(other_port);
   receive_data();
   setup_send(other_port);
   receive_data();
   setup_send(other_port);

}
if (task_is("B")){
   printf("this is task B\n")
   sprintf(other_port_name,"A");

   /* lookup of send rights for task a's port */
   netname_look_up(name_server_port,"*",other_port_name,&other_port);

   receive_data();
   setup_send(other_port);
   checkpoint();                /* creates SIC <B,1> */
   receive_data();              /* This message will be saved */
   setup_send(other_port);
   receive_data();
}
```

Figure 5.3: Distributed program fragment depicting checkpoint creation flags

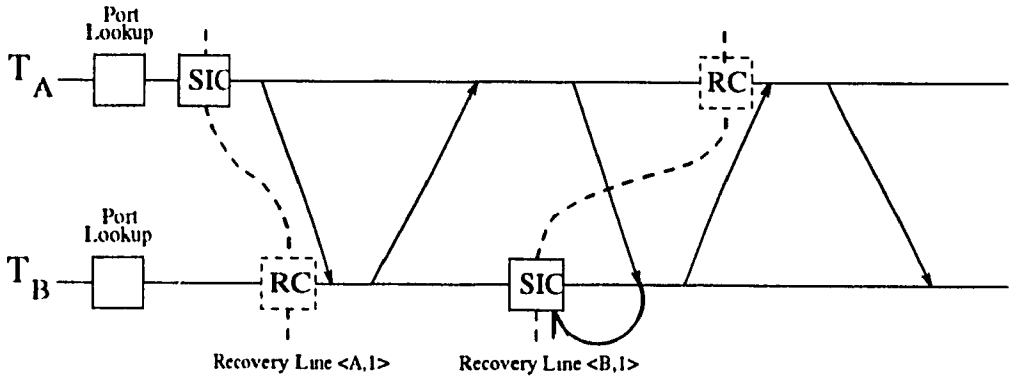


Figure 5.4: Transparent creation of recovery lines by XCDB.

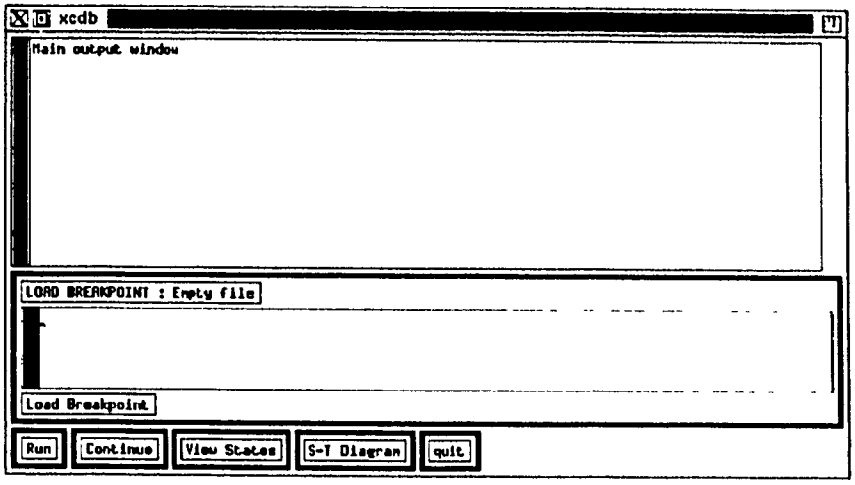


Figure 5.5: Main window of the XCDB debugger.

Figure 5.5 is displayed. The user can load a PDL breakpoint definition by clicking on the **Load Breakpoint** button with the mouse pointer on the main XCDB window. The PDL definition describes the names of the tasks, the machines on which they are supposed to execute, and the predicate which is to be detected. For the purposes of this discussion however, details about breakpoint detection are left out and it is assumed that no breakpoint predicate is specified by the user.

After the initial setup, the program can be executed by clicking on the **Run** button. When the program has halted , either due to program termination or detection of a predicate satisfaction, an ST-diagram depicting the execution can be displayed by clicking on the **St-Diagram** button. Figure 5.6 shows the St-diagram window after the execution of a program under XCDB. The squares along the task execution lines represent events which have occurred during the execution such as port allocation or task termination. Of particular interest to this discussion are the checkpoint events labeled as **CP**. These events represent the user-defined checkpoints which were specified in the program code prior to compilation. Since the user is not concerned with the actual recovery lines which are transparently built by the underlying mechanism, the checkpoints are represented as single entities on the user interface's St-diagram.

Clicking the **Rollback** button in the St-diagram window allows the user to select a

checkpoint (**CP** event) to rollback to. When this button is clicked, the checkpoints are highlighted in red to indicate that they are active. The user can then click on any of the highlighted **CP** events and XCDB will automatically rollback all necessary tasks to bring the task which owned the CP event back to the state it was in when the CP event occurred. All tasks which were rolled back are then restarted. Figure 5.7 shows the St-diagram window after the user has selected the first checkpoint in task 465510332 to rollback to. It can be seen by comparing Figure 5.7 with Figure 5.6 that the re-execution is identical to the initial execution starting from the selected **CP** event. Rollback to the selected checkpoint required a rollback of all tasks since they had all communicated with each other either directly or indirectly since that checkpoint had been created. The user can view both executions by using the scrollbar at the top of the St-diagram window and program executions are delimited by an *incarnation delimiter*. In Figure 5.7 the incarnation delimiter is labeled **Incarnation 1**. This indicates that all events after the incarnation delimiter are from the first re-execution of the program after the first rollback has taken place.

Rollbacks can be initiated by clicking on **CP** events from any of the executions depicted in the St-diagram window. For example, clicking on the **CP** event in task 910392826 from the St-diagram window in Figure 5.7 results in the re-execution depicted in Figure 5.8. Note that the delimiter now reads **Incarnation 2**. Also, note that only 2 tasks were rolled back this time because only these two tasks had communicated since the creation of the checkpoint which was selected for rollback. Also noteworthy is the **RCV** event which immediately follows the **CP** event. There was no need to rollback the sending task since this message had been buffered by the system according to the MRLV algorithm in order to keep the recovery line consistent. Hence, only the **RCV** event is depicted on the St-diagram.

The user driven rollback mechanism is useful since it allows the user to control execution of remote tasks from a central site through the St-diagram abstraction.

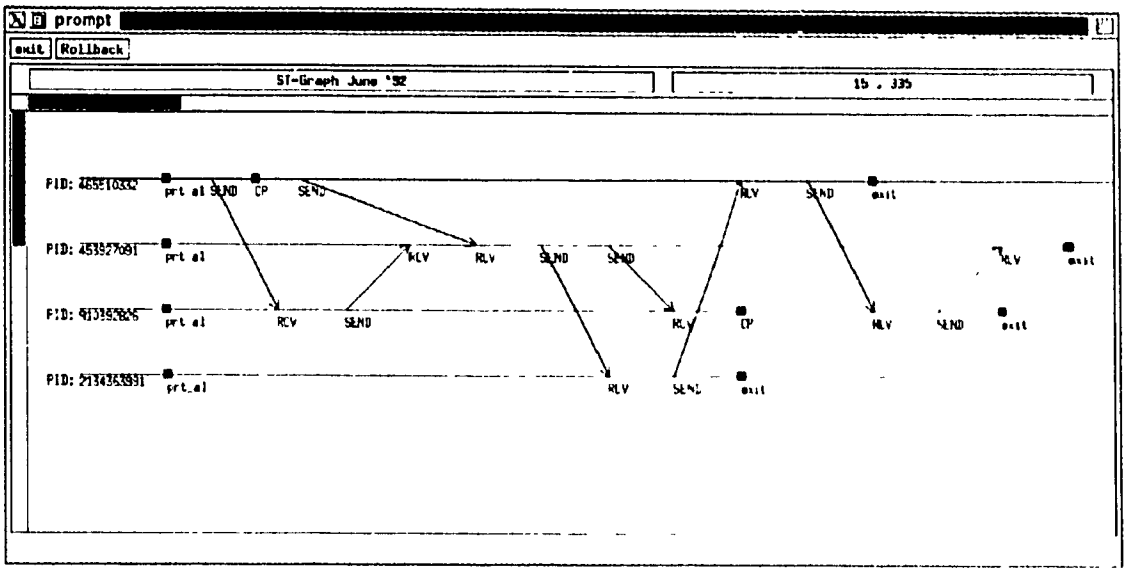


Figure 5.6: ST-diagram after first execution of distributed program under XCDB.

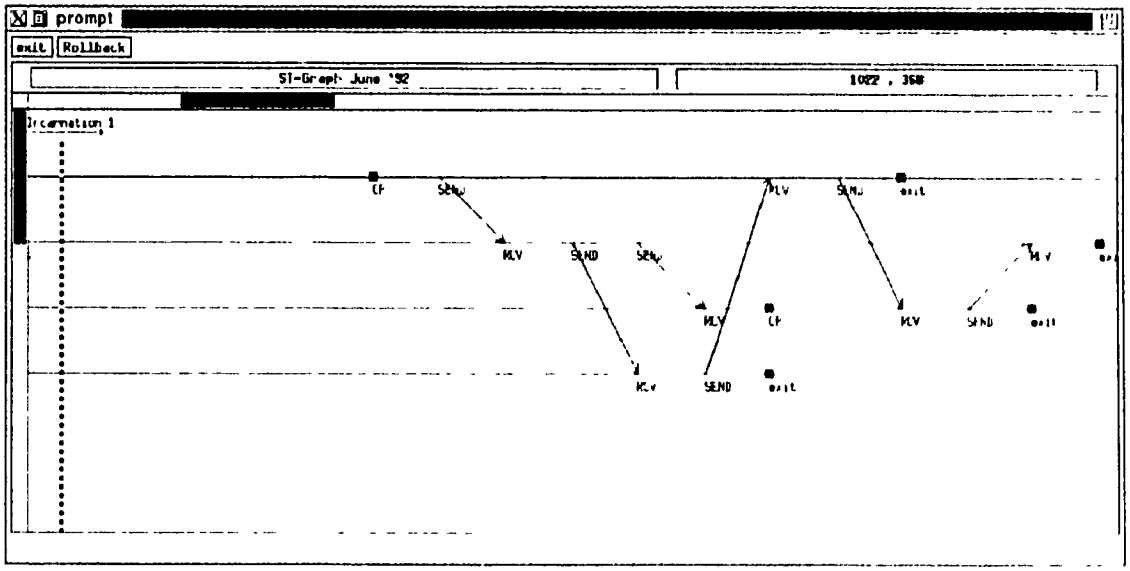


Figure 5.7: ST-diagram after first rollback of execution under XCDB.

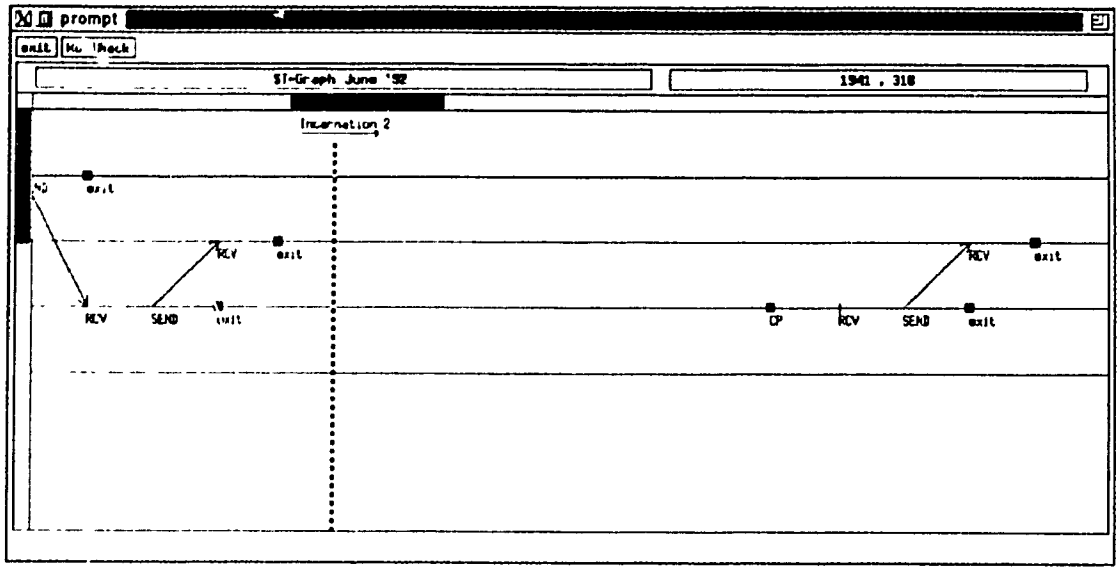


Figure 5.8: ST-diagram after first rollback of execution under XCDB.

## 5.3 Architectural aspects of XCDB with respect to MRLV

The checkpoint and rollback recovery facility module of XCDB uses the MRLV algorithm and the body and soul model. It has been fully implemented to support the checkpoint and rollback facilities previously discussed.

### 5.3.1 Implementation Assumptions

The implementation of the MRLV algorithm supports multiple threads and ports in the XCDB tool. However, previous work on XCDB [CYEP92, VKRA92] has restricted the implementation model to a single-thread per task model. Therefore, although MRLV can support multiple threads and ports, the XCDB tool does not allow the user to exploit them.

The program being debugged must run to completion before a rollback can be initiated by the user from within XCDB. This restriction has been made for the simplification of the integration of rollback recovery modules with the previous XCDB implementation. However, it should be noted that the checkpoint and rollback recov-

ery modules by themselves do support rollback before completion of the program.

The system has made the assumption that once a Mach port is allocated, it is not deallocated by the programmer. The deallocation may occur due to the death of the owner task. However, the programmer is not allowed to explicitly call a port deallocation system call or to check out a port from the central `netmsgserver`. This can be changed easily by replacing `port_deallocate()` and `netname_check_out()` system calls to macros which simply don't do the operations but return as if they did. However , we have not done it in the current implementation.

In order to detect the termination of a task which is part of the distributed program, the programmer is required to explicitly exit a task using an `exit()` system call. Otherwise, XCDB will still work, but will not be able to display to the user that a particular task has terminated.

The checkpoint and rollback recovery subsystem of XCDB does not support the random creation of ports. Tasks must create their ports and establish communication links with other remote processes before starting their computation. Furthermore, tasks are permitted to communicate with each other through only one user-defined port. There is no real need for multiple ports if a restriction has been placed on the number of threads allowed per task since only one port can be read at any time anyways.

For simplicity, it is assumed that only one task involved in the computation executes on each of the machines. This is to simplify the global naming convention used by XCDB.

### 5.3.2 XCDB system architecture

The complete XCDB system architecture including the modules needed for checkpointing and rollback recovery is depicted in Figure 5.9. Both breakpointing and rollback control lies within the XCDB debugger. The XCDB program itself is a multi-threaded Mach task. Within the breakpoint control, threads are spawned to monitor events occurring in remote tasks. The threads communicate with a local debugger (GDB) attached to each user task and a vector clock module is used to

determine the order of events for predicate detection. Along with the local debugger, a controller task (as in the body and soul model) is attached to each task. Also, the checkpoint and rollback daemons are spawned within the user task and they communicate with the controller. The rollback control of XCDB and all remote controllers get send rights to all other controllers through the CNS which is as described in the body and soul model. The source code of a user task is augmented for both the breakpointing and checkpointing facilities. Specifically, the code is augmented to allow transparent communication between the user task and the vector clock, local debugger, and controller modules.

When messages are sent or received by a user task, they go through two levels of processing by the augmented code. They are first processed by the breakpointing modules which append necessary information needed for breakpointing such as vector clock values, and then the checkpoint and rollback recovery augmented code appends further needed information such as incarnation numbers.

## 5.4 Implementation Issues

This section describes and discusses selected important aspects of the implementation of MRLV within XCDB. It discusses:

- How task states are saved and restored?
- How the user task's source code is augmented?
- How checkpoint creation flags cause the creation of checkpoint?
- How rollback is initiated through the user interface?

### 5.4.1 Saving and restoring of task states

Mach's port abstraction provides a simple way of manipulating a task's resources. All of a task's resources can be accessed through its *environment port.* Suppose a task $T_i$ owns send rights to a task $T_j$'s environment port . $A$ can access all port rights (i.e. extract or insert) owned by $T_j$. Furthermore, all of $T_j$'s virtual memory

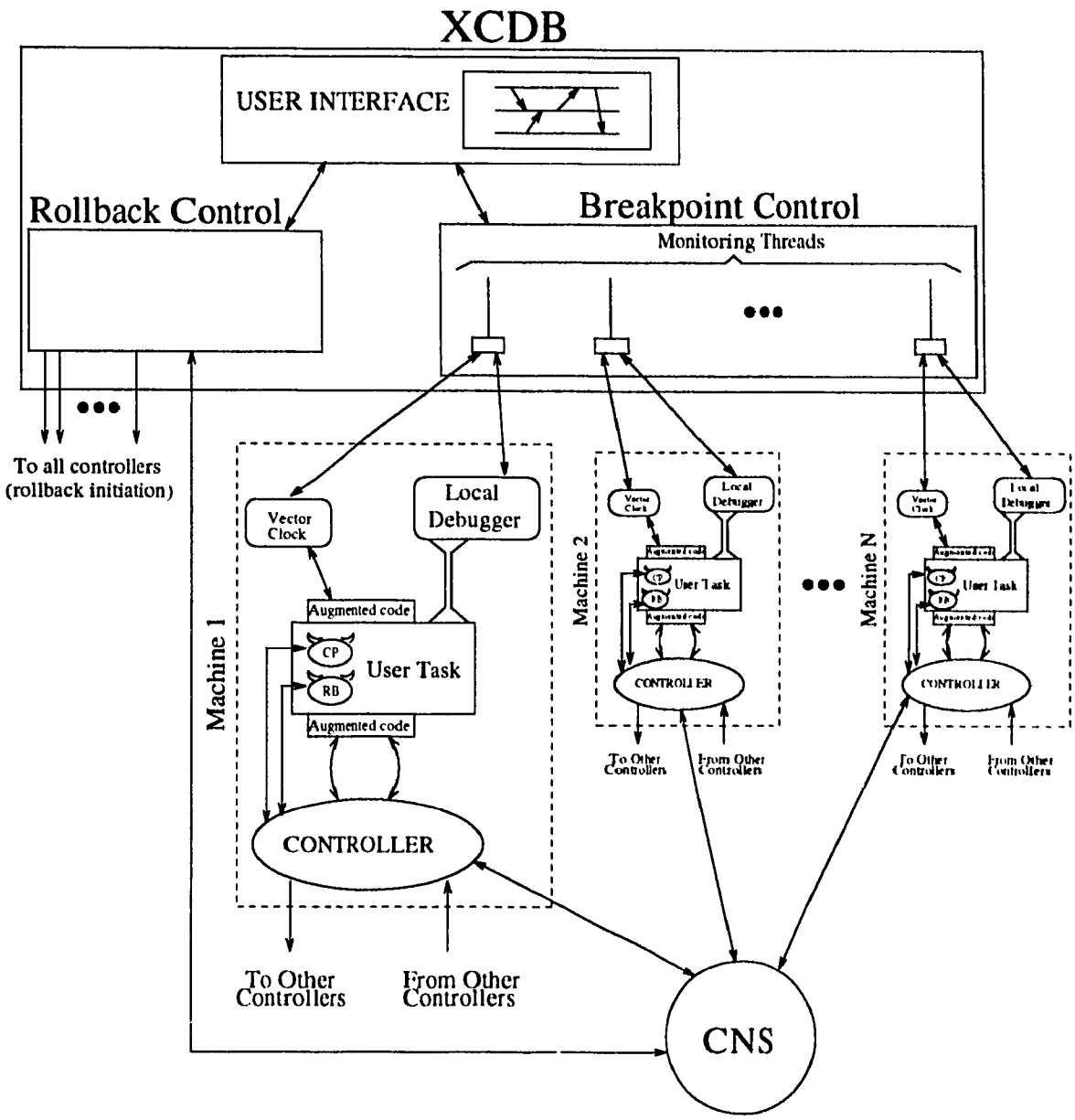


Figure 5.9: XCDB system architecture.

space and registers can be read or written by $T_i$. It is through this environment port that a controller task steals port rights from the user task which it is controlling and this is done by a straightforward use of Mach system calls which allow port right insertion and extraction. The controller simply follows the method for stealing ports as described in sections 4.6.2 and 4.6.2.

Although the environment port abstraction allows easy access to a task's virtual memory and registers, saving and restoring a task's virtual memory is not simple. The algorithms which have been used for saving and restoring task states are variations of those presented in [GOGO90].

### Saving the state of a task

The ORM system described in [GOGO90] also uses the idea of a checkpointing daemon which is responsible for saving the state of a task. When their system wishes a checkpoint of a task to be created, the checkpointing daemon within that task performs a Unix `fork()` system call to create a child task (see Figure 5.10). This child task's virtual memory space is an exact copy of the virtual memory of the parent task which created it and therefore can be used to save the parent task's state. Upon creation, the child immediately suspends itself and while it is suspended, the checkpointing daemon within the parent task copies its virtual memory space and register values to disk in a checkpoint using Mach system calls. The parent task then resumes the child which immediately exits.

In the MRLV implementation for XCDB, a similar approach is taken to that in ORM. The checkpoint daemon forks a child task when instructed to create a checkpoint by its controller. However, instead of having the parent task (i.e. the user task) save a copy of the child's virtual memory, it is the child task which copies the parent task's state to disk. After forking the child, the parent task (user task) checks its environment port into the `netmsgserver` so that the child task can look it up, and immediately suspends itself. While the parent is suspended, the child first looks up its parent's environment port and then copies the parent's virtual memory and register states to disk in a checkpoint using Mach system calls (eg. `vm_region()`,

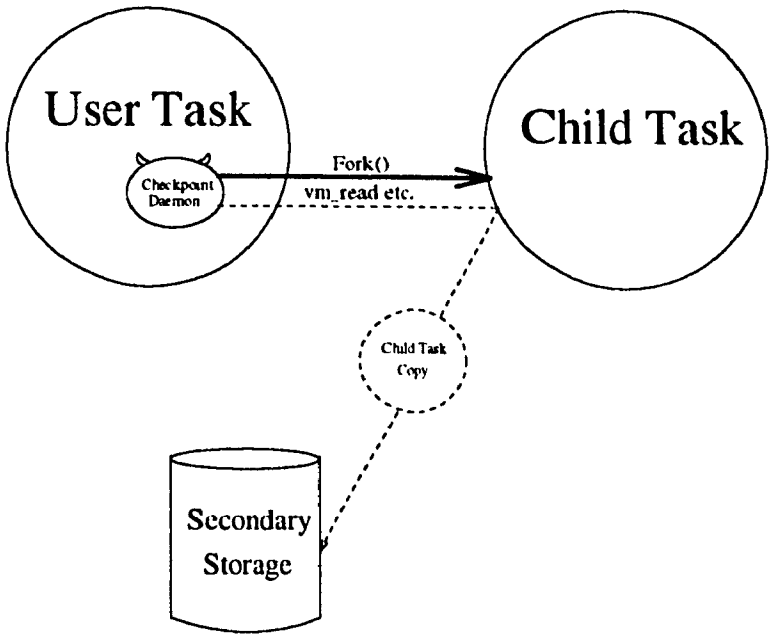


Figure 5.10: Saving the state of a task using `fork()` in ORM

`vm_read()`, etc.). This is depicted in Figure 5.11

### Restoring previously saved states

The ORM algorithm used for restoring states is similar to the one it uses for saving states. When the ORM system wishes to rollback a task, that task's rollback daemon[3] performs a unix `fork()` system call to create a child task. The child is used as a *template* into which the state which was stored in the checkpoint is to be copied. Before copying the state on disk into the child template, the rollback daemon in the parent deallocates all of the child's virtual memory. Once the parent has finished copying the state which was previously saved in the checkpoint from the disk into the child task, it terminates its task (i.e. the parent) so that the child task can take over the execution. The execution restarts from the point saved in the checkpoint. Note that since the child task would have had a copy of all register states and virtual memory, this state would include the state of all entities within that task (i.e. the

[3]The daemon in ORM is not actually called a rollback daemon but this term has been used here to make the comparison between XCDB and ORM more clear.

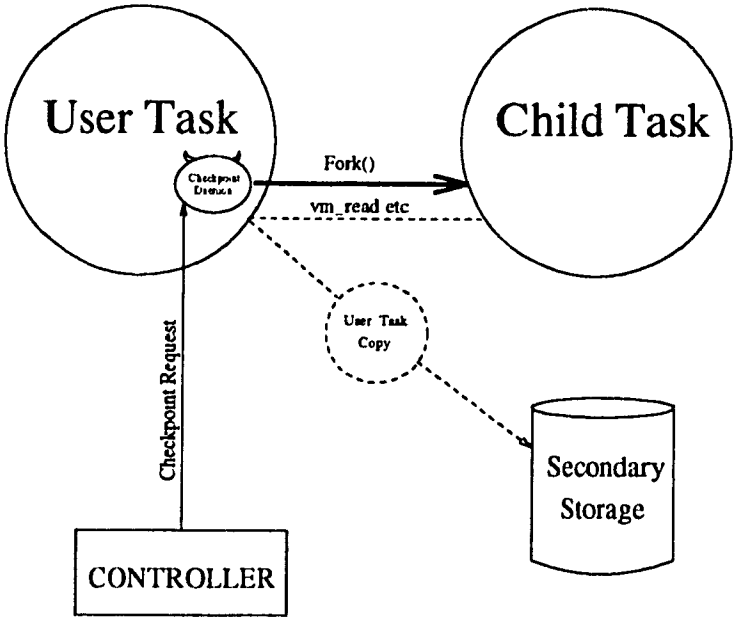


Figure 5.11: Saving the state of a task using `fork()` in XCDB

rollback daemon and the checkpoint daemon).

The XCDB restoration of states again uses the opposite approach to that of ORM. When the rollback daemon of a task is instructed by its controller to rollback the state of its task, it again performs a unix `fork()` call. However, unlike in ORM it is the parent task which acts as the template for restoring the state saved in the checkpoint. After forking a child, the rollback daemon suspends its user task. The child acquires rights to the parent's environment port and through this port it deallocates all of the parent's virtual memory, and clears all of its registers. The child then copies the state saved on disk in the checkpoint using Mach system calls (eg. `vm_write()` etc.). Once it has finished doing so, the child resumes the parent task and terminates itself. The parent then continues its execution from the point saved in the checkpoint without being aware of the rollback which has taken place.

#### Advantages of XCDB approach

The algorithms presented here for the implementation of MRLV with respect to saving and restoring states of tasks within XCDB are more efficient than those presented in [GOGO90] for the following reasons:

- Restart of daemons is simple after a rollback since they are in the same state that they were when the checkpoint was created.
- Communication need not be re-established between user task and its controller.
- No need to checkpoint port rights.
- No need to re-create ports after a rollback. Since the parent task is used as the template for state restoration upon rollback rather than the child task, the ports always remain intact. When the child is used as the template, its *port space* must be manipulated so that it owns the same ports as the parents did at the time of the checkpoint creation. The new algorithm makes this unnecessary and therefore less time is spent on rollback.

### 5.4.2 Augmented code

The user code is augmented to allow the transparent interaction with the XCDB system tasks. This is achieved by modifying the program code as described in section 5.1.1. The Mach kernel has not been modified, rather the system calls have been replaced by C macros which perform the XCDB functions along with the real system call. This is used to perform needed functions such as sending UCM's to the controller when system calls are called by the user task.

### 5.4.3 User-defined checkpoint mechanism

User-defined checkpoint flags are implemented as a function call which interacts with the controller of the user task. When a checkpoint flag is hit during execution of a program, the user task prepares itself for checkpoint creation by entering into a

busy wait. A message is sent to the task's controller informing it of the desire to create a SIC checkpoint (i.e. a SIC Creation UCM message; i see section 4.6.2). The controller in turn sends a message to the checkpoint daemon which actually creates the checkpoint. Once the checkpoint is created , the user task exits its busy wait and continues its execution as if nothing had happened.

### 5.4.4 Initiating rollback through the user interface

The rollback control within XCDB holds send rights to all controllers and is aware of their unique id's. When a rollback to a particular checkpoint is initiated by the user through the interface, the rollback control sends a message to the controller which is the owning task of the recovery line to which the checkpoint belongs. The controller in turn initiates a rollback according to the MRLV algorithm using the algorithm of section 4.6.3.

# Chapter 6

# Summary and Future Work

*"Is that all?" asked Alice.*
*"That is all." said Humpty Dumpty. "Goodbye."*
*– Lewis Carrol, "Through the Looking Glass"*

## 6.1 Summary

Checkpoint and rollback recovery algorithms which rely on FIFO communication between tasks are not well suited for operating environments such as Mach which allow multiple threads, and support a port-based communication model. This unsuitability arises because FIFO ordering of messages between tasks cannot be guaranteed due to the presence of multiple ports and threads within tasks. The RLV checkpoint and rollback recovery algorithm relies on FIFO ordering of messages in its rollback phase to detect pre-rollback messages. A modified version of RLV , called MRLV has been designed as a part of this thesis. In MRLV, the advantages of RLV are preserved but there is no dependency on FIFO channels for pre-rollback message detection thereby allowing the establishment of multiple channels between any pair of communicating tasks. The modification is in the rollback phase of the RLV algorithm and is based on the concept of task *incarnations*. Every time a task rolls back its execution, it is said to be in a new incarnation. By making incarnation information a global knowledge, the MRLV algorithm succeeds in determining which messages are from which incarnation of which task, and can therefore detect pre-rollback messages. The MRLV algorithm was shown to have no extra message overhead when compared to

the RLV algorithm. Furthermore, the detection of pre-rollback messages is simple and efficient.

In implementing the MRLV algorithm in a Mach environment, two basic problems arise. MRLV assumes that all tasks have knowledge of each other so that broadcasting of rollback messages can be possible. This is not so in a port-based communication model such as Mach. Also, when a task terminates , its perishable resources are deallocated by the underlying operating system and therefore, upon rollback these resources would no longer exist. In order to implement the MRLV algorithm , an architectural model called "Body and Soul" was designed which solves these problems by the use of re-direction of resources. The design is in keeping with the notion of incarnations and "Body and Soul" is a metaphor for the idea of controlling the execution of a task. The model uses the idea of a controller, or "Soul" task which is responsible for maintaining perishable resources across successive incarnations of the user task (Body) which it controls. The soul task carries out operations required by the MRLV algorithm in collaboration with daemons which are dormant within the user task. The maintenance of resources by the soul task reduces the overhead involved in rollback since resources such as ports do not need to be re-allocated upon rollback. Also, the model allows the operations of MRLV to be performed transparently without affecting the user task's execution in any way. However, the model introduces a second level of pre-rollback messages, and the possibility of deadlock between soul tasks. These problems were shown to be easily solvable by extending the use of the incarnations concept.

The MRLV algorithm has been implemented using the body and soul model in the context of a general purpose distributed debugger called XCDB which is an ongoing project at Concordia University. MRLV provides XCDB with its checkpoint and rollback facility which can be used to "narrow" the area of code being debugged, or to interactively detect unwanted non-determinism. XCDB provides the user with a simple view of checkpointing and rollback through its user interface. The user can place checkpoint creation flags anywhere in his/her application code and using an abstracted view of the computation, interactively select positions in the computation

for rollback.

The implementation architecture is composed of reusable entities which can be used to implement any other checkpoint and rollback recovery algorithm in an environment which supports a port-based communication model (refer to section 4.8.1). Furthermore, the entities can collectively be looked upon as a complete rollback recovery module which can be used within different encapsulating applications.

## 6.2 Future Work

The following is a list of possible extensions to the current work, and future work which can be done to improve the present system.

- The current implementation only supports one task per machine. This is due to the naming convention used by the checkpoint and rollback recovery algorithms for system ports. i.e. ports in the controller and checkpoint and rollback daemons. The reason for this is that currently, the names are distinguished by appending the name of the machine on which the task is running onto the names of the ports. This could be changed by appending the name of the machine, and a unique identifier. This change would allow multiple tasks to run on each of the machines available.

- The current implementation only supports a single user-defined port. Once multiple threads are supported by XCDB, this could be easily changed by modifying the controller to keep track of more than one user-defined port. Allowing multiple ports would allow the user to create more flexible applications.

- Users are not allowed to specifically deallocate ports. By changing the Mach `netname_check_out()` and `port_deallocate()` system calls to macros which simply return without performing the operations, the user would be allowed to make these calls. By allowing the user to make these calls, the underlying system would become more transparent.

- Currently, the breakpoint modules have not been set up so that breakpoints can be re-specified after a rollback. This is needed to allow an interactive user to narrow the area of code being debugged.

- An alternate implementation could replace XCDB's checkpoint creation flags with automatic checkpoint creation by the system based on the breakpoint events which have been specified. i.e. XCDB would create checkpoints whenever an "interesting" event occurred. Automatic checkpoint creation would allow the system to use heuristics in aiding the user to pinpoint an error. However, this would require more efficient methods of storing checkpoints due to the potentially large number of checkpoints which may be created.

- Currently, useless checkpoints are always destroyed. However, the destruction of checkpoints is not always reflected in the user interface. As of now, if a checkpoint which no longer exists is selected for rollback, the system does not perform any rollback. This could be changed such that non-existing checkpoints are completely removed from the user interface representation.

- XCDB is currently a "Live Detection" debugger. Therefore, it can only effectively debug deterministic programs. A future implementation could include a "Record and Replay" module which would ensure that non-deterministic programs are always re-executed the same way. The module could run under the checkpoint and rollback recovery module. A deterministic replay module has been built in [VKRA92] for such purposes, however it is not complete.

# Bibliography

[BACH86] Maurice J. Bach, "The Design of the UNIX Operating System.", *published by Prentice-Hall, Inc.*, 1986.

[BDAN89] B.M. Dang, "Methodology and Tools for Distributed Debugging", M.Sc Thesis, Concordia University, Montreal, December 1989.

[BRCI84] D. Briatico, A. Ciuffoletti, L. Simoncini., "A Distributed Domino effect free recovery algorithm", *Proc. Symposium on Reliability in Distributed Software and Database systems*, Oct. 1984, pp. 207-215.

[CABL89] Deborah Caswell and David Black, "Implementing a Mach Debugger For Multi-threaded Applications", *USENIX*, Winter, 1990.

[CHAN85] K.M. Chandy, L.Lamport., "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM TOCS*, vol.3, no.1, pp.63-75, 1985

[CHKE91] Fuyuan Chao, Kenevan J.R., " A non-FIFO checkpointing protocol for distributed systems ", *1991 Symposium on Applied Computing.*, Kansas City, MO, USA, 1991, pp. 266-72.

[CHKR88] Moon Jung Chung, M.S. Krishnamoorthy, " Algorithms of Placing Recovery Points", *Information Processing letters*, vol. 28, 1988, pp. 177-81.

[CHLI80] Chuen-Pu Chou, Ming T. Liu. "A Concurrency Control Mechanism and Crash Recovery for a Distributed Database System (DLDBS)", *Distributed Databases*, S. Delobel and W.Litwin (eds.), North-Holland Publishing Company, 1980, pp.201-214.

[CHWO89] C.K. Chang, L.F. Wong. "Design of Monitoring System for distributed applications", *Information and Software Technology*, Vol. 31, No. 6, July-August 1989, pp.295-304.

[CRJA91] Christian F., Jahanian F., "A timestamp-based checkpointing protocol for long-lived distributed computations.", *Proceedings of the Tenth Symposium on Reliable Distributed Systems.*, Pisa, Italy, 1991, pp. 12-20.

[CYEP92] C.Yep, "A Debugging Support Based on Breakpoints for Distributed Programs Running Under Mach ", M.Sc Thesis, Concordia University, Montreal, November 1992.

[FRAZ89] Tiffany M. Frazier, Yuval Tamir, "Application Transparent Error-Recovery Techniques for Multicomputers", *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, Vol 1, March 1989, pp. 103-8.

[FUKA91] Fukumoto S., Kaio N., Osaki S., " A Note of Checkpoint Policies Taking Account of Unsuccessful Rollback Recovery", *Methods of Operating Research*, vol. 64, 1991. pp. 149-58.

[GOGO90] Goldberg A., Gopal A., Kong Li, Strom R., Bacon D. F., "Transparent Recovery of Mach Applications", *USENIX Workshop Proceedings on MACH*, October 1990, pp.169-83

[GREG85] S.T. Gregory, J.C. Knight, "A New Linguistic Approach to Backward Error Recovery", *The 15th FTCS*, 1985, pp.404-409.

[HAWE88] Dieter Haban, Wolfgang Weigel. "Global Events and Global Breakpoints in Distributed Systems.", *21st Hawaii International Conference of System Sciences 1988*, IEEE, pp.166-170.

[HORN71] J.J. Horning *et al.*, "A Program Structure for Error Detection and Recovery", pp. 171-187 in *Lecture Notes in Computer Science 16*, ed. E. Gelenbe and C. Kaiser, Springer-Verlag, Berlin, 1974.

[HSEG93] H. Segel, "Monitoring Distributed Systems", M.Sc Thesis, Concordia University, Montreal, March 1993.

[IHAM88] I. Hamamtzoglou, "A Model and Methodology for Distributed Debugging", M.Sc Thesis, Concordia University, Montreal, July 1988.

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

[JOZW91] Johnson D.B., Zwaenepoel W., " Transparent Optimistic Rollback Recovery", *Operating Systems Review*, vol. 25, no. 2, April 1991, pp. 99-102.

[KELL91] John P.J. Kelly, Thomas I. McVittie, Wayne I. Yamamoto. "Implementing Design Diversity to Achieve Fault Tolerance", *IEEE Software* , IEEE, July 1991, pp. 61-71, Vol. SE-10, No. 2, 1984, pp.210-219.

[KIYO86] K. Kim, J. You, A. Abnouelnaga, "A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Systems.", *The 16th FTCS*, 1986, pp.130-135.

[KOTO87] Richard Koo, Sam Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, January 1987, pp.23-31.

[KRKA84] C.M. Krishna, Kang G. Shin, Yann-Hang Lee, " Optimization Criteria for Checkpoint Placement", *Communications of the ACM*, vol. 27, no. 10, October 1984, pp. 1008-1012.

[KUMA90] Akhil Kumar. "A Crash Recovery Algorithm Based on Multiple Logs that Exploits Parallelism", Graduate School of Management, Cornell University, Tech Report, May 1990.

[LAMP78] L.Lamport, "Time, Clocks, and Ordering of Events in Distributed System", *Communications of the ACM*, Vol 21(7), July 1978, pp. 558-565.

[LECR87] Thomas J. Leblanc and John M. Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, Vol. 36, No. 4, pp. 471-482, April 1987.

[LISH89] Tein-Hsiang Lin, Kang G. Shin, "Optimal Rollback Recovery with Checkpointing and Damage Assessment for Concurrent Processes", Computer Science and Engineering Technical report no. 17, The University of Michigan, 1989.

[LRAK90] H.F. Li, T. Radhakrishnan, V. Krawczuk, "A Toolkit for Debugging Distributed Programs", *Concordia University*, December, 1990.

[MACHE89] Linda R. Walmer, Mary R. Thompson, "A Programmer's Guide to the Mach User Environment", *Department of Computer Science, Carnegie Mellon University*, November 1989.

[MACHK90] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Avadis Tevanian Jr., Michael Wayne Young, "MACH Kernel Interface Manual", *Department of Computer Science, Carnegie Mellon University*, August 1990.

[MACHS89] Linda R. Walmer, Mary R. Thompson, "A Programmer's Guide to the Mach System Calls", *Department of Computer Science, Carnegie Mellon University*, December 1989.

[MCDO89] Charles E. McDowell, David P. Helmbold. "Debugging Concurrent Programs", *ACM Computing Surveys*, Vol. 21, No. 4, December 1989, pp. 593-621.

[MOGE84] Hector Garcia-Molina, Frank Germano Jr., Walter H. Kohler, "Debugging a Distributed Computing System", *IEEE Transaction on Software Engineering*, Vol. 10, No. 2, pp. 210-219, March 1984.

[MORG85] Carroll Morgan, " Global and Logical Time in Distributed Algorithms.", *Information Processing Letters*, no.20, 1985, pp. 189-194.

[NYOR90] Adrian Nye, Tim O'Reilly, "X Toolkit Intrinsics Programming Manual 2nd ed.", *published by O'Reilly and Associates, Inc.*, Second Edition, 1990.

[PASS88] C.Passier, "Experimental Evaluation of Distributed Rollback and Recovery Algorithms", M.Sc Thesis, Concordia University, Montreal, May 1988.

[RAND75] Randell B., "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, SE-1(2), June 1975, pp.220-232.

[ROAB90] M. Rozier , V. Abrossimov, *et al.* "Overview of CHORUS Distributed Operating Systems.", CS/TR-90-25, April 1990.

[SAJA86] Robert D. Sansom, Daniel P. Julin, Richard F. Rashid, " Extending a Capability Based System into a Network Environment" *ACM Symposium on Operating System Principles*, 1986, pp. 265-275.

[SAMK90] Sam Kweon Oh, "A survey of the Basic Concepts and Techniques for Fault Tolerance in Distributed Real-Time Systems", Technical Report, Department of Computing and Information Science Queen's University, June 1990.

[SILB92] A. Silberschatz, J. Peterson, P. Galvin, "Operating System Concepts, 3rd ed.", *published by Addison-Wesley*, Third Edition, pp 597-629, 1992.

[SOAG85] Sang Hyuk Son, Ashok K. Agrawala. "A non-intrusive checkpointing scheme in distributed database systems. ", *FTCS-15*, June 1985, pp.99-101.

[SOAG89] Sang Hyuk Son, Ashok K. Agrawala. "Distributed Checkpointing for Globally Consistent States of Databases.", *IEEE Transactions on software Engineering*, vol. 15, no. 10, October 1989, pp.1157-67.

[SPEZ88] M. Spezzialetti, J.P. Kearns. "A General Approach to Recognizing Event Occurrences in Distributed Computations", *IEEE*, 1988, pp.300-307.

[SPEZ89] M. Spezzialetti, J.P. Kearns. "Simultaneous Regions: An approach to the Consistent Monitoring of Distributed Computations for Event Occurrences ",

*Proceedings of the 9th International Conference on Distributed Computing Systems*, 1989, pp. 61-69.

[STAL89] Richard M. Stallman, "GDB Manual", The GNU Source-Level Debugger, Third Edition, Version 3.4, Oct 1989.

[STRO85] Strom, R. E., and Yemini, S. A., "Optimistic Recovery in Distributed Systems", *ACM Transactions on Computer Systems 3*, Vol 3, August 1985, pp. 204-226.

[TAMI84] Y. Tamir, C. H. Sequin, "Error Recovery in Multi-Computers using Global Checkpoints", *13th International Conference on Parallel Processing*, Bellaire, Michigan, August 1984, pp.32-41.

[TEVRA87] Avadis Tevanian, Jr., Richard F. Rashid, "MACH: A Basis for Future UNIX Development", *Department of Computer Science, Carnegie Mellon University*, June 1987.

[VENK87] K. Venkatesh, T. Radhakrishnan , H.F. Li. "Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery", *Information Processing Letters*, Elsevier Science Publishers B.V., pp. 295-303, Vol. 25, No. 5, July 1987.

[VENK88] K. Venkatesh, "Global States of Distributed systems: Classification and Applications", Ph.D Thesis, Concordia University, Montreal, Jan. 1988.

[VKRA92] V.Krawczuk, "Distributed Debugging Based on Deterministic Reexecution-Methodology and Design of a Working Prototype", M.Sc Thesis, Concordia University, Montreal, September 1992.

[WOOD85] W.G. Wood, "Software Fault Tolerance", *Reliable Computer Systems*, Ed. S.K. Shrivistava, Springer-Verlag 1985, pp.249-291.

[WOWO90] Wojcik Z.M., Wojcik B.E., " Fault tolerant distributed computing using atomic send-receive checkpoints", *Proceedings of the second IEEE Symposium on Parallel and Distributed Processing.*, Dallas, TX, USA, 1990, pp. 215-22.

# Appendix A

# MRLV rollback algorithms: Detailed Discussion

This appendix describes the algorithms given in section 3.7.2 in more detail and gives justification for the steps in the algorithms. It is intended to clarify the algorithms given in that section.

## A.1 Rollback protocol assumptions

The rollback initiation algorithms for MRLV assume the following:

- Only one rollback occurs in the system at any one time and that rollback is uninterrupted.
- A task which initiates a rollback, rolls back its state to a SIC checkpoint, other tasks which participate in the rollback roll back their state to RC's.
- The SIC to which the initiating task is rolled back is selected by:
  1. A user in the context of debugging, or by
  2. Some selection algorithm in the context of fault tolerance.
- The algorithm operates in a reliable network.

```
Algorithm Initiate_Rollback(CP){
1    disable application message acceptance
2    increment current_inc_i
3    rollback state to CP
4    discard all messages saved in any checkpoint
         after the creation of CP
5    reset value of CCP_i vector to that saved in CP
6    purge all checkpoints created after CP was created
     /* broadcast rollback msg */
7    for all tasks T_j {
7.1      include current_inc_i in rollback_msg
7.2      include unique_task_id in rollback_msg
7.3      send rollback_msg to task T_j
     }
     /* update incarnation info */
8    for all tasks T_j{
8.1      wait for reply r from T_j
8.2      inc_tab_i[j] = r.current_inc
     }
9    update incarnation information in messages saved in CP
10   restore messages saved in CP
11   enable application message acceptance
}
```

Figure A.1: MRLV rollback initiation algorithm

## A.2 Rollback initiation algorithm

The algorithm for rollback initiation described in section 3.7.2 is shown in Figure A.1 with step numbers added.

The algorithm describes the steps taken by a task $i$ wishing to initiate a rollback by selecting its SIC checkpoint $CP$ for rollback. Step 1 of the algorithm ensures that by disabling the application message acceptance, no application messages will be lost while the rollback is taking place. This would be undesirable to the application program because of the lost message, and also because the message could be one that would necessitate the creation of a response checkpoint in the initiator. Therefore if the message was lost, an inconsistent recovery line could exist. Step 2 increments the initiating task's current incarnation number since the initiator is certain to rollback and this information must be reflected. In step 3, the actual rollback of the task state to the state saved in the SIC checkpoint $CP$ is performed. In step 4, all messages

which were saved in any checkpoint after the creation of $CP$, are purged. This is to ensure that messages are not duplicated in the checkpoints . The messages will be resent by the sending tasks which are participants in the rollback due to causal dependency. The CCP vector is reset to the value it had when the checkpoint was created in step 5 so that upon rollback, the algorithm can still make use of the vector to determine checkpoint creation. In step 6 all checkpoints created after the checkpoint $CP$ are discarded. Since the state is restored to the state at $CP$, and the execution will continue from that state onwards, any event which occurred after the creation of $CP$ will be repeated including the creation of all checkpoints which follow $CP$. Therefore to avoid duplicate checkpoints all checkpoints following $CP$ are discarded. Step 7 broadcasts a rollback message to all tasks in the system. Steps 7.1 and 7.2 include incarnation number and the unique identifier of the initiating task into the rollback message which is to be broadcast. Upon receipt of this message a task will follow the algorithm described in the next section for rollback participation.

In step 8, the initiating task, waits for replies from all other tasks. The reply includes the current incarnation of the sending task. This information is updated in the incarnation table immediately (step 8.2). Since application messages are still disabled, and the initiator will wait for replies from all tasks, the global knowledge of task incarnation numbers will be known to the task when the application messages are enabled. Therefore, there is no possibility of receiving a pre-rollback message before all incarnation information has been made globally available.

Steps 9 and 10 deal with the elimination of backward dependencies. These are messages which have been buffered during the initial execution into the checkpoint CP because of a backward dependency. The messages contain the incarnation number at the time that they were saved. By updating the incarnation number in these buffered messages, the initiating task will not mistake them for pre-rollback messages, upon re-execution. Once the incarnation is updated, the messages are restored into their proper queues for re-consumption. Finally, step 11 enables the application message acceptance. This ensures that the steps between 1 and 11 are uninterrupted by the effects of application messages.

```
Algorithm Participate_Rollback{
1    disable application message acceptance
2    update incarnation of task that sent rollback
         participation msg in inc_tab_i
3    if n indicates that task should rollback {
3.1      increment current_inc_i
3.2      rollback state to CP indicated by n (n.CP)
3.3      discard all messages saved in any checkpoint
3.4          after the creation of n.CP
3.4      reset value of CCP_i vector to that saved in n.CP
3.5      purge all checkpoints created after n.CP was created
     }
1    for all tasks T_j { /* broadcast rollback msg */
1.1      include current_inc_i in n
4.2      include unique_task_id in n
4.3      send message n to task T_j
     }
     /* update incarnation info */
5    for all tasks T_j (T_j ≠ original sender of rollback message n){
5.1      wait for reply r from T_j
5.2      inc_tab_i[j] = r.current_inc
     }
6    if rollback occured in this task{
6.1      update incarnation information in messages saved in CP
6.2      restore messages saved in CP
     }
7    enable application message acceptance
}
```

Figure A.2: MRLV rollback participation algorithm

## A.3 Rollback Participation algorithm

The algorithm for rollback participation described in section 3.7.2 is shown in Figure A.2 with step numbers added to it.

This algorithm is performed by all tasks upon receipt of a rollback message from either a participating task, or a rollback initiator task which has performed the algorithm described in the previous section (the rollback message would correspond to the message send in step 7.3 of the rollback initiation algorithm or the message send in step 4.3 of this rollback participation algorithm). All tasks except for the initiator of rollback *participate* in the rollback whether their state is actually rolled back or not.

Step 1 of the algorithm disables application messages for the same reason as the rollback initiation algorithm. Step 2 is identical to that of the rollback initiation algorithm.

In step 3, the message $n$ corresponds to the rollback message which has been received by some remote process causing this algorithm to be invoked. The message $n$ includes the SIC id of the recovery line being rolled back. It is this id which is used in step 3 to determine whether or not the receiving task of $n$ (i.e. the one performing the rollback participation algorithm) has any checkpoints which form part of the recovery line being rolled back. Only if the task is to roll back, steps 3.1 through 3.5 are performed. These steps are identical to steps 2 through 6 in the rollback initiation algorithm.

Whether the task actually rolls back its state or not, it must still communicate with other tasks to distribute the global incarnation information. Therefore, steps 4.1 through 4.3 which are identical to steps 7.1 to 7.3 in the rollback initiation algorithm must be performed.

In step 5, the participator waits for a reply from all tasks except the task from which it received the rollback message $n$. Steps 5.1 and 5.2 are identical to steps 8.1 and 8.2 in the rollback initiation algorithm.

If a rollback has actually incurred due to the satisfaction of the condition in step 3, then steps 6.1 and 6.2 are performed. They are identical to steps 9 and 10 in the rollback initiation algorithm.