Experimental Evaluation of Distributed Rollback and Recovery Algorithms

Chris Passier

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirement
for the Degree of Master of Sciences at
Concordia University
Montréal, Québec, Canada

May 1988

ISBN 0-315-64679-9

ABSTRACT

Experimental Evaluation of Distributed Rollback and Recovery Algorithms

Chris Passier

One of the aims in the design of Distributed Computer Systems (DCS) is to achieve relatively better system reliability. Software reliability is an essential part of the total system reliability. One way to gain software reliability is through the use of rollback and recovery, a method of fault tolerance. The bulk of the literature pertaining to distributed rollback and recovery is more theoretical than practical, in nature.

We have designed and implemented a Rollback and Recovery Kernel (RRK), used in the experimental evaluation of two rollback and recovery algorithms. In this thesis, we discuss some of the critical design issues and implementation details which must be addressed in a work of this kind. The implementations of the rollback and recovery algorithms are evaluated through the use of a distributed application program. Results of the experiment are presented and interpreted.

Distributed computers are often crucial to such real-time applications as avionic control, nuclear power plants, and various other process control tasks. The importance of these tasks requires that real-time systems be designed to guarantee very high reliability. With this in mind, we examine the real-time computing environment with respect to the use of rollback and recovery as a means of providing real-time software fault tolerance. This aspect of the thesis needs further study.

To my wife

Martine

and to my mother

Corrie

ACKNOWLEDGEMENTS

The implementational work, reviewed in this thesis, was made possible by the combined efforts of the following people: Joachim ("Akis") Hamamjoglu, Prof. T. Radhakrishnan, Prof. H. F. Li, and Dr. K. Venkatesh. Together, we took on the task of setting-up the rudiments of a distributed workshop, at a time when the RRK was nothing more than a twinkle in Venkatesh's eye. Akis was my comrade in the wars against the dreaded "distributed bug". Through adversity, I believe we've forged a lasting friendship. Profs. Radhakrishnan and Li provided us with judicious amounts of "carrot" and "stick", always keeping us headed in the right direction. I thank them for their dedication to having a job well-done, and for their patience. Dr. Venkatesh proved to be invaluable as a "walking" source of references on anything to do with distributed computing. Over the many months, and long hours of consultation and conversation, he also became a good friend. I must reserve a special thanks for Prof. Radhakrishnan, my supervisor, who not only showed me how to write in English, but who was always there when I needed him, even during the most trying of times. His sound judgement was always appreciated.

I would also like to thank my brother in law, John Prokopenko. John was the one who first got me interested in computers. He helped me write my first assembly language program (over the phone), and has continued to help me right up to the present, running around to print out the figures for this thesis.

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

AS	Algorithm Specific (software)
BCS	Briatico, Ciuffoletti, and Simoncini (algorithm)
BSD	Berkeley Software Development
CCP	Current Checkpoint vector of a Process (RLV)
CM	Checkpointing Module
DARPA	Defense Advanced Research Projects Agency
DCS	Distributed Computer System
EDM	Error Detector Module
EHM	Error Handler Module
FIFO	First In First Out
ICB	Input Channel Buffers
ID	Identification
IPC	Inter Process Communications
K counter	Checkpointing counter (BCS)
LAN	Local Area Network
LLCP	Low Level Control Procedure
LPIset	List of Processes Invited (in BCS rollback)
Mbps	Mega bits per second
PEset	Set of Process names (in BCS checkpoints)
RC	Response Checkpoint
RCP	Received Checkpoint vector of a Process (RLV)
RCtbl	Response Checkpoint table
RI	Response Initiator
RLV	Radhakrishnan, Li, and Venkatesh (algorithm)
RRK	Rollback and Recovery Kernel

RRM	Rollback and Recovery Module
RTDCS	Real-Time Distributed Computer System
RTRRK	Real-Time Rollback and Recovery Kernel
SIC	Self Induced Checkpoint
SICtbl	Self Induced Checkpoint table
STE	State Transition Event
STM	Space Time Model
TCP	Transmission Control Protocol
TTBL	Time Table
TVEC	Time Vector
UDP	Unacknowledged Datagram Protocol

Chapter 1

Introduction

1.1 Software Fault Tolerance

The reliability of computer software has long been of interest to computer scientists. Essentially, it is the presence of residual design faults which can cause a software to be considered unreliable. To make more reliable software, either the occurrence of these faults should be prevented or techniques must be incorporated into the software to ensure an acceptable level of performance in the presence of such faults. These two approaches have been termed "fault intolerance" and "fault tolerance" respectively [3].

Fault intolerance is practised by all computer programmers, to varying degrees. Through various software engineering methodologies, we try to produce programs which are free of faults. However, even the most carefully planned and tested systems are destined to contain some residual design faults. Only through rigorous mathematical proofs of correctness can a software be guaranteed to be void of design faults. Such proofs, due to their difficulty, are only feasible for very small programs. Therefore, for most software systems, fault tolerance techniques must be added in order to achieve a greater reliability.

Two methods are used to implement software fault tolerance, "forward error recovery" and "backward error recovery".

In forward error recovery, the system attempts to cancel the effects of a fault which has occurred, by following alternate modes of computation. The most common approach used for this method of fault tolerance is N-version programming.

In N-version programming, a specified task is computed concurrently by two or more programs of independent design. The results of the computations are frequently compared and some criteria (e.g. "majority rules" or some other easily implemented criterion) are used to determine the correct result. With this arrangement, it is presumed that if a fault should occur in one of the software modules causing an error in its computation, its result would be rejected on the basis of it being different from the others'.

The advantage of this method lies in its uniform speed of computation in the presence or absence of faults. Not surprisingly, this is the chosen fault tolerance method in many critical real-time applications. The disadvantages of this approach are the high cost of the additional hardware modules needed to run the different versions of software, the logistics of making sure that the software modules are properly synchronized into identical communication and computational patterns, and the difficulty of developing programs of independent design.

The last point is perhaps the most important since recent publications have pointed out that making programs of truely different design is difficult, and that some of the same design faults appear coincidently in independently designed programs [17,18]. Therefore, the robustness of fault tolerance through N-version programming is questionable; it is conceivable that two versions harbouring a common design fault could override a correctly executing third version.

In backward error recovery, the state of the system is saved, at regular intervals, in stable storage structures called "checkpoints". If a fault occurs and the system fails, the state can be restored to an error-free condition by selecting and loading one of the saved checkpoints and restarting the system. This action is referred to as "rollback and recovery". Following the rollback, during reexecution, an "alternate software module" may be substituted for the faulty "primary module" as is described in the "recovery block" model [14]. If the alternate software module should prove to contain the same or another design fault, a second alternate, of more primitive design, may be employed. This second alternate would offer a degraded service but would be guaranteed to produce a satisfactory result.

The time required to checkpoint, select and load a checkpoint, and reexecute the rolled back computation is a definite overhead incurred in this approach. To minimize the checkpointing overhead, a process may record only the changes made to its data rather than saving the full process state. Upon rollback, these changes would be undone to regain an error-free state. Alternatively, if there is a greater knowledge of the program behavior, optimized checkpoints may be used which contain only the data which will be altered between checkpointing actions.

The checkpointing frequency is application dependent. A suitable rate is achieved by weighing the need to minimize the amount of computation to be rolled back, or "rollback distance", against the overhead of checkpointing. The more frequent the checkpointing action, the shorter will be the distance from the fault to the last valid checkpoint.

The advantages of the rolback and recovery approach to fault tolerance are that no extra hardware need necessarily be added, and the detection of faults can be actively pursued through special-purpose fault detection software.

In this thesis, we are concerned with rollback and recovery, particularly in the distributed processing environment.

1.2 Fault Tolerance in Distributed Systems

A Distributed Computer System (DCS) is made up of a finite number of sequential processes, which communicate only via message passing. This interprocess communication takes a finite, but variable, amount of time. The processes do not have a shared memory, nor do they have a global clock.

Like all parallel processing arrangements, a DCS offers a potential speedup in computation. In addition, a DCS adds the ability to connect geographically separate processors. This latter characteristic enables a DCS, with a dynamic task allocation mechanism, to exhibit a graceful degradation in computation upon the occurrence of such hardware related faults as local power failures. However, for the purpose of this thesis, it is assumed that the hardware is dependable and that the number of processes remains static for the duration of a computation.

In distributed systems, the faulty state of a process, may, through the transmission of erroneous data, contaminate the computations of other processes. In this case, more than one process may have to be rolled back in order to undo the effects of the fault. If the checkpointing actions of the different processes are carried out in an unrelated manner, the rollback of a process might result in an avalanche rollback of the entire system, known as the "domino effect". To illustrate this effect, we refer to the Space Time Model (STM) [1], shown in Fig. 1.1. Here, the progression of three processes is depicted along the time axis and the messages exchanged between them are represented by directed arrows. If, due to the detection of a fault, process P2 should roll back to its third checkpoint, it would force processes P1 and P3 to roll back, to their second and third checkpoints respectively, since the messages M8 and M9 may have been sent in error. However, due to the interaction between P2 and P1 through messages M6 and M7, P2 is forced to rollback to its second checkpoint. This continues until, in this case, the three processes are forced to roll back to their initial states, undoing possibly a large portion of valid, error-free computation. A systematic study of the causes of the domino effect, as well as a summary of the approaches which can be used to prevent it, are given in [23]. In general, to prevent the domino effect from occurring, the checkpointing actions of the processes must be coordinated in some manner in order to form consistent global states called "recovery lines", which can be used during subsequent rollback operations to minimize unnecessary rollback.

Distributed rollback and recovery algorithms can be classified into two broad categories, "preplanned" and "unplanned" recovery algorithms [27]. In the former case, recovery lines are objectively formed at the time of checkpointing, while in the unplanned case a recovery line is deduced at the time of rollback and, as we have seen, this may result in the domino rollback effect.

Another way in which these algorithms can be classified is based

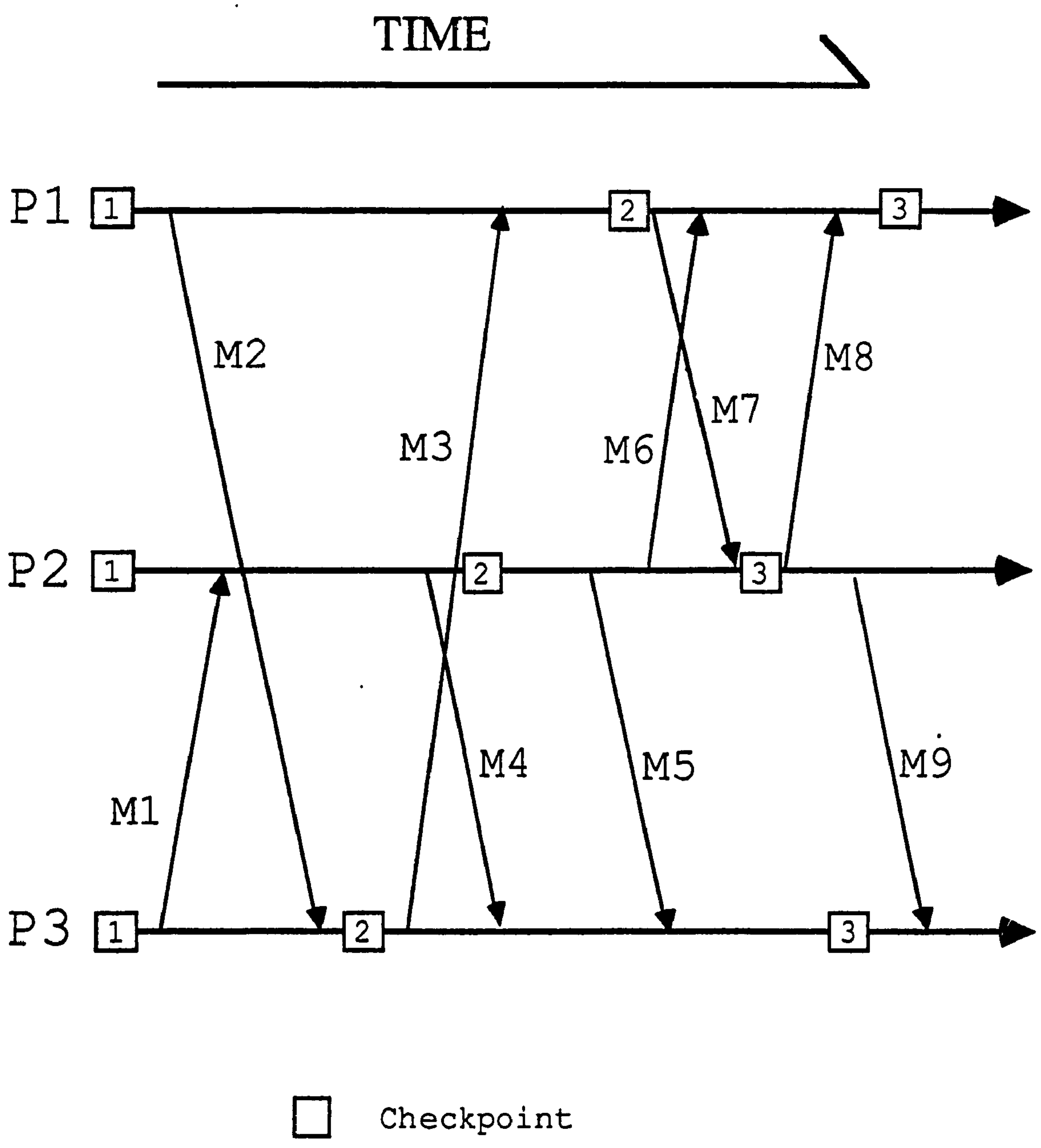


Figure 1.1 Uncoordinated checkpointing in a distributed system

on their manner of checkpointing. The checkpointing can be enforced intrusively or non-intrusively. When done intrusively, checkpointing perturbs the underlying application program. For example, in Randell's "conversation" scheme [22], checkpoints are taken in a predetermined synchronous manner and interprocess communication is restricted to pre-set patterns. While with the non-intrusive variety of algorithms, checkpoint creation is coordinated dynamically with no restrictions on interprocess communications.

The rollback and recovery algorithms studied in this work are of the preplanned recovery, non-intrusive checkpointing variety. These algorithms create checkpoints dynamically, in response to application messages, and are hence referred to as "dynamic checkpointing" algorithms.

1.3 Real-Time Computer Systems

Nowhere have computers become more indispensable than in the real-time domain. The evolution of real-time computing technology and, more specifically, Real-Time Distributed Computer Systems (RTDCS) has not only enabled the solving of problems which are too complex to be handled by conventional electrical or pneumatic analog devices, but has actually opened new avenues in the fields of science and engineering [24]. Today, RTDCS is crucial to such applications as avionic control, hospital life-support systems, military systems, nuclear power plants and various other process control tasks.

The importance of the tasks to which real-time systems are typically assigned require that they be designed to guarantee very high reliability. As was outlined earlier, both fault intolerance and fault tolerance must be practised in the design of reliable software. In the real-time domain, both of these design approaches are made more difficult by the introduction of timing constraints.

In the non-real-time environment, the execution speed of a process need not be considered in order to determine its validity. The validity of a sequential program can be deduced from the static program text and the set of possible input data. With the addition of synchronization signals and mutual exclusion the validity of coordinated program modules can be similarly established. No assumptions about the speed of execution need be made other than it being greater than zero. In the real-time environment, however, a computation is deemed valid if, and only if, the result is correct and it is produced on time [26].

This timing constraint, and the fact that real-time systems often carry out several tasks concurrently [9], requires that, in attempting to design fault-free real-time software, i.e., fault intolerance, one must take into account such things as the raw processor speed, the processor scheduling policy of the operating system and, in the case of RTDCS, the delay of interprocess communications. These additional considerations make the designing and the debugging of real-time software especially difficult [12], and has resulted, for the most part, in ad hoc solutions for individual design problems [20].

Any approach to fault tolerance in this environment must cope with the same design problems as in the case of fault intolerance. With respect to rollback and recovery, the real-time systems must be able to detect a fault and reexecute the computation before a response is required by the controlled object. The matter is further complicated by the fact that any erroneous results which have been output to the environment cannot be rolled back, and the resulting effects cannot be undone.

1.4 Outline of the Thesis

In this thesis, we are more interested in the practical, as opposed to the theoretical, aspects of rollback and recovery as a means of software fault tolerance.

In the non-real-time environment, we first describe (in Chapter 2) a Rollback and Recovery Kernel (RRK) designed to offer the following: a practical test-bed environment for the implementation and testing of dynamic checkpointing algorithms; a flexible rollback and recovery software which may be easily used by future application programmers. Ironically, before one can use a fault tolerant RRK one must get the bugs out of it. For this purpose, we developed a distributed version of J.H. Conway's "Game of Life" [11], called "N-Life". This test-bed application program proved to be a flexible, and an easy to use, debugging and testing tool (Chapter 3). We go on to describe the comparison of two dynamic checkpointing algorithms, RLV [23] and BCS [5], to gain insight on their relative performances as well as to gain some experience on the overhead due to rollback and recovery in general (in Chapter 4).

With respect to real-time computing (in Chapter 5), we offer a brief discussion on the more salient aspects of this environment, in particular with respect to rollback and recovery as a means of software fault tolerance. We also introduce a new model for real-time fault tolerance, the "Coexistence Model". The impetus for the latter is what is seen as an ineffective treatment of the occurrence of sporadic events.

We end the thesis (in Chapter 6) with some conclusions, and suggestions for future work.

Chapter 2

Design of a Rollback and Recovery Kernel for a DCS Environment

The primary goal of this work is to construct an environment in which we can implement and experimentally compare rollback and recovery algorithms in order to gain information on their relative performance characteristics. This work marks the beginning of a series of projects to be carried out in the area of distributed computing. Future endeavors will include research in distributed debugging, real-time fault tolerance, and discrete event simulation. Since some of these efforts will build upon what is achieved here, a second objective of this work is to design a Rollback and Recovery Kernel (RRK) which could be used in future applications.

In this chapter, we first present the design environment of the RRK, namely the chosen DCS architecture and OS software. Then we describe the particular type of rollback algorithms to be implemented. Finally, the top-down design considerations used in the design of the RRK are systematically presented along with the solutions adopted in this thesis.

2.1 Design Environment

The DCS Architecture

The distributed system architecture, used for this work, is composed of twelve Sun 3/50M-4 diskless workstations connected through a high-speed local area network (LAN) to a central file server.

Each diskless workstation is based on the Motorola MC68020 32-bit microprocessor and the 32-bit VMEbus, and is equipped with 4 megabytes of RAM. The non-volatile memory is provided by the remote network file server, the Sun 3/160s, which can be accessed by all the nodes in the system. Interconnecting the workstations and the file server is a 10 Mbps. Ethernet. Fig. 2.1 shows a block diagram of the system.

This, diskless workstation, type of arrangement has become popular with a range of network-based research applications. The centralized file server avoids the problem of each node in the system maintaining its own file systems and thereby duplicating resources, while the high-speed LAN enables low file access times.

Some aspects of this architecture are of particular importance to our RRK design. "Internet" is a multi-network communication protocol developed by the Defense Advanced Research Projects Agency (DARPA). The inter-node communication in the Sun system is based on the internet concept. In this protocol, each inter-node communication is accompanied by the translation of a network-node-name to an internet address. The tables needed for these mappings are maintained in a "Yellow Pages" read-only database. In our diskless workstation environment, the lack of disk space on each node requires this database to be kept exclusively on the network file server. The unfortunate result of this arrangement is the creation of a communication "bottle-neck" at the point of access

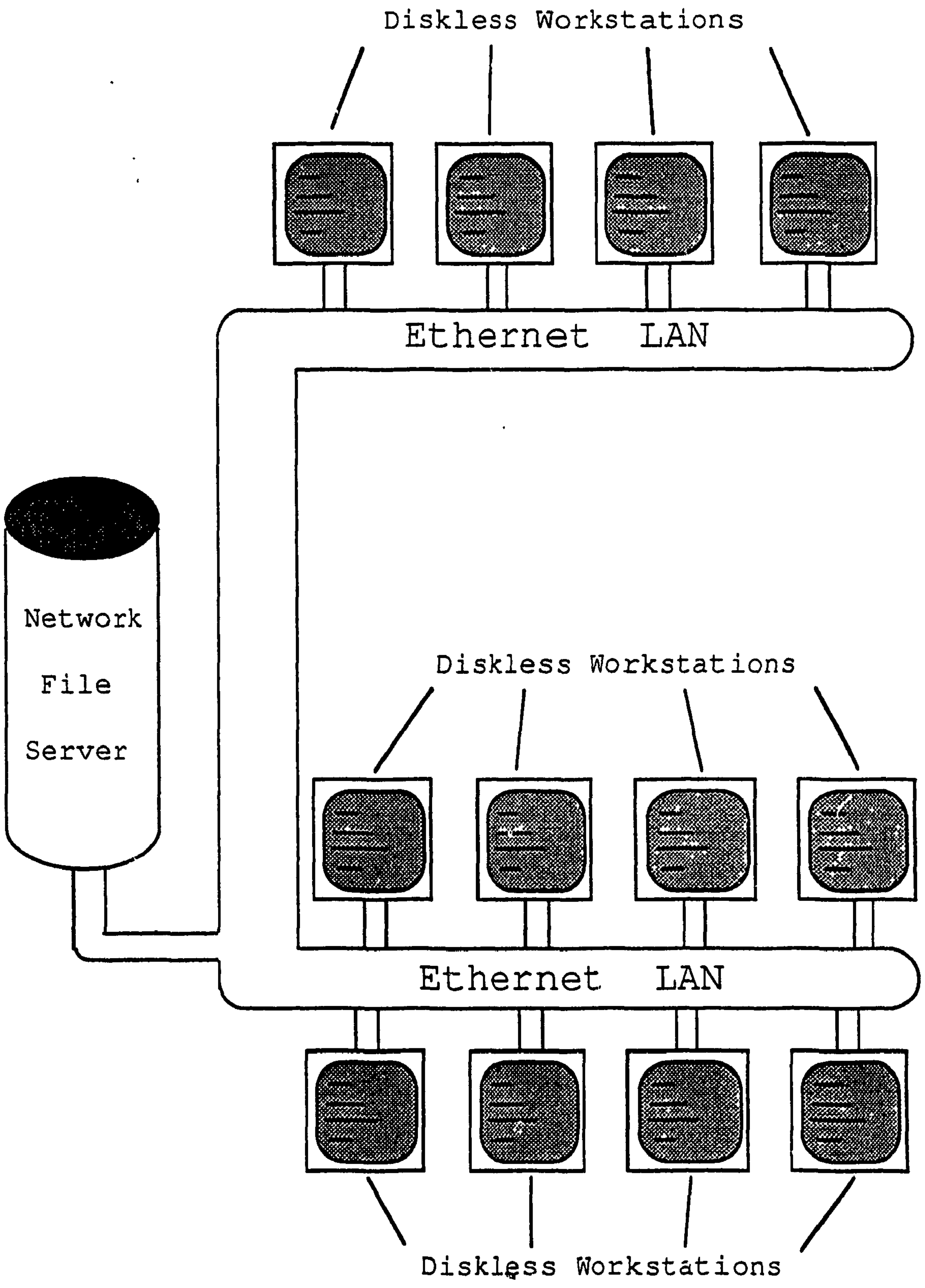


Figure 2.1 DCS architecture

to the Yellow Pages database. Its effects are further discussed in Chapter 4.

Also, the workstations have no local stable store which could have been used for the saving of checkpoints. Since our primary goal was to gather performance measurements, we did not want to exacerbate the bottle-neck situation mentioned above by making the file server also handle the saving and retrieving of checkpoints. We therefore decided to use the workstations' own volatile memories for the storage of checkpoints.

The DCS System Software

The system software used in this work, Sun UNIX, is an enhanced Berkeley (4.2BSD) release. Berkeley UNIX is recognized for its support of standard networking protocols such as the DARPA internet family of communication protocols.

Inter process communications (IPC) are realized through "sockets". Sockets can be thought of as a generalization of pipes, a more familiar UNIX concept. Sockets have the advantage of being able to carry data in both directions and the ability to connect processes running on two different machines. With sockets, one can use both the Unacknowledged Datagram Protocol (UDP) and the Transmission Control Protocol (TCP), at the transport layer.

UDP or datagram sockets enable a fast but unreliable form of communication which retains message boundaries. On the other hand, TCP or stream sockets enable reliable FIFO message passing, but may take longer than datagram messages and do not retain message boundaries.

Another IPC facility, available in Berkeley UNIX, which is useful to researchers is the "select" system call. Among other things, this select facility allows one process to test for the presence of input from one or more sources before actually attempting to read any data.

BSD UNIX also offers a more robust **"signaling"** mechanism. Signals are modeled after hardware interrupts, and this mechanism permits processes to detect and handle signals at any time. The signal handling mechanism can handle multiple signals.

2.2 Type of Rollback and Recovery Algorithms Implemented

The two algorithms implemented in this work are the RLV algorithm, due to Radhakrishnan, Li, and Venkatesh [23], and the BCS algorithm due to Briatico, Ciuffoletti, and Simoncini [5]. Both algorithms are dynamic checkpointing algorithms, as classified in Chapter 1. These algorithms place few constraints on the distributed application. The application program is viewed as a collection of interacting asynchronous processes, which have the ability to roll back their computations. A reliable and FIFO message passing mechanism is also assumed.

Two sorts of checkpoints are taken by these algorithms, Self Induced Checkpoints (SIC) and Response Checkpoints (RC). SICs are established in response to an explicit request from the application program. The application makes this request at its own convenience, with no knowledge or concern for checkpoints taken by other processes in the distributed system. RCs, on the other hand, are taken implicitly by the RRK upon receipt of messages containing information about newly established SICs on other processes. The RCs are taken transparently without the awareness of the application program. The rules governing the creation of RCs are specified by the particular algorithm being used (e.g. RLV or BCS).

The characteristics which distinguish one dynamic checkpointing algorithm from another are found in their rules for RC creation, the manner in which the algorithms use SICs and RCs to form their preplanned recovery lines, and the way in which they execute their rollback and recovery upon detection of a fault. The latter includes the use of some sort of special rollback control messages.

In Fig. 2.2, a STM is shown where three processes exchange application messages. Here, the SIC established by process 2 and the resulting RCs on processes 1 and 3, due to messages M6 and M5 respectively, form a valid recovery line. This recovery line may be used by the system during rollback.

Two typical characteristics of dynamic checkpointing algorithms are exhibited in this figure. Firstly, we see that RC checkpoint creations coincide with the reception of certain application messages. Secondly, the figure shows that some messages (e.g. M3 and M4) intersect the recovery line. These messages would not be re-sent upon rollback to this recovery line. However, the notion of their being sent exists in process 1. To retain consistency among the rolled-back processes, these messages are stored in a checkpoint of the receiving process (e.g. the SIC of process 2) to be "played-back" during the rollback procedure.

It will be seen how these characteristics affect both the design of the [illegible]RK described in this chapter and the choice of a test-bed application described in Chapter 3.

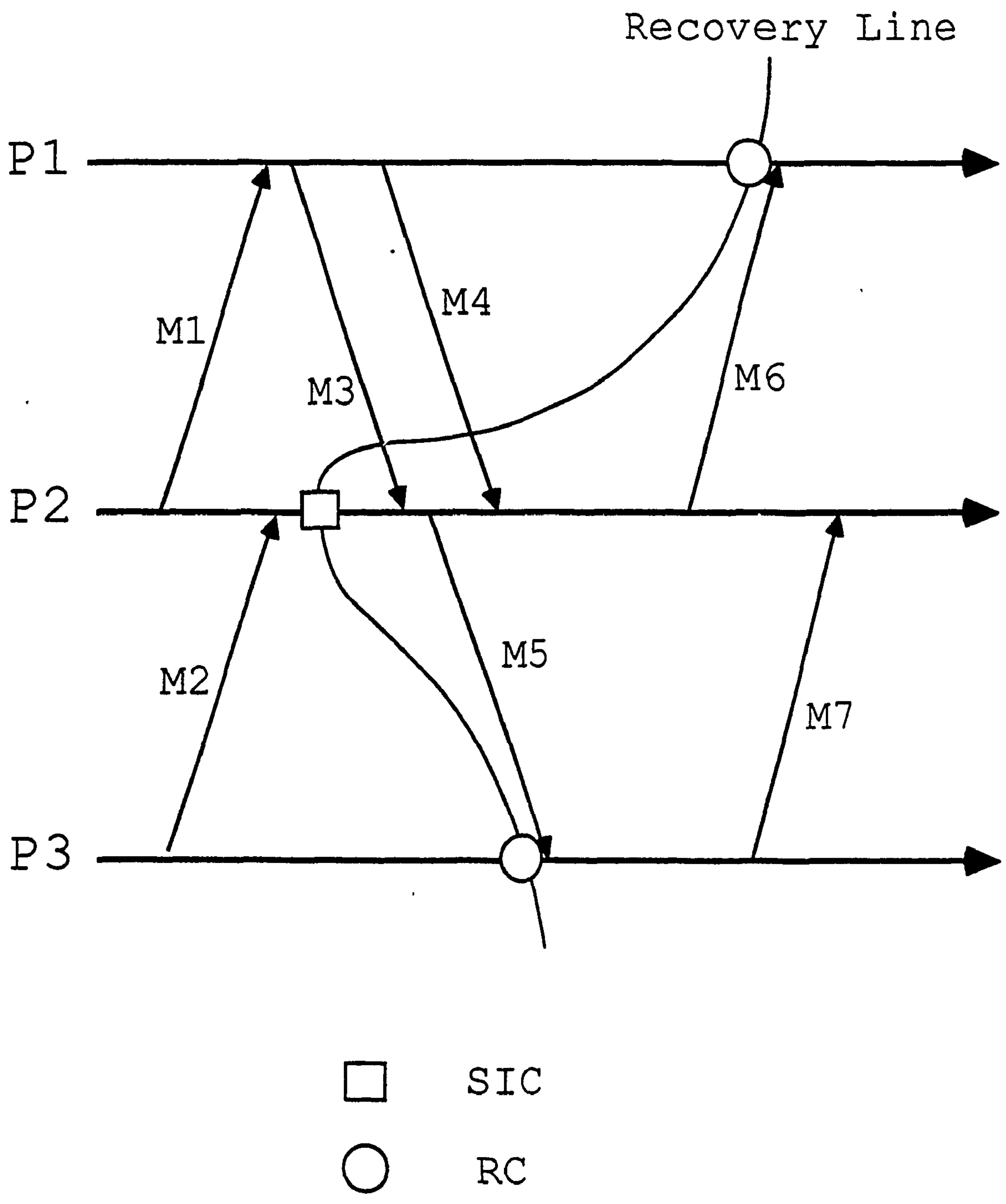


Figure 2.2 Recovery line via dynamic checkpointing

2.3 Design of the RRK

Ideally, the RRK should be considered by the user as a module of OS software, hence the usage of "Kernel" as in UNIX. In this work, we have endeavored to make the RRK a convenient receptacle for the dynamic checkpointing algorithms mentioned earlier, while keeping it easy to use by future users.

In order for the RRK to determine when an RC is to be taken, some control information reflecting the global checkpointing status must be conveyed from one process to another. With the transmission of each application message in the system, the RRK at the source node of the message piggybacks control information just prior to the sending of that message. The RRK at the destination node strips-off this data before delivering this message to the local application process.

This piggybacking/stripping-off behavior necessitates the RRK to be logically situated between the communication layer and the application program. There are basically three ways in which this could be implemented as is shown in Fig. 2.3.

(i) One approach would be to have the RRK exist as a unique process on each node in the system (Fig. 2.3a). This arrangement would require the creation of additional communication channels to integrate the RRK and a message routing protocol which may be difficult to hide from the application. Also, since a message passed between two nodes in the system would be associated with multiple process switches in the two CPUs, the added time delay would be a definite overhead.

(ii) A more desirable tack is illustrated in Fig. 2.3b. In this case the RRK is integrated as part of the existing OS software (i.e. Sun UNIX), where it could have easy access to the lower levels of the

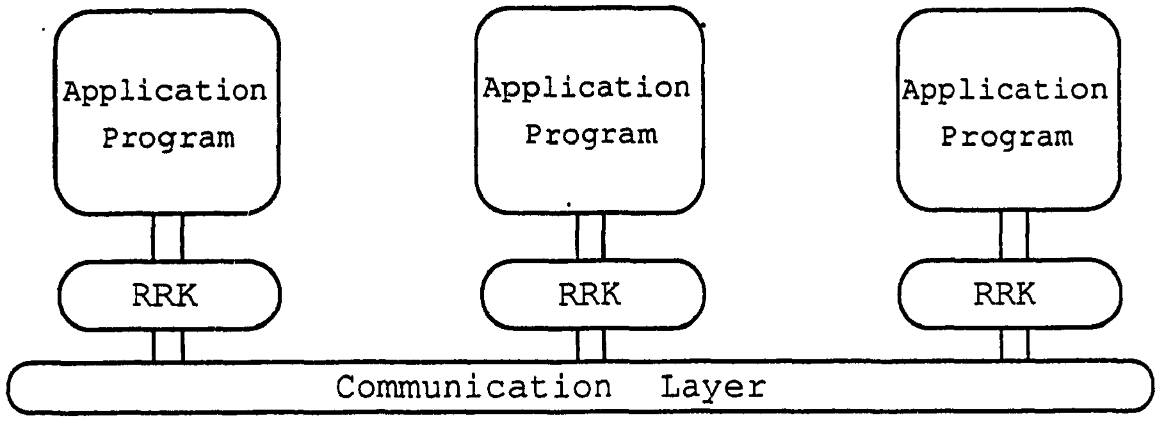


Figure 2.3a RRK as a separate process

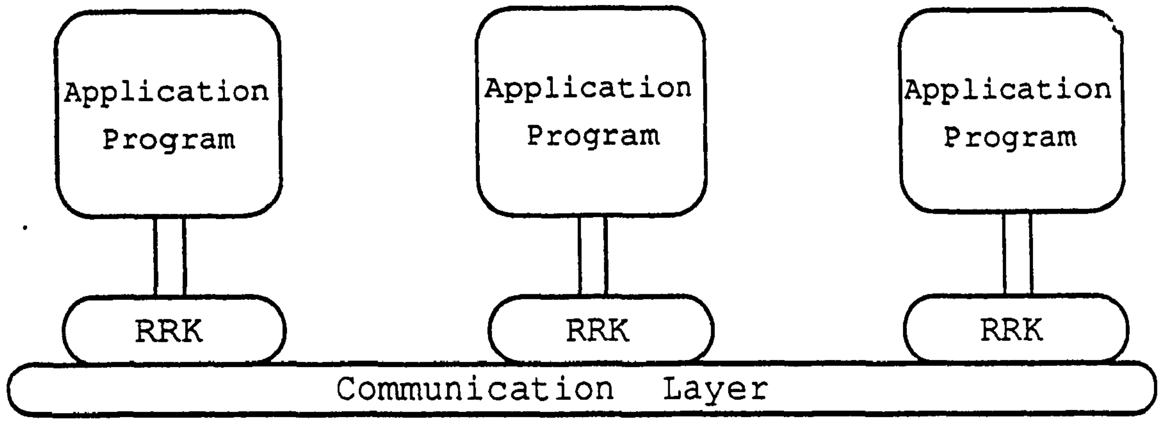


Figure 2.3b RRK integrated with UNIX OS

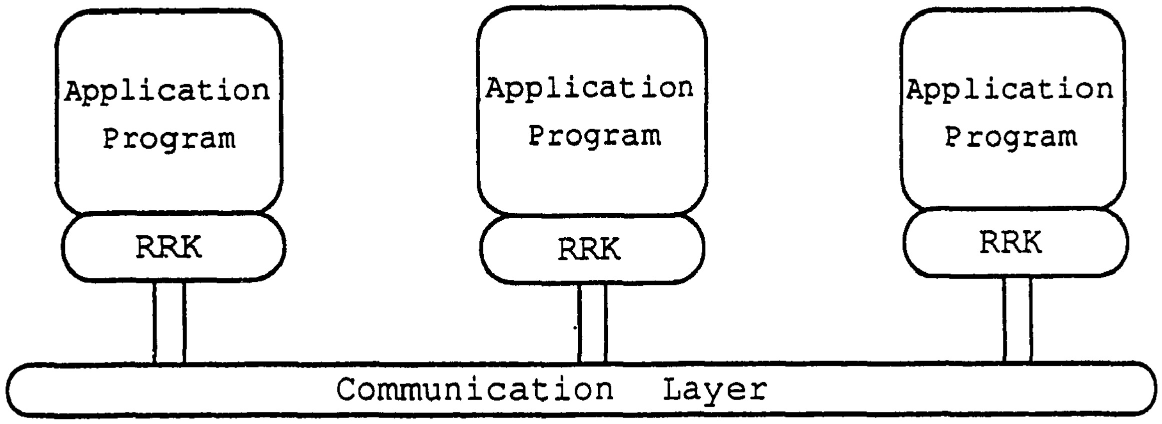


Figure 2.3c RRK appended to application

communication layer and function transparently with respect to the application. This would involve a major development, involving the UNIX source code, which we chose to defer to a later stage.

(iii) In our design a copy of the RRK software is appended to each participating application process as is shown in Fig. 2.3c. In terms of true fault tolerance, this option is not very robust since execution errors causing the OS to interrupt the application may also result in the RRK itself being aborted. However, for the purpose of performance analysis, this is sufficient to reveal the overhead cost in rollback and recovery, bearing in mind it could be reduced further if approach (ii) is followed.

Fictitious faults are "detected" by our own Error Detector Module (EDM) which then alerts the RRK through a user-defined signal. This arrangement serves the purpose of a performance analysis environment, which is our primary goal.

As far as making the RRK into a resident facility that could be used by other application programmers, we can ask the following: What is a reasonable amount of effort to be expected of the application programmer, in order to make his program fault tolerant? Should it be a simple matter of him pushing a button marked "Fault Tolerance"? Or, should it be that someone wishing to use this facility be well-versed in the intricacies of distributed fault tolerance and on whom we can place the lion's share of the fault tolerance programming responsibilities?

Neither extreme seems feasible. The, oversimplified, first approach would require a very general RRK design. In computer science generality begets enormity and, in this case, would result in a RRK which would cause an unacceptable slow-down in the execution of the application. The other extreme would result in only a select group of users being able to use the facility, which is not our intent.

In our design, an effort was made to make it possible for the application programmer to use the services of the RRK through a relatively simple and well-defined procedure, while still retaining a rollback kernel which could perform efficiently. This was achieved through a breakdown of the design into two sections. The first is composed of those parts which are application invariant (the RRK proper). The other deals with the tasks to be carried out by the user in order to link his program to the RRK.

2.4 Application Invariant Part of Design

In this section we find those parts of the design which are application invariant. The application programmer wishing to use the RRK facility treats this as a "black box", but he respects the proper interface requirements.

The invariant part is made up of two modules, the Checkpointing Module (CM) and the Rollback and Recovery Module (RRM), as shown in Fig. 2.4. Messages are sent onto the communication layer via the CM. All incoming messages are first filtered by the RRM, for rollback control messages. The application messages are forwarded to the CM through the RRK's Input Channel Buffers (ICB).

As seen in Fig. 2.5 and Fig. 2.6, both modules (CM and RRM) contain portions of Algorithm Specific software, symbolically denoted as AS. This software contains the constructs which are changed when implementing the different dynamic checkpointing algorithms (e.g. RLV and BCS). Therefore, the AS software is algorithm specific but

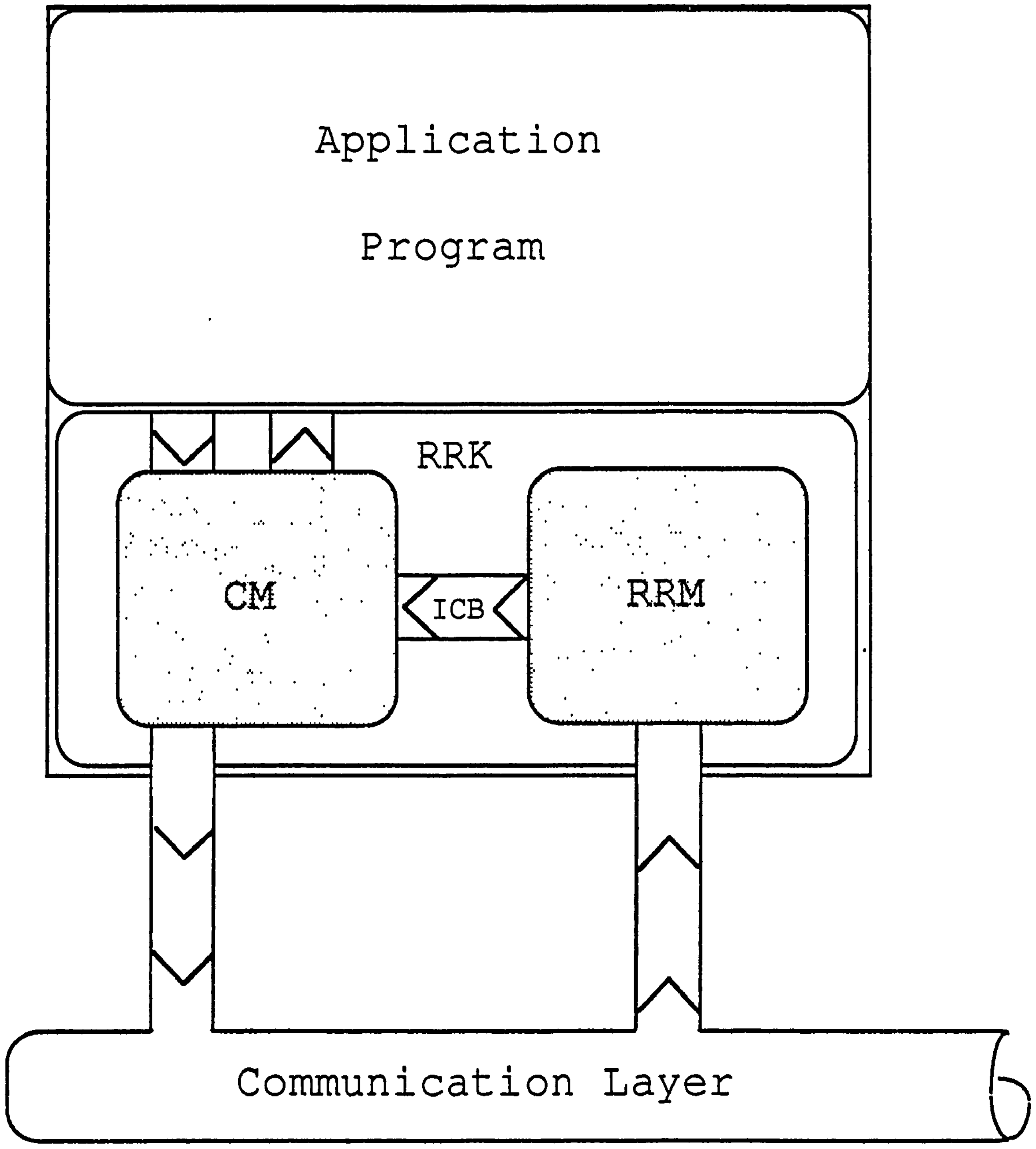


Figure 2.4 Modules of the RRK

application invariant. The AS software will be shown in more detail in Chapter 4.

The CM, shown in Fig. 2.5, is activated by the application program through one of four function calls: Kwrite, Kselect, Kread and Chckpnt. The first three perform essentially the same functions as their namesakes (Write, Select and Read) as far as the application is aware, and are infact substituted for the originals within the application code on a one to one basis. The actual role of these RRK interface functions will now be briefly outlined.

Kwrite accepts messages from the application program, to which it appends the checkpointing control information before sending it on to the communication layer. Depending on the rollback algorithm in use, Kwrite may make use of AS functions, as will be seen later.

When Kselect is called, it examines the ICB for the presence of messages and the source ID of messages, and then relays this information on to the application program.

If the application chooses to read any of the messages in the ICB, it executes a call to Kread. This function strips off the appended control information. In conjunction with AS software, it also determines if any RC checkpoints are to be taken. The AS software also decides if copies of this message should be stored into previously saved checkpoints (e.g. in the case where it crosses recovery lines). Kread finally delivers the application's portion of the message to the application program.

Chckpnt is called when the application wishes to take an SIC. More information on the use of the Chckpnt function is found in the following section.

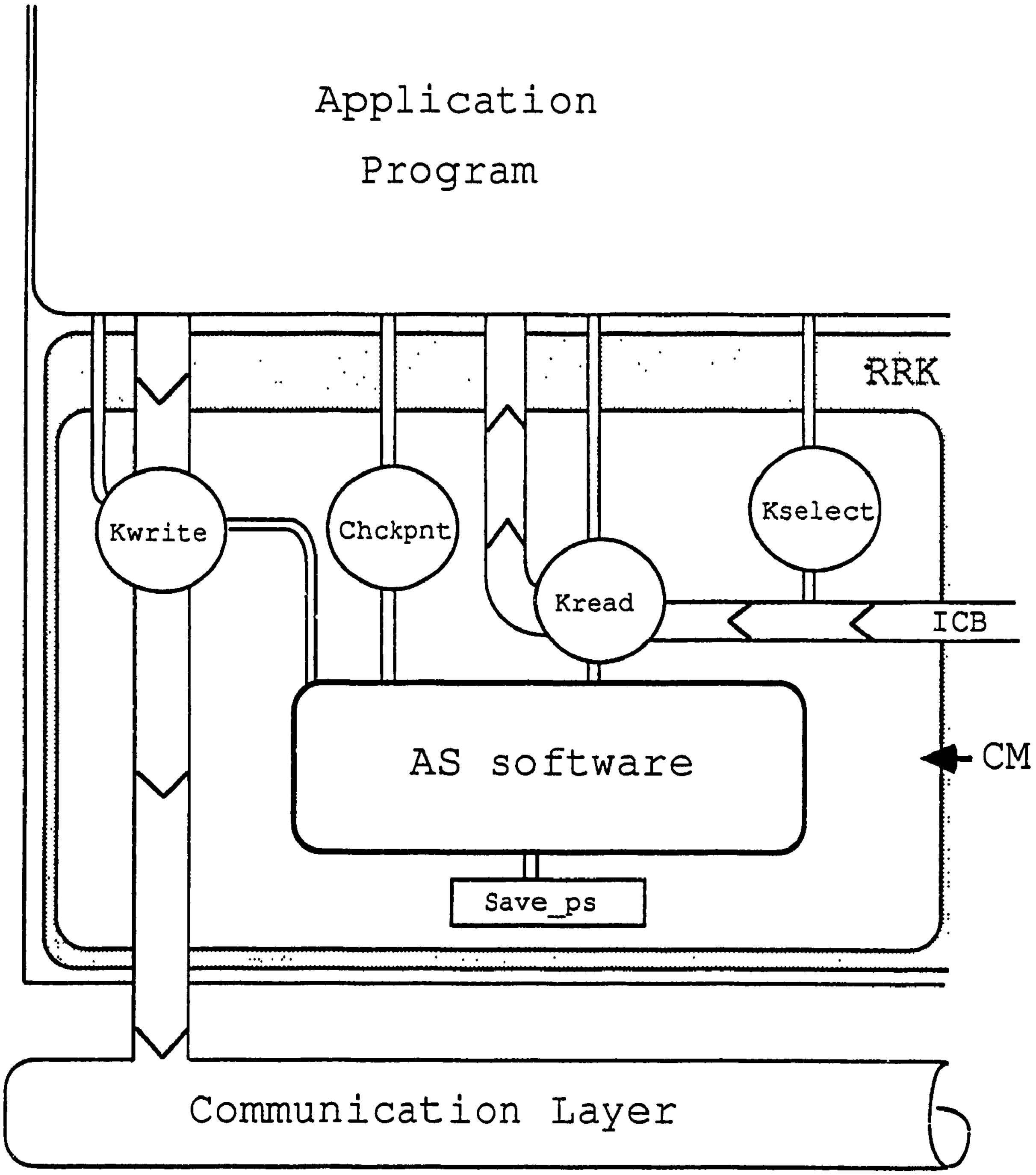


Figure 2.5 Checkpointing Module (CM)

When a checkpoint is to be taken, whether it is a SIC or a RC, the application's process state must be saved as part of the checkpoint. To do this, the AS software calls the Save_ps function. This function is actually part of the application dependent software, as wil be seen later.

The rollback and recovery module or RRM, shown in Fig. 2.6, is responsible for rolling back the application program upon the detection of an error in the system. This rollback and recovery procedure is governed by AS software and can be triggered by one of the two following means. The local EDM, when it detects an error, may activate the AS software by an interrupt mechanism, in which case this node is the "initiator" of the distributed rollback procedure. Alternatively, the RRM may learn of a rollback, initiated by some other process in the system, through a rollback control message, and hence activate the AS.

Both scenarios require the RRM to quickly respond to the triggering mechanisms in order to minimize redundant application computation. As was mentioned earlier, BSD UNIX allows for immediate handling of signals, so, in the case where the local EDM triggers the RRM, the AS rollback logic may be instantly activated. The EDM's error detection signal may, however, be temporarily blocked by the RRK if the RRK is already executing some other action, such as checkpointing, storing of messages, processing a control message from a simultaneous rollback procedure etc.. Such RRK actions must remain atomic in order to retain consistency among the RRK's data structures.

In the case where the RRM learns of a rollback through a rollback control message, one of the following two methodologies could be implemented to enable the timely detection of such messages. Either the

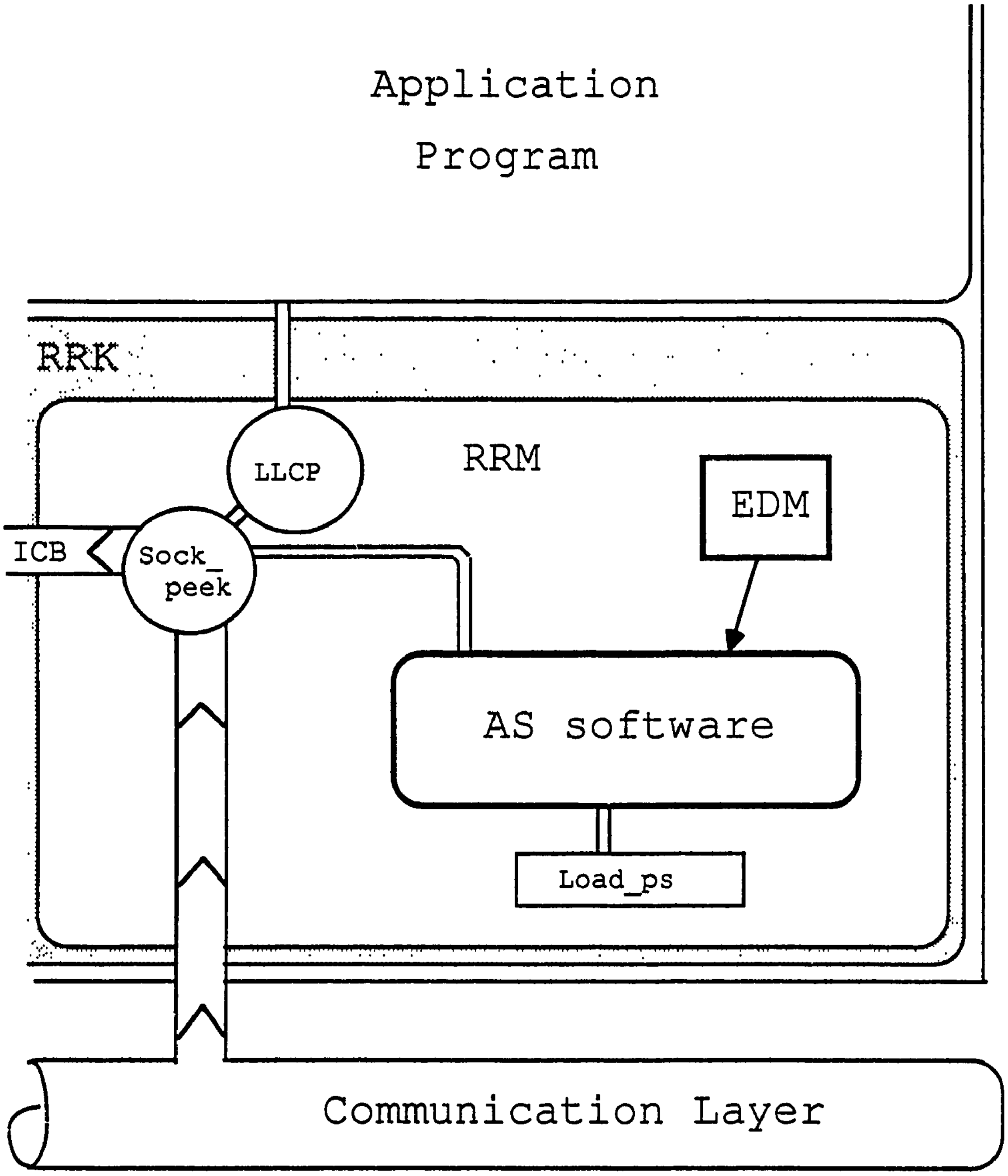


Figure 2.6 Rollback and Recovery Module (RRM)

application interrogates the communication layer through frequent polling; or the communication layer itself raises an asynchronous interrupt to herald the arrival of a message. In either scenario, the Low Level Control Procedure (LLCP), shown in Fig. 2.6, is activated. This function uses the "select" system call, mentioned earlier, to determine the source channels of any queued messages. The LLCP then calls the Sock_peek function, which analyses these messages. If a message is an application message, it is stored in the ICB for later use. If a message is a control message, it causes instant activation of the AS.

If the rollback logic in AS determines that the local application process should roll back, then the application's process state is loaded via a call to Load_ps. As with the Save_ps function, Load_ps is an application dependent part of the design.

2.5 Application Dependent Part of Design

This section deals with the steps which must be carried out by the application programmer who wishes to link his program to the RRK. In steps 1 through 3, it will be seen how the link-up procedure has been facilitated through minimal interface requirements. Regarding checkpointing, all decisions involving the placement, and frequency of the SIC creations, and the size and composition of both the SICs and RCs, have been left entirely to the application programmer. Although these steps, 4 through 6, require a larger commitment on the part of the user, the arrangement allows for a more tailor-made RRK design which makes no assumptions on the checkpointing needs of the application programmer.

Once these steps have been completed, the application program can still be used in a manner as if the RRK were not attached. All checkpointing, and rollback and recovery mechanisms will be executed by the RRK without the awareness of the user.

The following assumptions were made about the application program that uses the RRK: the number of processes in the system is static and their topology of interconnection is known to every one of them; each process is assumed to have a unique process identification.

What follows, then, is a step by step outline of the link-up procedure along with some hints on implementation.

Steps to Link-Up an Application Program to the RRK

Step 1: Application Specifications

Application specific data is required by the RRK and should be defined within the application, using a specified nomenclature. These include the following...

APP_MSG_SZ	- the size of an application message in bytes
PROC	- the number of processes in the distributed application
my_job	- the ID number of the local application process
fd[PROC]	- a vector of communication socket file descriptors of size PROC, indexed via process ID

The need for the specification of the application's message size is required for FIFO message passing using sockets. The reader will recall that stream soc'.cs are used for FIFO communication, and that these sockets do not retain message boundaries. Therefore, unless the application programmer informs the RRK of the size of application messages (APP_MSG_SZ), the RRK will not be able to find the appended checkpointing control information within the queued stream of concatenated messages.

"PROC" is used to dictate the size of many of the data structures used by the RRK. The "my_job" data is needed to distinguish the local application process from the others in the system. With "fd[PROC]", the processes' IDs are used as an index to the vector. The vector elements contain the socket file descriptors, which are used to address the communications to the processes of the distributed system.

Step 2: I/O Functions

In the application program, all occurrences of the following system calls: Write, Read and Select, should be replaced by their RRK equivalents: Kwrite, Kread and Kselect. To facilitate this replacement, all arguments normally supplied to the original system calls are also supplied to the RRK versions. These RRK functions also behave as the originals with respect to the application program (this includes all error flags returned) and no special action need be taken to incorporate them within the code.

In this way, the RRK duties, such as the appending and stripping-off of the checkpointing control information and the interim storing of the application messages in the ICB, are executed transparently with respect to the application program.

Step 3: LLCP Insertion

As was mentioned in the discussion about the RRM, the rollback control messages can be detected by either frequent polling or asynchronous interrupts. If the former method is used, then calls to the LLCP function need to be interspersed within the application code. Although the frequency of these calls is not specified, they should be often enough to quickly detect the presence of a rollback message, yet not so often as to degrade the perceived performance of the application. For iterative applications, one or two calls to LLCP per cycle should be sufficient.

Step 4: SIC Creation

At chosen points in the execution of the application, SICs will need to be created. This is done through calls to the RRK's "Chckpnt" function. The question of where to place these function calls may prove difficult to the application programmer.

Since this checkpointing action will entail the saving of the application process state, which is in itself a user defined action (see step 6), it makes sense to have the calls to Chckpnt coincide with points in the computation where the contents of application's data set is predictable and, preferably, small. In an iterative application, this would correspond to the beginning of a new cycle.

When trying to decide upon the frequency of calls to Chckpnt, the particular needs of the application program must be considered. A compromise must be reached between minimizing the rollback distance and minimizing the overhead of checkpointing, through more and less frequent checkpointing, respectively.

Step 5: Setjmp/Longjmp Usage

The Setjmp and Longjmp system calls are supplied by UNIX. Together, they make it possible to "jump" to a position in the code where the program has been previously executing. It is the responsibility of the application programmer to insert a call to Setjmp directly after each call to Chckpnt. Setjmps saves the stack position in a buffer which is supplied as an argument.

Later, upon rollback, a call to Longjmp is made by the RRK. The previously mentioned buffer is supplied as an argument. This action restores control to the position immediately following the call to Setjmp by popping the stack to the saved level.

This facility is of obvious value in rollback and recovery. However, the Setjmp/Longjmp constructs introduce a design restriction, which is summarized in Appendix A.

Step 6: Definitions of Save_ps and Load_ps

The actions of saving and loading the application process state are handled by the application programmer in the functions Save_ps and Load_ps. This arrangement allows for the most efficient use of the RRK. If these duties were left to the RRK, the entire UNIX-defined process state would have to be saved. No optimized data set could be used for fear of omitting data which is needed by the application. Nor could the packing of data be generally applied, since the extra cost in terms of execution time may not be warranted (e.g., in an environment with a wealth of stable storage).

The ease with which this step is implemented depends on the expertise of the individual programmer and the degree of his familiarity with his code. Only he would know which data is likely to be altered, and must therefore be saved, and whether any space saving tactics (e.g., packing) should be employed.

All that is required by the RRK is that a call tc Save_ps should save the appropriate data and then return the address of this saved data. This address is stored by the RRK in the checkpointing action. The address can be later used as an argument in a Load_ps call during a rollback to the selected checkpoint.

Chapter 3

Selection of a Suitable Test-Bed Application

A distributed application program is required for experimentally testing the rollback recovery kernel. We will refer to such a program as a "test-bed application program". Although there exist numerous published distributed algorithms, experiences in the development of distributed programs are not much. There are no established guidelines for the selection of an appropriate test-bed application.

In this chapter we address the question of what constitutes a good test-bed application program. The answer to this question is more qualitative in nature than quantitative. The distributed application program selected, in our case, is developed based on the "Game of Life" proposed by J.H. Conway [11]. Throughout this chapter we discuss the desirable characteristics of a test-bed application program using this example. However, it is not to be concluded that the Game of Life is the most suitable candidate in every respect.

3.1 Algorithm Simplicity

Normally, one expects the application program to be considered as a "black box" by the underlying RRK. Then it does not matter, as far as the RRK is concerned, how complex the distributed application program is. However, debugging a distributed program is quite time consuming, not to mention exasperating, and the burden could be lessened if the algorithms involved in the test-bed application are simple.

The Game of Life is a simple simulation game. It simulates the lives of cellular automata which live and die by certain rules. The game is normally played on a two-dimensional rectangular grid of unlimited size. A location on the grid is termed a cell, each of which has eight neighbors. Given an initial pattern of living cells, the successive generations or cycles of the game can be computed using the following rules [4].

Birth Rule : A dead cell having exactly three living neighbors will be alive in the next generation.

Death Rules : A live cell having four or more living neighbors will be dead in the next generation due to overcrowding.
A live cell with less than two living neighbors will be dead in the next generation due to loneliness.

Survival Rule: A live cell having either two or three living neighbors will remain alive in the next generation.

The surprising results achieved with the above set of simple rules have made the Game of Life very popular with computer enthusiasts over the past many years. Examples of some of the classic "Life Forms" are shown in Fig. 3.1, where we see the Life Histories of various patterns followed through five generations.

Figs. 3.1a and 3.1b are examples of "Spaceships", Life Forms which travel across the playing surface in one direction. The first one (Fig. 3.1a) is a "Featherweight" Spaceship which moves diagonally at one quarter the "Speed of Light" (1 cell/generation in any direction) and is called a "Glider". Fig. 3.1b shows a "Lightweight" Spaceship which moves orthogonally at one half the Speed of Light. In the next figure, we see some examples of small "Oscillators", i.e., "Blinkers", which alternate from vertical to horizontal three-celled lines. The final figure shows a few examples of "Still Lifes", Life Forms which stay unchanged from generation to generation.

Spaceships and Oscillators are periodic. That is, their patterns repeat themselves after a certain number of cycles. This fact was of particular use to us during the testing of the RRK, as will be seen later.

Our initial task was to adapt this algorithm to construct a distributed program called "N-Life". The full grid or global matrix of the game can be divided into rectangular subregions of various shapes and sizes. Each of the processes in N-Life can compute successive generations of one or more of the subregions. Processes, whose respective regions are adjoining, communicate with each other about the status of cells on their joint boundaries.

We assume "wrap-around" (both horizontally and vertically) in the

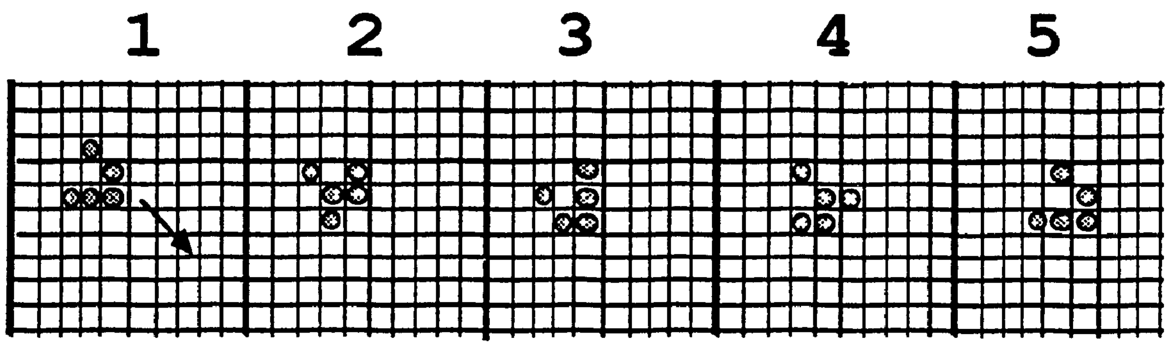


Figure 3.1a A "Glider": a featherweight spaceship of period 4,(moves diagonally at 1/4 the speed of light).

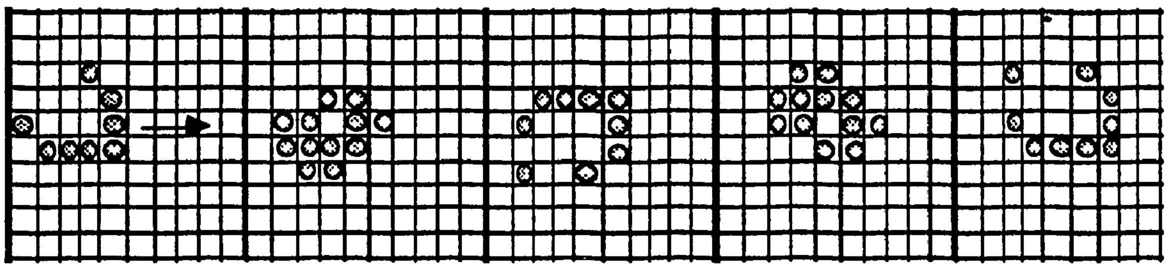


Figure 3.1b A "Lightweight Spaceship" of period 4, (moves orthogonally at 1/2 the speed of light).

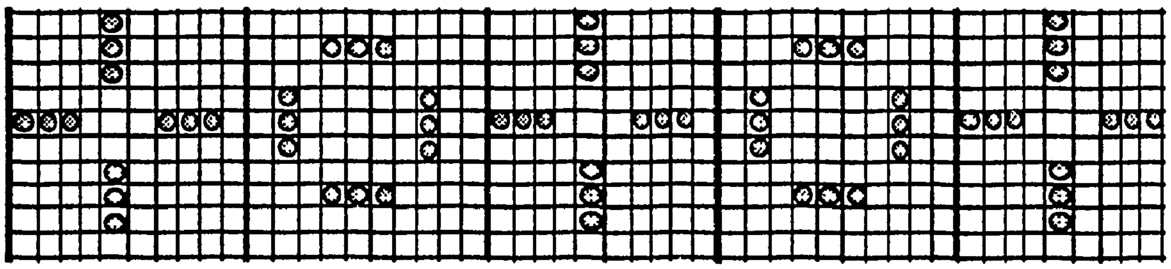


Figure 3.1c "Traffic Lights", composed of 4 "Blinkers": an oscillator of period 2.

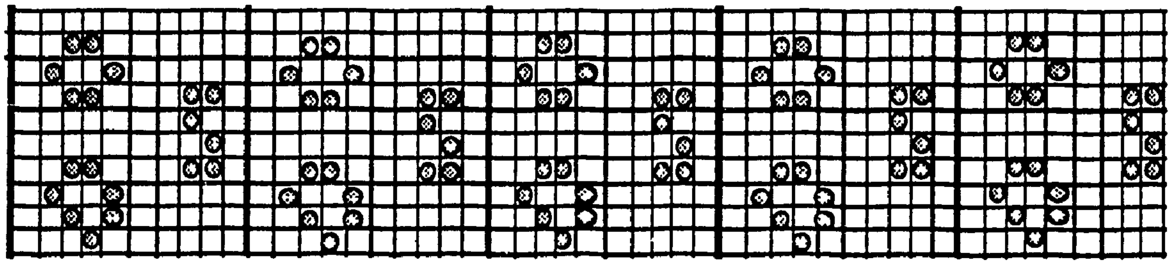


Figure 3.1d "Still Life" examples: a "Beehive", a "Snake", and a "Loaf".

global matrix so that each cell, even those on the edges, has eight neighbors. This allows Life Forms such as Spaceships, which move in one direction, to remain active for the duration of the game. Conceptually, the playing surface can be viewed as a three-dimensional donut rather than a two-dimensional plane.

These modifications to the Game of Life are illustrated in Fig. 3.2.

3.2 Ease of Verification

The execution of the test-bed application should be made easily verifiable in order to monitor the RRK during its development. An error in the RRK could possibly affect the N-Life. It will be very desirable if N-Life possesses characteristics which could make an error at its interface with RRK easily and quickly visible on the screen to the programmer. In this way, the development, debugging and installation of the RRK can be facilitated.

To illustrate how errors in the RRK could affect N-Life we first recall the behavior of the rollback algorithms described in Chapter 2. Fig. 3.3 shows a STM of three communicating processes along with a preplanned recovery-line which has been established. In this example, process P2 must store messages M2 to M5 in its SIC checkpoint, so they may be played-back during rollback. If we suppose that these messages were initially received by P2 spread over four generations, or iterations, of N-Life, then during rollback, the RRK should replay the messages in their proper sequence and during the correct iteration. Messages, containing information on neighboring cells, which are lost, duplicated, or corrupted result in aberrant Life Forms. The presence of

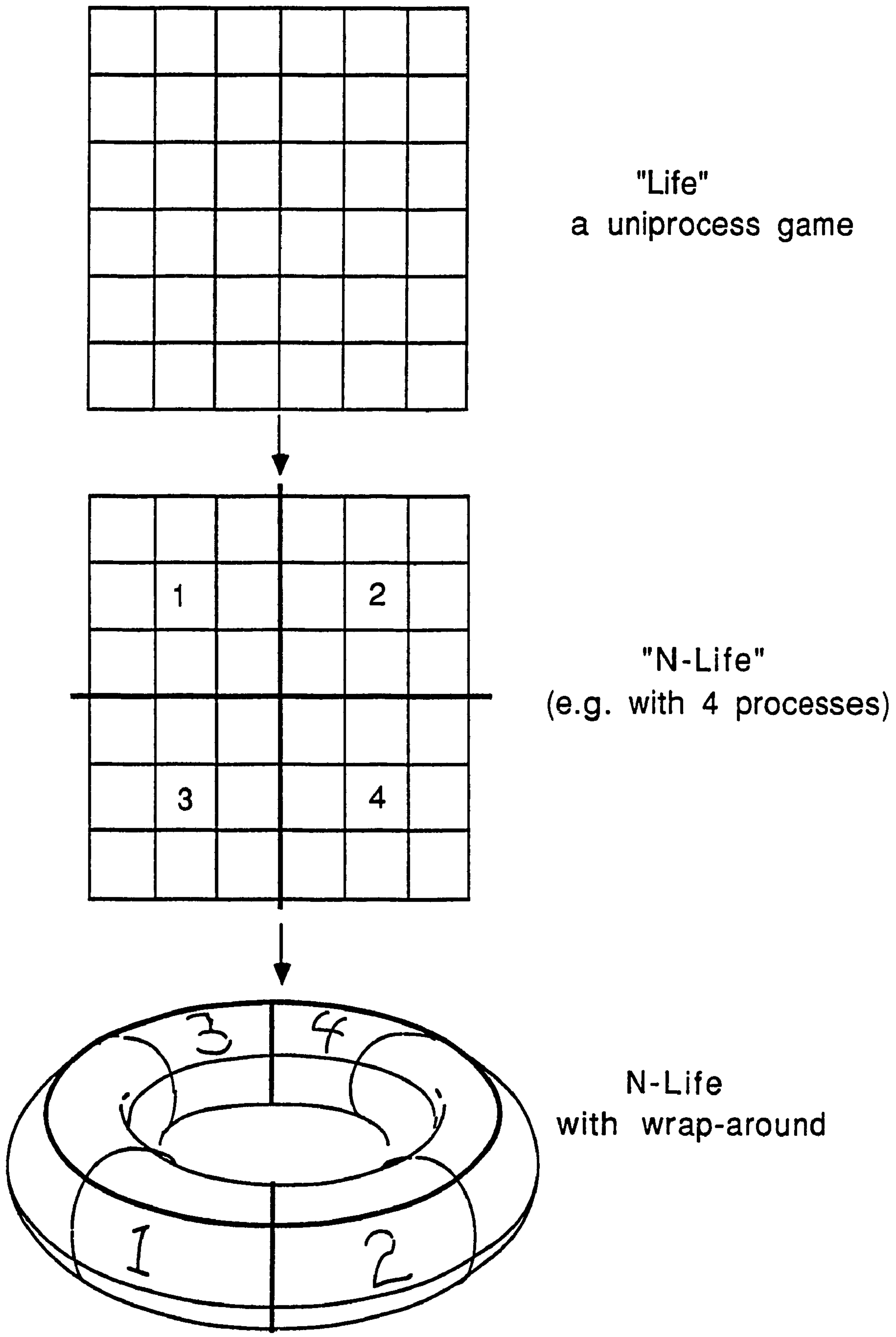


Figure 3.2 Modifications to the Game of Life

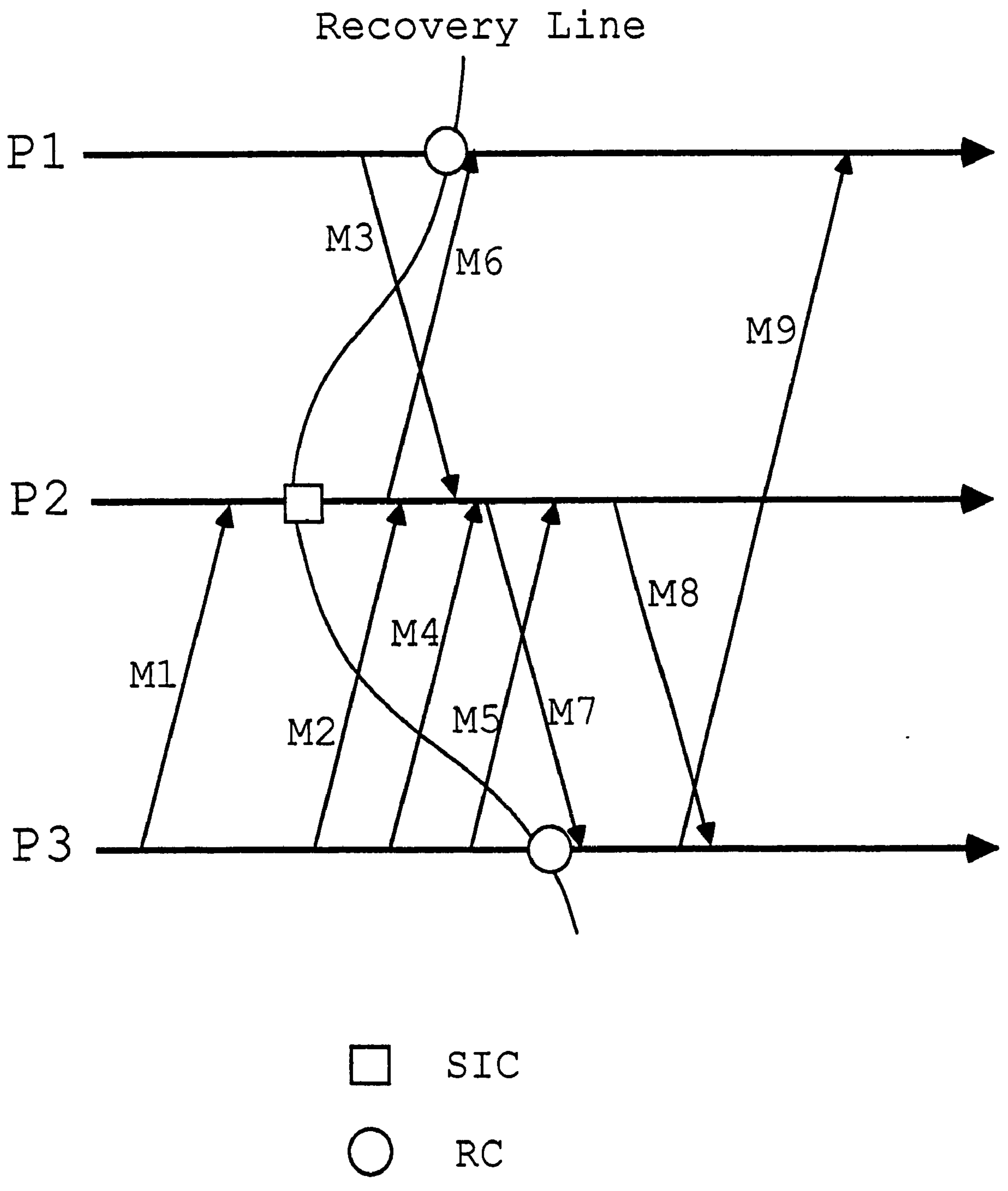


Figure 3.3 Messages which cross a recovery line

which can be immediately observed.

The potential effect of improper RRK execution on the behavior of N-Life can be seen by comparing the patterns depicted in Figs. 3.4a to 3.4c. Here, we see how the faulty introduction of a message (Fig. 3.4b) or the omission of a valid message (Fig. 3.4c) results in the destruction of a Glider. With visual feedbacks such as this, researchers familiar with the normal Life Form behaviors are able to detect and track down certain bugs in a systematic manner.

Therefore, the ease of verification of proper N-Life execution has made it possible to monitor the RRK for errors during its development.

3.3 Flexibility of Behavior

The test-bed application should allow sufficient parametric changes so as to fully exercise the algorithms employed in the RRK. In N-Life, a range of behaviors can be exhibited through the manipulation of two parameters: the "task-assignment map" and the "initial pattern file". These will now be briefly explained.

As was mentioned earlier, the global matrix can be divided into an arbitrary arrangement of subregions. Each of the processes in the system can then be allocated one or more of these regions as their computational responsibility. This assignment information can be predefined and saved in numbered task-assignment maps. By specifying a map number at run time, a particular arrangement of processes can thus be called forth. The regions allocated to different processors remain static for a given game but can be altered from run to run.

The second parameter is a file containing data on the initial pattern of Life Forms to be used during a run. The row and column

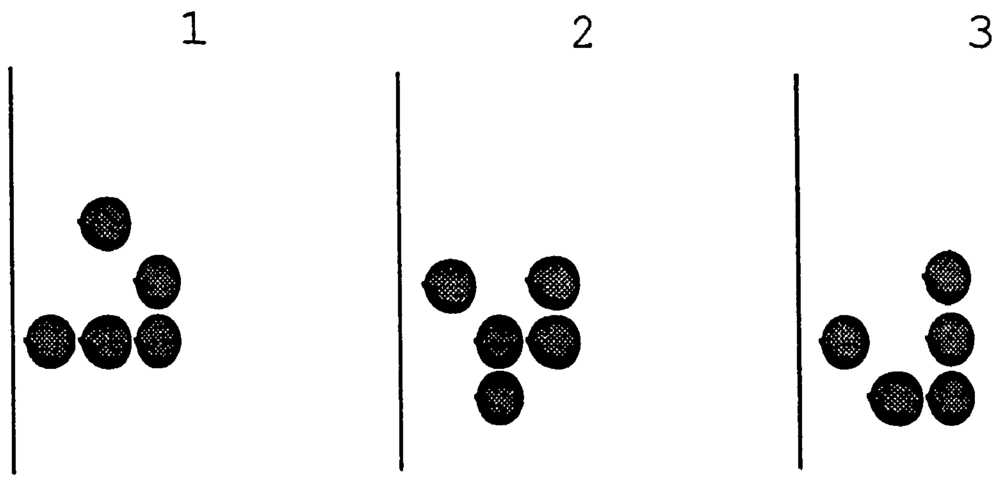


Figure 3.4a A normal Glider

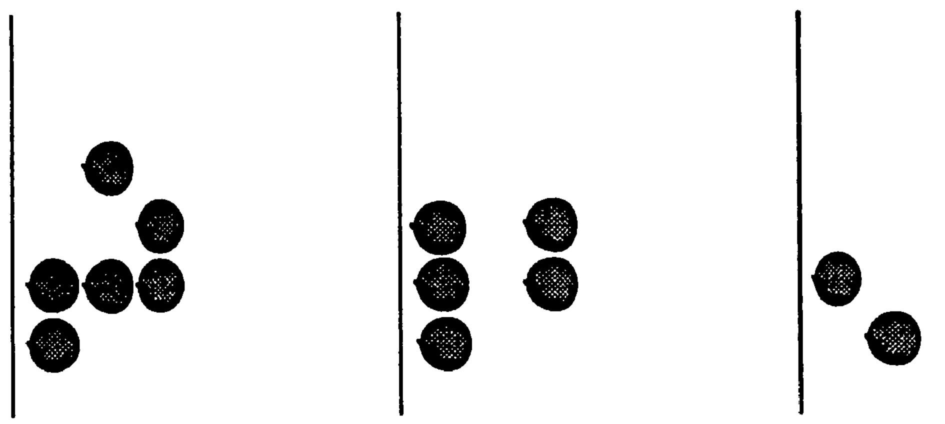


Figure 3.4b A faulty cell is added

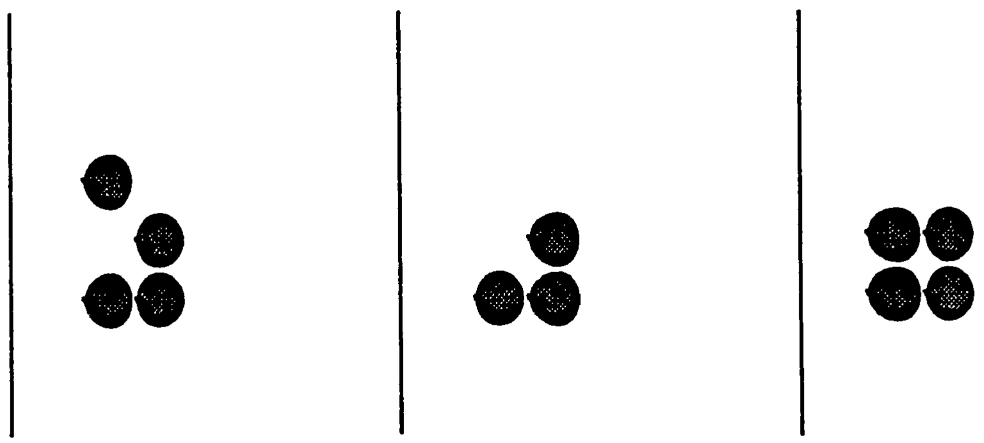


Figure 3.4c A cell is omitted

positions of the Forms' cells are saved in named files which can be specified at run time. It will be seen how these two parameters can combine to mould N-Life's behavior in the testing of the RRK.

In a distributed environment, the events occurring in one process that affect another are of particular interest. Causal relationships between send and receive events are typically observed in message based systems. During the development of the RRK, the designer should have at his disposal the tools and means, ideally speaking, to create whatever message passing behavior is required. For instance, suppose it is necessary to be able to produce an arbitrary pattern of messages that could be drawn on a space-time diagram. Consider the example shown in Fig. 3.5. The question is, can this message pattern, or an acceptable approximation to it, be generated by varying a set of parameters on the distributed application test-bed?

In our experimental work, a two-part approach has been used to achieve the desired flexibility: by varying the communication topology through different task-assignment maps, and the message frequency through different initial pattern files.

Communication Topology

Depending on the interests of the RRK designer, it may be necessary to simulate a range of network architectures. These may include anything from a thin strip layout [10], where each node has at most two communicating neighbors, to a fully-interconnected arrangement. In N-Life, task-assignment maps enable the designer to dictate which processes will share boundaries. Processes which control adjoining

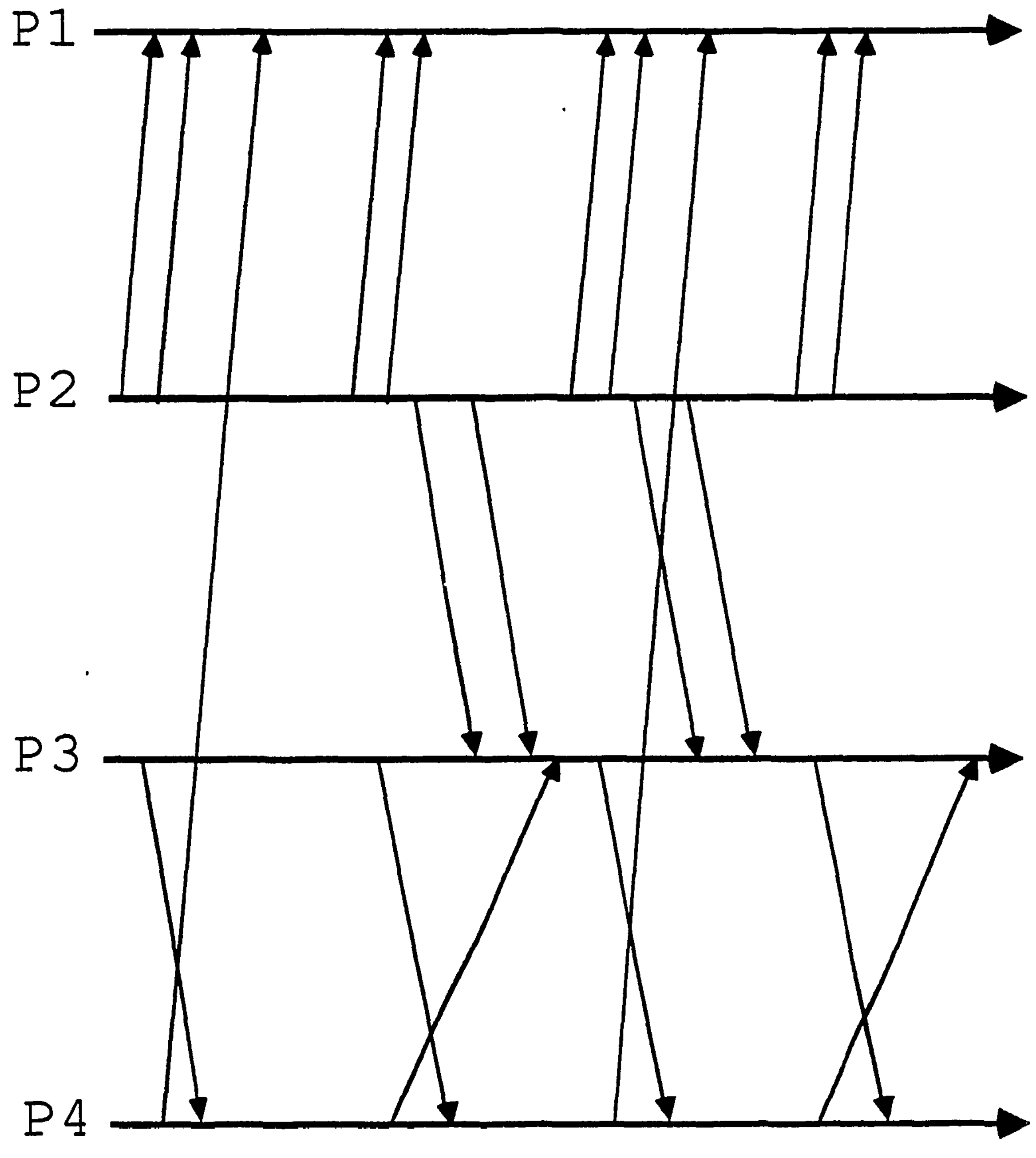


Figure 3.5 A sample message pattern

regions communicate with each other. Therefore, by specifying a particular task-assignment map, one is in effect specifying a particular channel arrangement, or communication topology, as well. Figs. 3.6a and 3.6b show how simple hierarchical and fully-interconnected topologies, respectively, can be realized.

Message Frequency

Given an appropriate communication topology, by varying the message passing frequency one can generate the message pattern of a space-time diagram.

Many iterative distributed algorithms communicate in a predictable pre-set manner which does not allow for the degree of flexibility required. N-Life was made more malleable by having a process send information for only its live cells. The receiving process, then, can deduce the complete boundary picture by what has and has not been communicated. This, for example, could result in two given processes not communicating for many iterations and then exchanging in a burst of messages.

Armed with a modest arsenal of Spaceships, Oscillators and Still Lifes and having the ability to subdivide the global matrix as one sees fit, the designer is well equiped to simulate a wide variety of message patterns using N-Life.

To illustrate how a given space-time specification may be simulated using N-Life, we refer to Fig. 3.7. In the top of the figure we see the space-time diagram shown earlier. Added to it are orthogonally drawn lines which separate the diagram into four iterative regions. Below, a series of four iterations of N-Life are displayed.

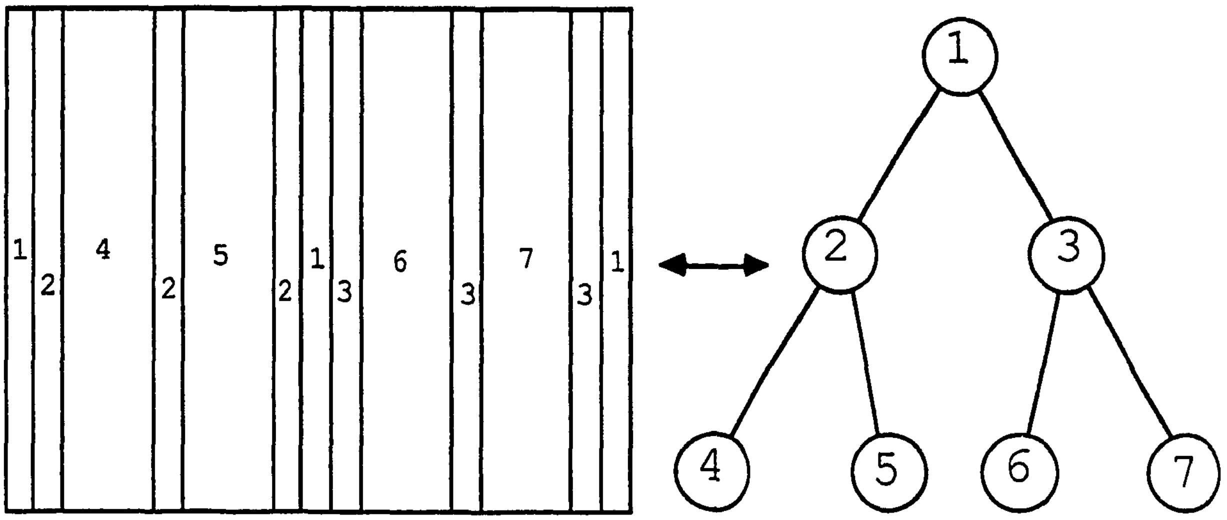


Figure 3.6a A hierarchical arrangement

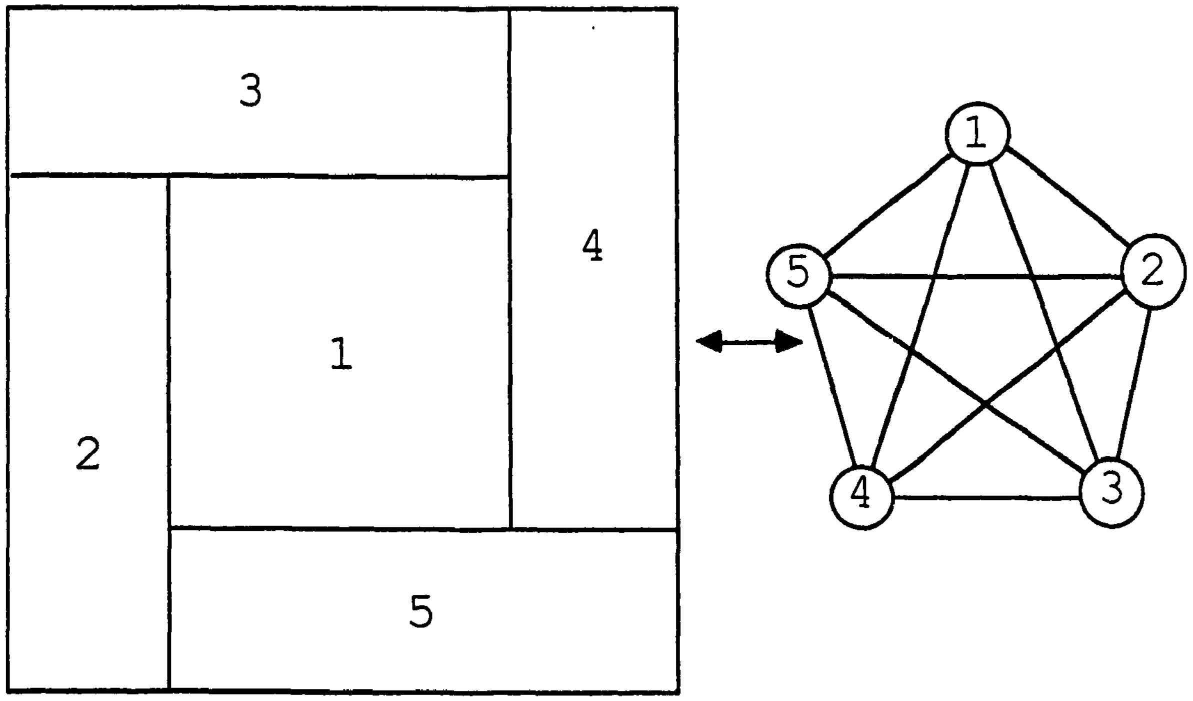


Figure 3.6b A fully-interconnected arrangement

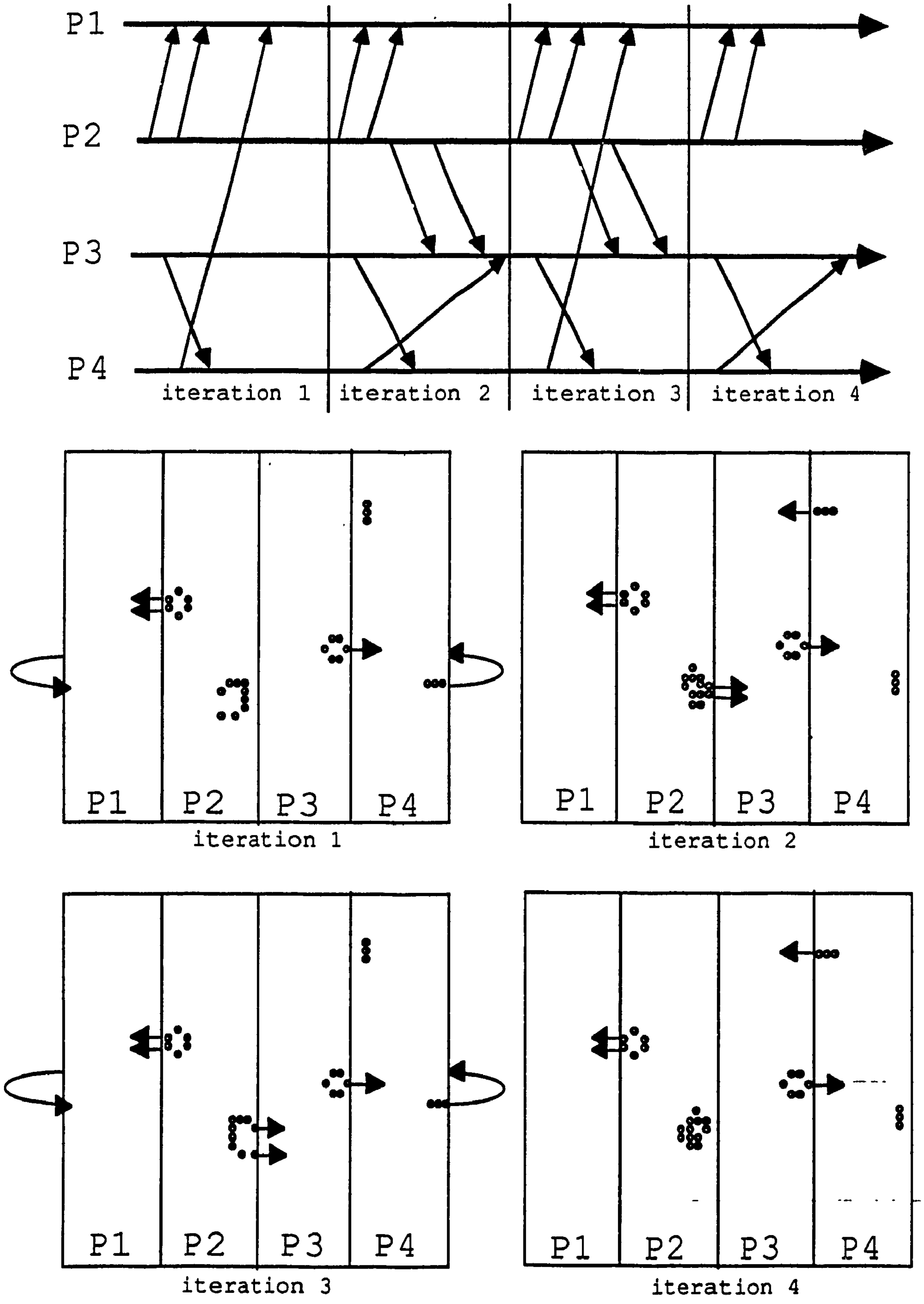


Figure 3.7 Using N-Life to simulate a message pattern

Messages passed between processes are illustrated as arrows. We see how information is only passed for the live cells which touch borders. By selecting a suitable task-assignment map and superimposing an initial pattern file upon it, the processes have been manipulated to communicate in the manner specified by the space-time diagram.

Number of Processes

The ability to vary the number of processes in the system is required for the performance analysis of the RRK. In the area of speedup analysis, for instance, this would be of particular use.

In other test environments, the ability to dynamically add and delete processes may be required. However, in our work, it was assumed that the number of processes will remain static. Although, the number could be changed from one test run to another. This was easily dictated by varying the task-assignment map parameter.

In Fig. 3.8 we see how a given global matrix, which is to be calculated, can be subdivided into first two, then four and eight-process arrangements using different task-assignment maps.

Problem Size

In the performance analysis of the RRK implementations, varying the problem size, or CPU workload, for the individual processes may also be desired. In N-Life, by adjusting the size of the matrix the workload is proportionally altered. So, to change the problem size one simply has to redefine the matrix dimensions. Once again, by specifying a suitable task-assignment map this can be easily done.

As was the case in varying the number of processes, the ability to

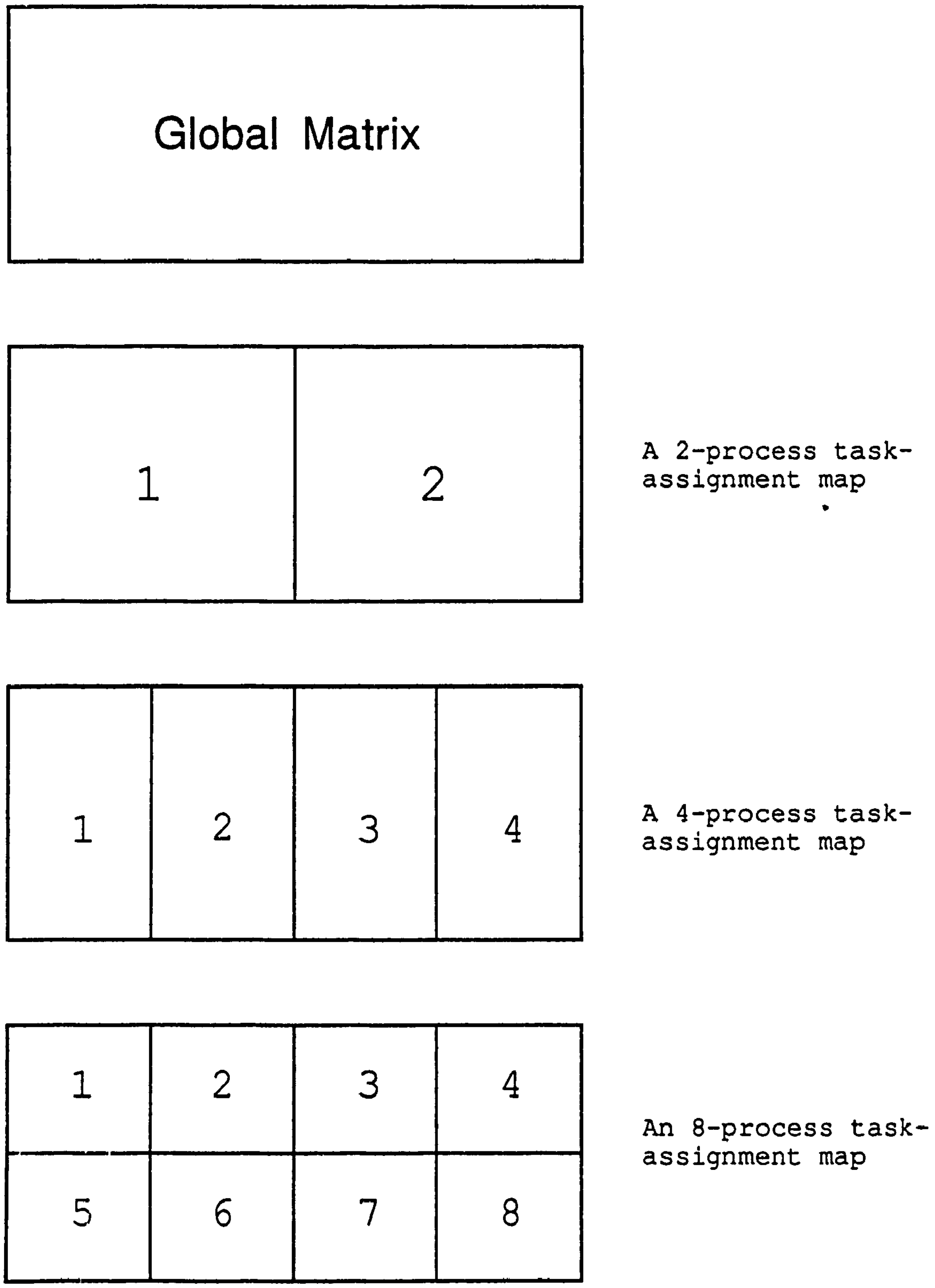

Global Matrix
1
2
A 2-process task-assignment map
1
2
3
4
A 4-process task-assignment map
1
2
3
4
5
6
7
8
An 8-process task-assignment map

Figure 3.8 Varying the number of processes via task-assignment maps

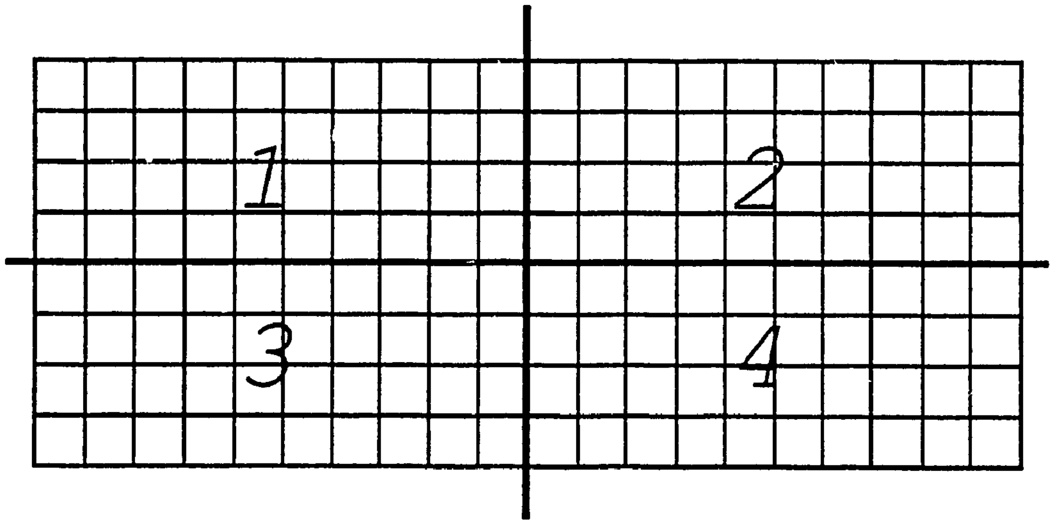
dynamically change the problem size may be required in certain environments. But it is not required in our case. The question of whether N-Life could be altered in order to offer dynamic task-assignments has not yet been addressed.

As we have seen, the problem size, or CPU workload, is determined by the matrix size, and because a process communicates information pertaining only to its borders, the message passing overhead is roughly proportional to the perimeter of the region it has been assigned.

Observe that the global matrix is decomposed into regular regions in Fig. 3.9a. The CPU workload and the communication overhead for the individual processes can be summarized in t[illegible] of "cell units", as shown at the bottom of the figure. In Fig. [illegible] [illegible]e s[illegible]e that by varying the relative sizes of these regions, one could [illegible] a particular process more message-bound or less message-bound, in relation to the computation done by that process.

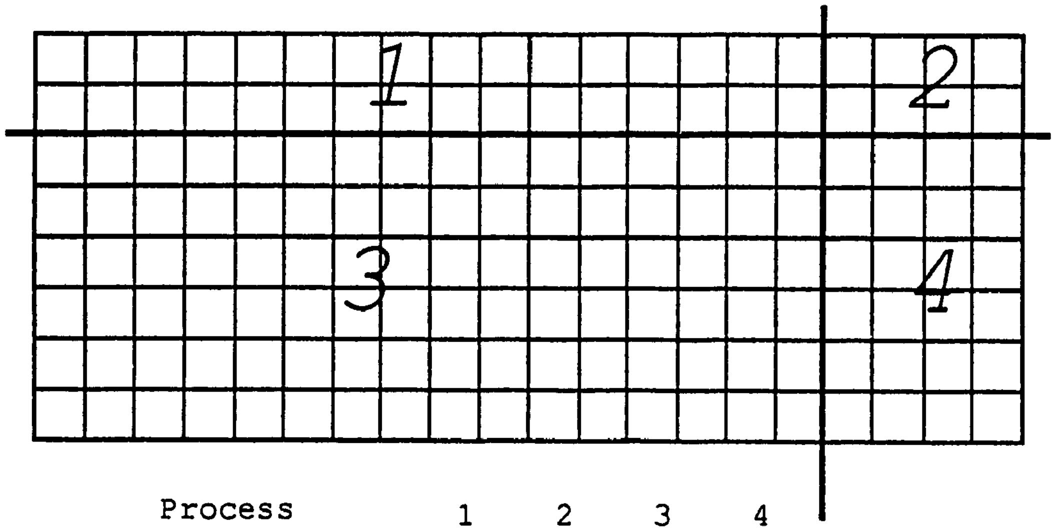
3.4 Other Considerations

Every distributed algorithm seems to have certain inherent synchronization and communication requirements. Some require "fully-synchronous" blocking sends and receives, as available in ADA's "Rendezvous" [13]. Others need only the "half-synchronous" constructs. That is, blocking sends with non-blocking receives or, more often, non-blocking sends with blocking receives. Truly asynchronous algorithms require no explicit synchronization and can be implemented using strictly non-blocking send and receive primitives. Finally, there are algorithms which employ both the blocking and non-blocking receives during normal execution. These algorithms, termed "loosely-



Process	1	2	3	4
Perimeter(P)	24	24	24	24
Area (A)	40	40	40	40
Ratio (P/A)	0.6	0.6	0.6	0.6

Figure 3.9a Cell unit data for regular regions



Process	1	2	3	4
Perimeter(P)	32	8	40	16
Area (A)	32	8	96	24
Ratio (P/A)	1.0	1.0	0.41	0.66

Figure 3.9b Cell unit data for variably sized regions

synchronous", only block for messages when their local requirements so dictate.

These types of properties are likely to be invariant for a given application program. In this context we can ask the following questions when selecting a suitable test-bed application:

Q1. Is the application algorithm basically synchronous, asynchronous, or loosely-synchronous?

Q2. What assumptions are made by the RRK in regard to the synchronization properties of the application?

Q3. How rigidly must the application comply with these assumptions and what are the consequences of an inexact match?

The dynamic checkpointing algorithms, implemented in this work, view the distributed application as a collection of asynchronously communicating processes. Although processes which are synchronous could use the RRK, it is in the fully-asynchronous environment that these types of rollback algorithms are of the most use.

N-Life is loosely-synchronous. While processes compute the next generation of their respective regions, messages containing information on bordering cells are received, and the data stored, asynchronously. Before a process can complete one iteration and proceed with the next, all of the incoming messages for the present iteration must have been received (we saw earlier how messages received out of turn could result in aberrant Life Forms). If the full complement of messages have not been received the process will block, thus exhibiting the loosely-synchronous behavior.

Since the RRK we implemented accepts applications with varying degrees of synchronization, N-Life served as a test-bed application. However, since the RRK should perform best in a fully-asynchronous environment, the loosely-synchronous N-Life is not the best choice, from a synchronization point of view.

In this chapter we have presented some of the desirable characteristics of a suitable test-bed application to be used in the development and testing of the RRK. N-Life, a distributed test-bed application, which was designed using J.H. Conway's Game of Life as a base algorithm, was used throughout as an illustrative example. Algorithm simplicity should be considered when selecting a test-bed application; the simpler the algorithm the quicker its implementation. Ease of verification of the test-bed application was determined to be a very desirable trait when tracking-down bugs in the RRK. When one wishes to thoroughly test the RRK a selection of variable parameters should be made available in the application. In N-Life, two parameters, the task-assignment map and the initial pattern file, were used in order to give the application a flexibility of behavior in such things as communication topology, message frequency, number of processes and problem size. Finally, it was concluded that the relative synchronization properties, assumed by the RRK and exhibited by the prospective test-bed application, respectively, should be compared for their compatibility.

Chapter 4

Comparison of Two Rollback and Recovery Algorithms

Very little is known about the practical performance of distributed rollback and recovery algorithms. One of the aims of this thesis is to gain some understanding of the experimental performance for selected rollback and recovery algorithms. We also wanted to explore the relative strengths and weaknesses of two specific algorithms, the RLV [23] and BCS [5] algorithms, through experimental implementations and evaluations.

In Chapter 2, we discussed the general characteristics of the type of rollback and recovery algorithms which we are concerned with, the dynamic checkpointing algorithms. In this chapter, we will first look more closely at the two algorithms we have chosen to compare. Then we will briefly show how these algorithms were integrated into the RRK design. Next, we will outline the measurements taken, along with the special measurement software designed for this purpose. Then, the measurement results will be presented and interpreted. Finally, we will conclude the chapter with some remarks and conclusions on what we have learnt.

4.1 Two Rollback and Recovery Algorithms

The algorithms are described to a level of detail suitable for the purposes of this thesis. For detailed discussions beyond this level see [5] and [23]. Although both algorithms allow for simultaneous rollback (the occurrence of two or more separately initiated rollbacks during the same time frame), the specific actions to be taken in the case of the BCS algorithm are not presented in full by its authors. We have, therefore, decided to omit simultaneous rollbacks in our study.

In fact, upon further examination, we found the BCS algorithm to be deficient in its treatment of the rollback procedure. In order to implement and to make any performance measurements of rollback and recovery, we therefore had to make the appropriate augmentations to the BCS algorithm. In this context, we pondered the merits of any measurements which would be obtained, since the changes we would make may not be exactly what Briatico et. al. [5] would have wanted.

It was decided that, since our intent was not to make judgements about the absolute superiority of one algorithm over another but rather to gain some valuable insight on the relative performances of algorithms of their sorts in different environments, we would proceed with making the alterations to the BCS algorithm, in as fair a means as possible.

What follows, then, is a brief summary of the two algorithms, followed by a description of the changes made by us to the BCS algorithm.

4.1.1 RLV Algorithm

Checkpointing

The application processes create SIC checkpoints at appropriate points in their execution. In the RLV algorithm, each SIC created in the system is uniquely labelled by a two-number identification tag <p,q>, where p indicates the ID of the process which created the SIC, and q indicates the ordinal number of the SIC, within that process. For example, <2,7> corresponds to the seventh SIC created by process number 2.

The creation of a SIC may, through the passing of application messages, cause the subsequent creation of a RC in one or more of the other processes. For each process P, the RRK maintains a local checkpoint vector or "CCP" composed of n counters, one counter for each of the n processes in the distributed system. The counters indicate the SIC counts of the individual processes, as perceived by P. One may note that, at a given time, the only counter which is guaranteed to be up to date is the one corresponding to P itself. The processes inform one another about newly created SICs by appending their CCP to each application message sent out. When a process receives a message, it strips off the received CCP vector or "RCP". It then compares the counters of the RCP with the corresponding counters of its own CCP. If RCP(i) > CCP(i) then the following actions are carried out: CCP(i) is set to RCP(i); a RC is created and the RC is assigned the two-number tag of the SIC which caused its creation (i.e., <i,RCP(i)>). This RC is now regarded as being owned by process i. In this way, a SIC and the set of RCs which it caused to be created on the different processes form a valid recovery line. This recovery line is identified by the same two-number tag used by its checkpoints, and is owned by the process which created the SIC (e.g., process i).

However, if RCP(i) < CCP(i) then this message intersects one or more recovery lines and is stored in all local checkpoints owned by process i with an ordinal number greater than RCP(i). If the process is later rolled back to any of these checkpoints, this message will be played back.

Fig. 4.1 shows three communicating processes and the creation of three recovery lines. One sees how a recovery line, and the RCs which it contains, inherit the two-number tag of the instigating SIC.

Rollback and Recovery

Upon detection of an error, a process P initiates the rollback and recovery procedure by rolling back to its most recent SIC, <p,q>, which precedes the occurrence of the fault, as determined by an "error handler module" (see Section 4.3). It then loads the saved application process state variables, deletes all checkpoints created after <p,q>, inserts any saved messages to the head of their respective ICBs, and then sends out a rollback control message, R<p,q>, on all its outgoing channels. The application program is then allowed to resume execution with its new error-free state.

When another process receives its first R<p,q> message, it searches, beginning with the first checkpoint created, for the first checkpoint with an ID of <p,r>, where r is greater than or equal to q. If such a checkpoint exists, then this process is involved in the rollback. The application process state is loaded, <p,r> and all the

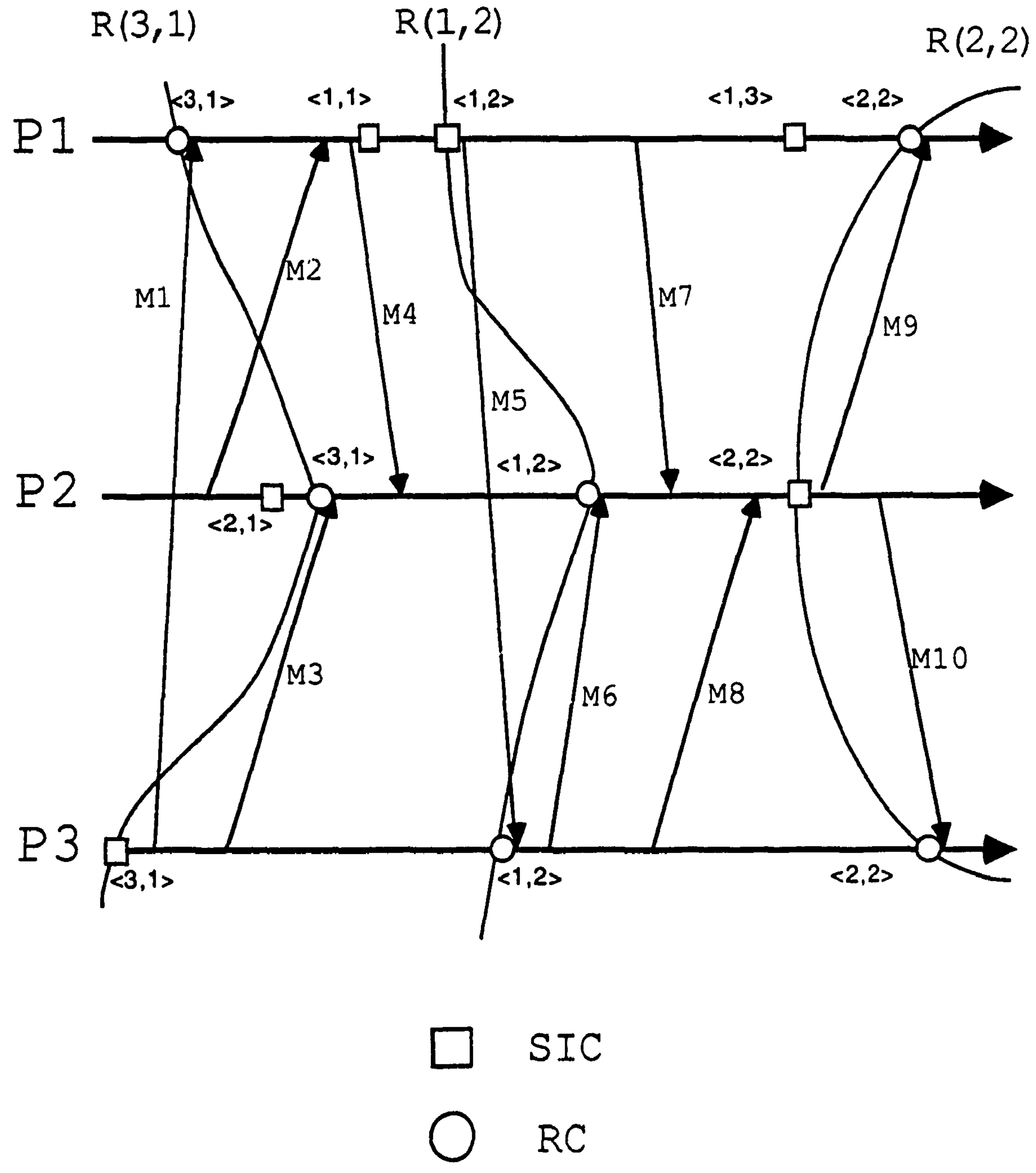
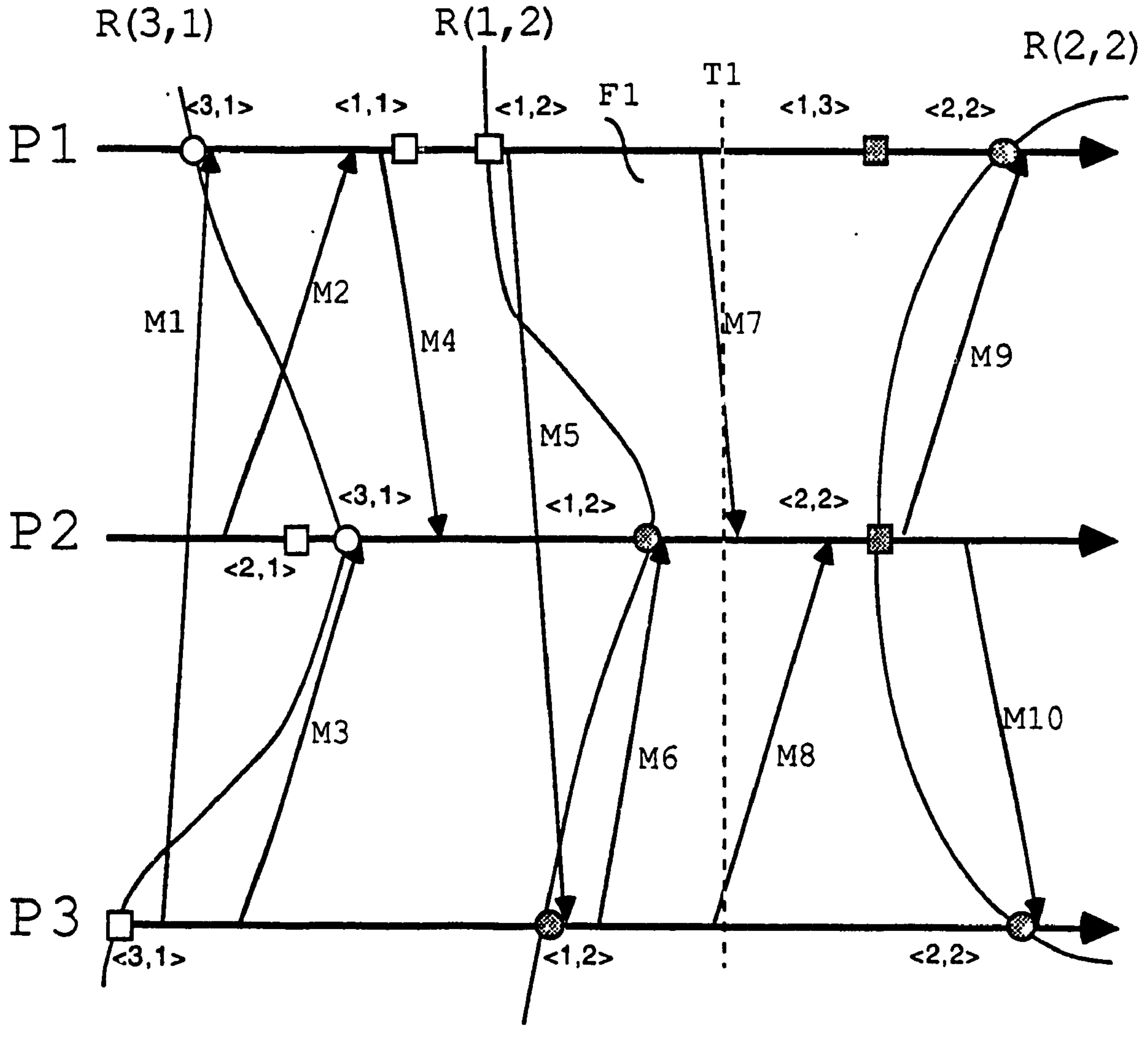


Figure 4.1 RLV checkpointing

checkpoints following it are deleted (since <p,r> is a RC and will be recreated after rollback, it must also be deleted), messages are inserted to the head of their ICBs, and the rollback control message R<p,q> is sent out by this process on all its outgoing channels. Control is then returned to the application program.

Referring to Fig. 4.2, we see how a fault F1, detected at time T1, has caused process P1 to roll back to the SIC <1,2>. Processes P2 and P3 have also rolled back, to their corresponding <1,2> RCs. All the processes have deleted the appropriate checkpoints (symbolized as shaded checkpoints), carried out the other rollback actions described above, and have resumed normal execution. One sees that two application messages sent after the recovery line was created, M7 and M8, had not yet been received at the time the fault was detected. These messages, called pre rollback messages (since they were sent before the rollback began), must be purged since they represent the result of computations which must be redone. To distinguish between pre and post rollback messages, each process sets all its input channels, except for the one (if any) on which it was informed of rollback, to the "cautious" state. While in the cautious state, a channel examines the RCPs on incoming messages, if RCP(p) is greater than or equal to q (i.e. for R<p,q>) then this message is discarded. Otherwise, it is a post rollback message and is processed in the manner described in the checkpointing part of the algorithm. Once a rollback message is received on a channel in the cautious state, the channel is reset to the "normal" state, since all pre rollback messages must now have been flushed through.



□ SIC

○ RC

Figure 4.2 RLV rollback

4.1.2 BCS Algorithm

Checkpointing

As was the case with RLV, processes create SICs at appropriate points in their execution and the creation of a SIC may result in the creation of RCs by other processes. However, in the BCS algorithm, all the checkpoints that a process creates, be they SICs or RCs, are assigned ordinal numbers from a monotonically increasing counter, called the "K" counter. An RC does not, therefore, inherit the ID of the SIC which caused its creation, as was the case in the RLV algorithm. In this scheme, a recovery line q is made up of the checkpoints in the system which have the same q ID, be they SICs or RCs.

Each checkpoint of a process contains a list of process IDs called the "PEset". The PEset represents the set of processes to be rolled back, as far as that process knows, if the checkpoint in question is used in rollback. The PEset is implemented as an ordered list of process IDs.

Every process appends the number of its latest checkpoint, indicated by K, to every application message it sends out. When a process receives a message, it strips of the received checkpoint counter R and compares it to its own K tally. If R > K then the process creates (R-K) RC checkpoints, in order to bring its checkpoint count up to that of the sender. In our implementation, the first of these (R-K) checkpoints contains the process state. The remainder of these checkpoints simply point to this stored state, and hence are referred to as "dummy" or "hollow" checkpoints.

If, however, K > R, then this message is stored in all previously taken checkpoints with ID numbers of i, where i > R. In addition, for every message received, all the checkpoints with ID numbers less than or equal to R, append the process ID of the sender to their PEset.

Fig. 4.3 shows the same application process behavior, in terms of SIC creations and messages passed, that was depicted in Fig. 4.1. Here, we see how the BCS algorithm has caused a different set of RCs and recovery lines (RLn) from those created by the RLV algorithm.

Rollback and Recovery

The rollback and recovery procedure is described as a two-phase operation. In the first phase, the process which initiates the rollback, from here on referred to as the "rollback initiator", selects a recovery line, and then constructs a rollback "Offer" message. This Offer includes the number of the recovery line, and the LPIset, which is a list of the processes invited to rollback. The LPIset initially contains the PEset of the rollback initiator's "rollback checkpoint", that is, the checkpoint to which it is rolled back. The rollback initiator then sends this Offer message to the first process, listed in the LPIset, and then enters a "Wait" state.

The process which receives this Offer, examines its corresponding rollback checkpoint. It then updates the Offer's LPIset by adding any processes, contained in the PEset of its own rollback checkpoint, which were not previously included in the Offer's LPIset, and then propagates the Offer to the next process listed in the updated L*PIset. The process then enters a Wait state. This procedure is repeated until the Offer message reaches the last process listed in the LPIset.

The last process in the LPIset changes the Offer to an "Accept" taken checkpoints with ID numbers of i, where i > R. In addition, for every message received, all the checkpoints with ID numbers less than or equal to R, append the process ID of the sender to their PEset.

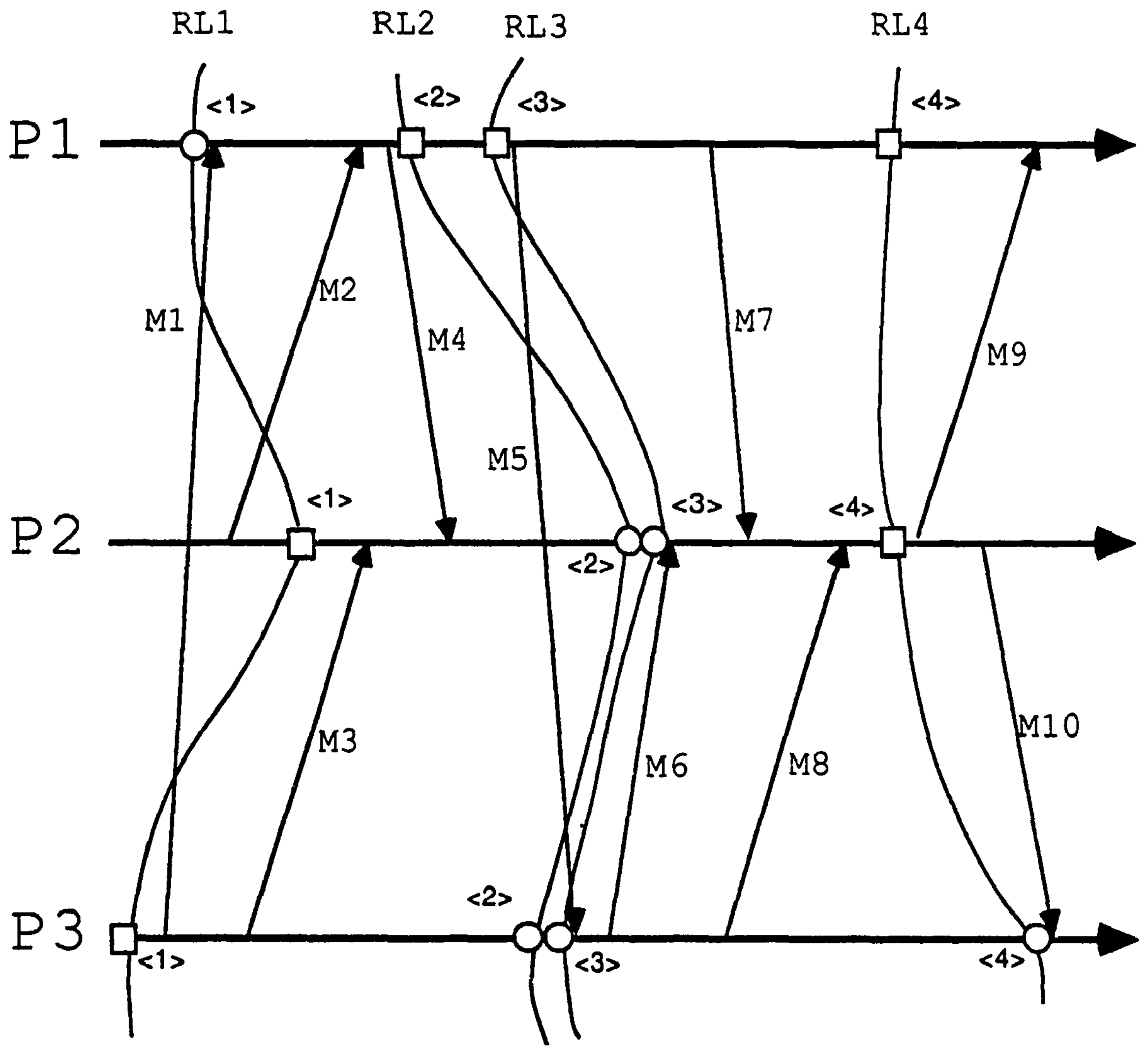
Fig. 4.3 shows the same application process behavior, in terms of SIC creations and messages passed, that was depicted in Fig. 4.1. Here, we see how the BCS algorithm has caused a different set of RCs and recovery lines (RLn) from those created by the RLV algorithm.

Rollback and Recovery

The rollback and recovery procedure is described as a two-phase operation. In the first phase, the process which initiates the rollback, from here on referred to as the "rollback initiator", selects a recovery line, and then constructs a rollback "Offer" message. This Offer includes the number of the recovery line, and the LPIset, which is a list of the processes invited to rollback. The LPIset initially contains the PEset of the rollback initiator's "rollback checkpoint", that is, the checkpoint to which it is rolled back. The rollback initiator then sends this Offer message to the first process, listed in the LPIset, and then enters a "Wait" state.

The process which receives this Offer, examines its corresponding rollback checkpoint. It then updates the Offer's LPIset by adding any processes, contained in the PEset of its own rollback checkpoint, which were not previously included in the Offer's LPIset, and then propagates the Offer to the next process listed in the updated L*PIset. The process then enters a Wait state. This procedure is repeated until the Offer message reaches the last process listed in the LPIset.

The last process in the LPIset changes the Offer to an "Accept"



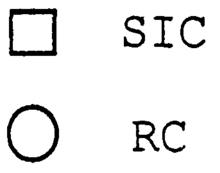


Figure 4.3 BCS checkpointing

message, and sends it to its predecessor in the LPIset, and enters a second Wait state. The Accept message finds its way back to the rollback intiator by reversing the path taken by the Offer message. Once the rollback initiator receives the Accept message, the second phase of the rollback procedure begins.

In this second phase a "compensation action" is carried out on the input buffers of all the processes involved in the rollback. This action is necessary to purge the pre rollback messages which may be in transit. The phase begins with the resending of the Offer message along the LPIset path. Each process receiving the Offer will compensate its input buffers by waiting for messages in transit, and then inserting the messages stored in the checkpoints to the head of the ICBs. Neither the length of time required to wait, nor the specific actions used for purging messages is specified by the authors of BCS.

The need to have a second Offer message start the compensation action results from the authors' assumption of the possible occurrence of simultaneous rollbacks. By the end of the first phase, only the rollback initiator would be aware of the global acceptance of the Offer. The rollback initiator must, therefore, inform the other processes that the offered recovery line is now to be used, at which point they can begin their compensation actions. Since simultaneous rollbacks are not studied in this thesis, the processes should be allowed to start their compensation actions upon completion of their duties in the first phase.

4.1.3 Augmentations to BCS Algorithm

There are two areas where the BCS algorithm is incomplete. The first involves the potential for incomplete PEsets and, therefore, the possibility for the existence of a faulty process not participating in the rollback operation. The second incompleteness lies in the input buffer compensation action.

To illustrate the first point, we refer to Fig. 4.4. The figure shows four communicating processes and the creation of a valid recovery line RL1. A fault F1 has been detected at time T1 by process P2, and it must now access the PEset contained in its rollback checkpoint to begin the propagation of the rollback Offer message. We recall that, during the checkpointing, a process updates its PEsets upon the reception of application messages. Accordingly, P2 has included process P3 in its PEset due to the checkpointing information it received through message M3. Similarly, P3 has included process P4 due to message M5. P2, therefore, sends the Offer to P3, which, in turn, sends the Offer to P4. However, none of these processes have been made aware of the need to include process P1 in the rollback. Ofcourse, P1 itself knows it should be included in a rollback to RL1, unfortunately it is unaware that this rollback is in progress. The problem originates in having the PEset updates based only on receive events. We see that if P2 had included P1 in its PEset on the basis of its sending message M1, that a complete rollback, including P1, would be possible. Accordingly, we have altered the BCS algorithm so that, with every application message sent, the sender adds the ID of the receiver to its PEsets (if it has not been previously included).

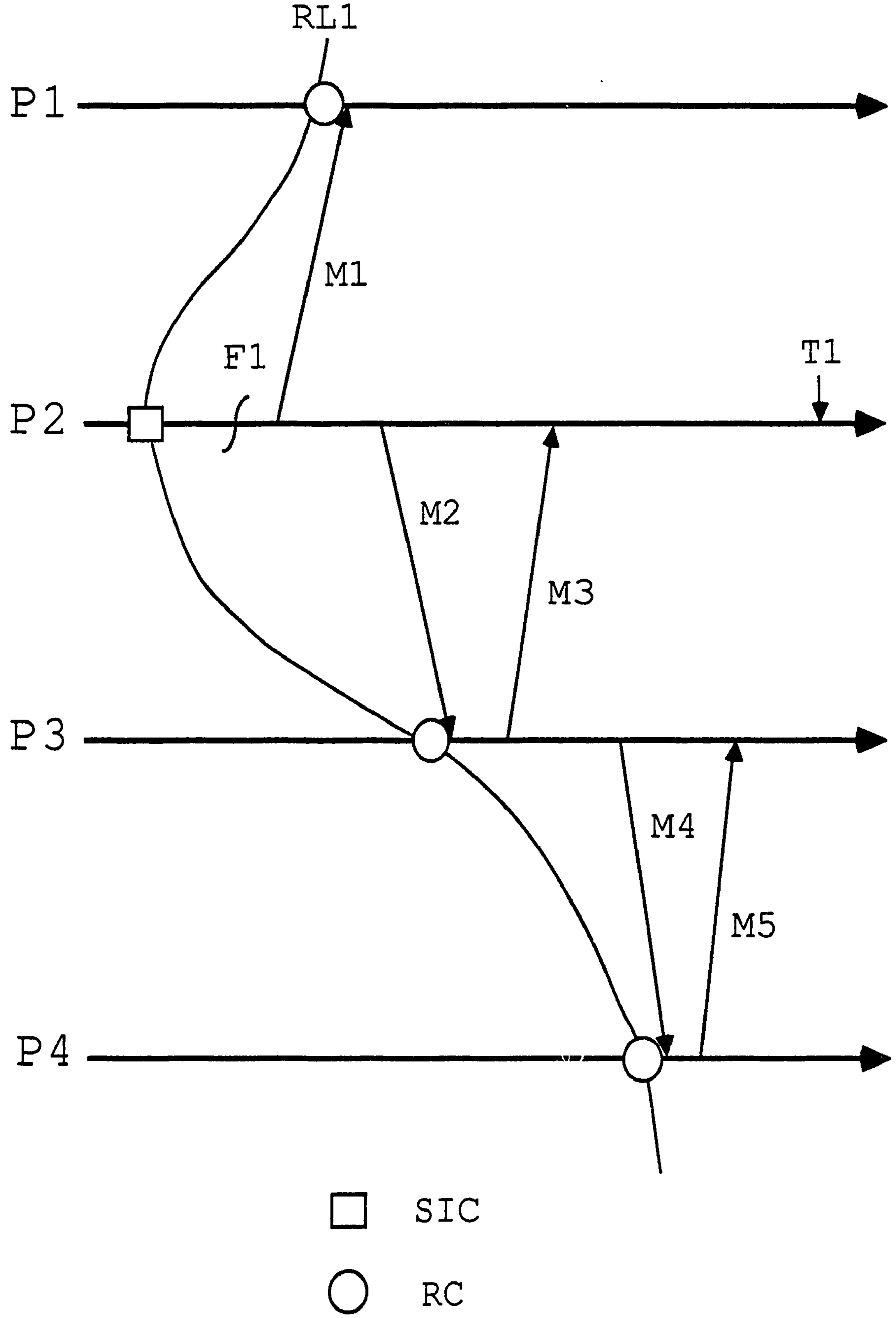


Figure 4.4 Incomplete PEsets

The second area where BCS exhibits a weakness is in the input buffer compensation action. This is illustrated in Fig. 4.5a. The figure shows three processes which have already gone through the first phase of the rollback procedure and are now carrying out the compensation actions. The process P1 is the rollback initiator. We see that each process, beginning with P1, waits for a specified period of time d, considered long enough to ensure the reception of any in-transit messages (e.g., M1 and M2). In the figure, P1 has finished its waiting and sent a new application message M3 to P3, which is received before P3 has finished its own wait period. This message is incorrectly identified as a pre rollback message by P3, and is purged. We see that, the weakness in the compensation action lies in its inability to select a proper value for d. If too short a waiting period is specified then the pre rollback messages may be allowed to slip through. If, on the other hand, a waiting period long enough to catch all in-transit messages is specified then valid post rollback messages may be purged.

In our implementation, a three-part solution was used to solve the above problem. First, during the passing of the first Offer message, the extent of pre rollback checkpointing is established. To do this, the rollback initiator includes its pre rollback K counter with the Offer message. Each process, in turn, compares its own K counter with that in the Offer and leaves the greater of the two in the Offer message. By the end of the first phase, the largest pre rollback K counter, say n, has been established, and is known to all the processes involved in the rollback to recovery line r.

Next, each process, after having deleted all checkpoints created after r, creates a series of (n-r+1) "hollow" checkpoints, starting at

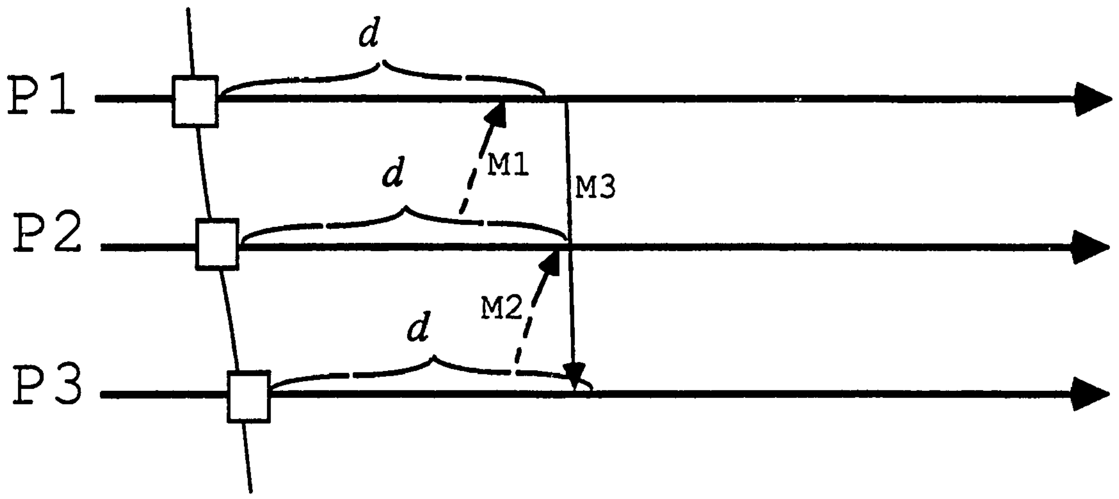


Figure 4.5a Faulty compensation action

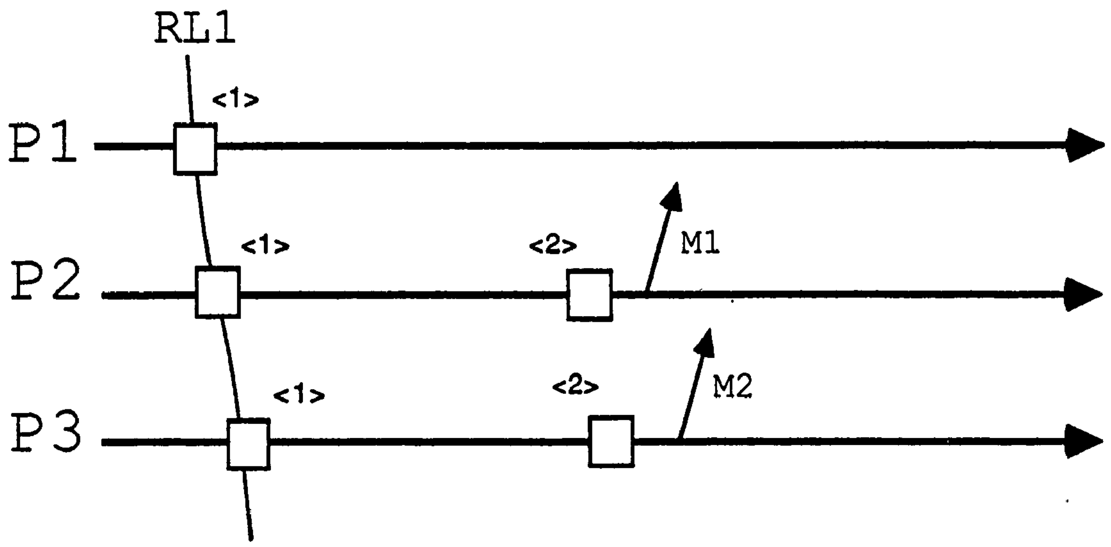


Figure 4.5b Pre rollback messages (K counters = 2)

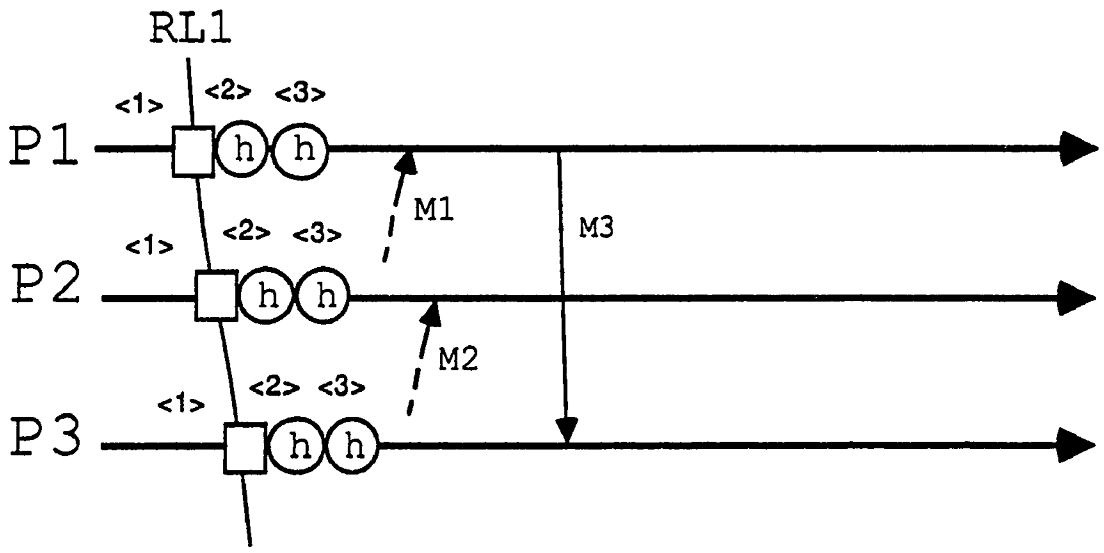


Figure 4.5c Post rollback message, M3 (K counter = 3)

number r+1. These hollow checkpoints are checkpoints with no saved process state of their own, but contain pointers to the checkpoint of the recovery line r. Additionally, each input channel of the rolled back process is set to a "cautious" state. After setting all its input channels, a process can return control to the application program. It does not have to wait for the duration `d`, as shown in Fig. 4.5a.

The third part of the solution is carried out during the execution of the application program. While a channel is in the cautious state, each incoming message is examined. The appended checkpoint counter c is compared with the last hollow checkpoint number h and the recovery line r. If c is greater than or equal to r and c < h, the message is a pre rollback message, and is purged. When the channel receives a message with a checkpoint number c, where c is greater than or equal to h, the channel is set back to the "normal" state.

To illustrate what has been described above, we refer to Fig. 4.5b and Fig. 4.5c. In Fig. 4.5b, two messages, M1 and M2, are sent prior to rollback and carry checkpoint counters of 2. The extent of pre rollback checkpointing is established as being 2 and, as is shown in Fig. 4.5c, the processes make hollow checkpoints up to and including number 3. Accordingly, the post rollback message, M3, carries a checkpoint counter of 3. We see how pre rollback messages, M1 and M2, can now be distinguished from post rollback messages, such as M3, on the basis of their appended checkpoint counters.

These modifications to BCS, make its implementation possible without degrading its performance. In fact, as we will see in Section 4.4, the removal of the second Offer message from the rollback procedure has probably resulted in an improved performance, since serialized rollback control messages, such as the Offer message, cause a definite overhead in the distributed computation.

4.2 Algorithm Specific Software

In this section, we briefly outline the AS software used for the RLV and BCS algorithms. Fig. 4.6a shows the Checkpointing Module's AS software, in the case of the RLV algorithm. Among the functions included in this software, we see the Create_chpt function, which carries out the actual creation of a checkpoint and calls Save_ps to save the application process state. The functions Save_msg, Insert_msg, and Create_msg_slot are responsible for the storing of messages which cross existing recovery lines, into the checkpoints of those recovery lines. When this AS module is compared to the corresponding software of the BCS algorichm, shown in Fig. 4.7a, we see that the BCS version is more compact, due to its simpler algorithm logic. We also observe that, in the case of BCS, the Kwrite function interfaces with the AS software through calls to Update_PEs. The latter is a direct result of the augmentations to BCS, described in the previous section.

As far as the Rollback and Recovery Modules of the AS software are concerned, we see, by looking at Fig. 4.6b and Fig. 4.7b, that the overall structure is quite similar for both RLV and BCS. Once again, due to the fact that the algorithm logic is more complex, we see that RLV requires the use of more functions. However, it should be recalled that RLV fully supports the occurrence of simultaneous rollbacks, and the mechanisms for this are incorporated into this design through the Scan_ARM and Append_ARM functions. We see that many functions in RLV have their namesakes in BCS. These functions carry out the common

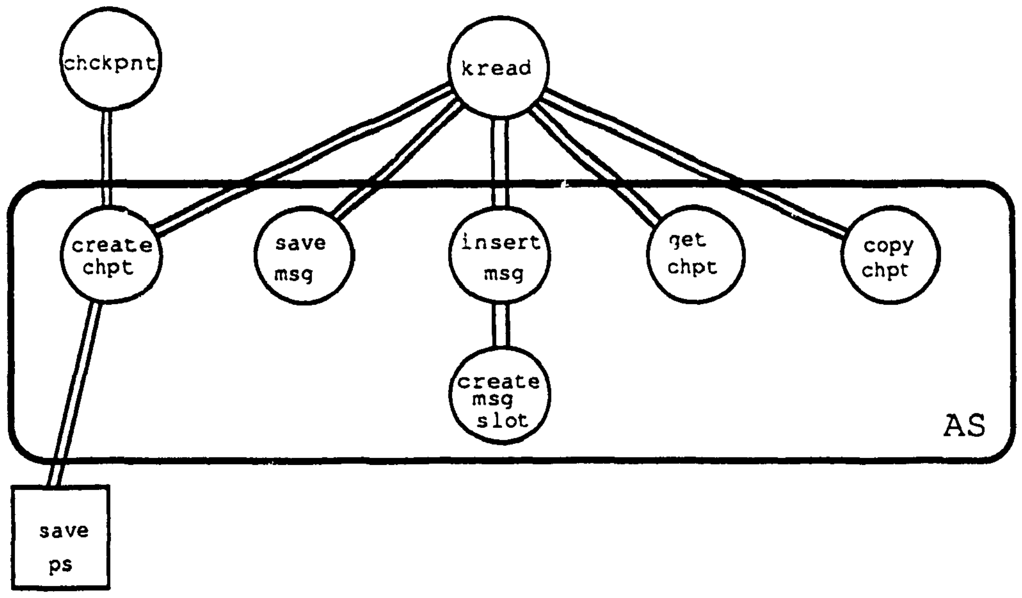


Figure 4.6a AS software in RLV checkpointing module

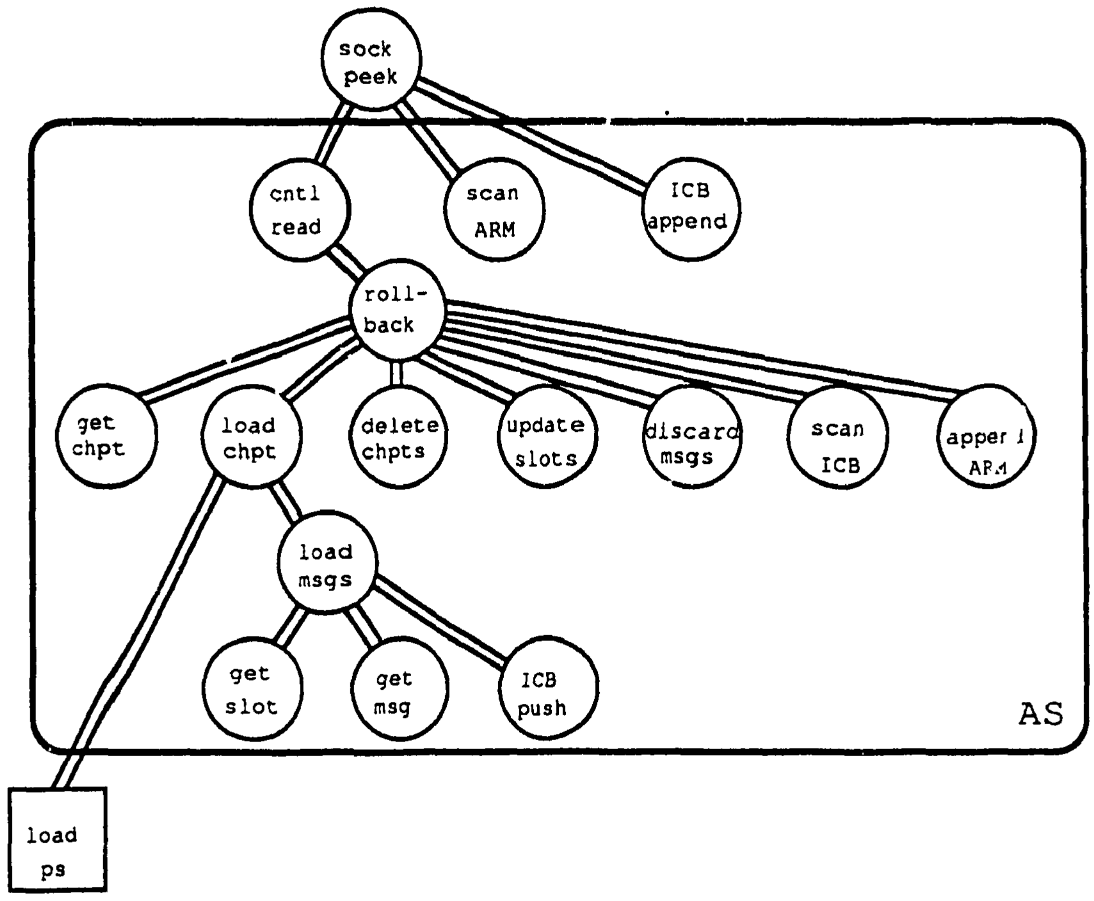


Figure 4.6b AS software in RLV rollback and recovery module

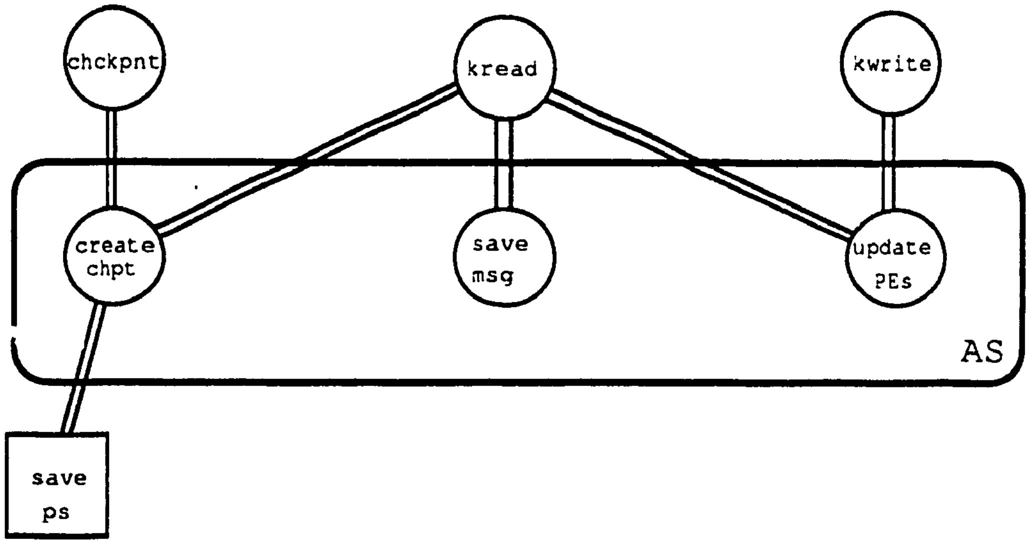


Figure 4.7a AS software in BCS checkpointing module

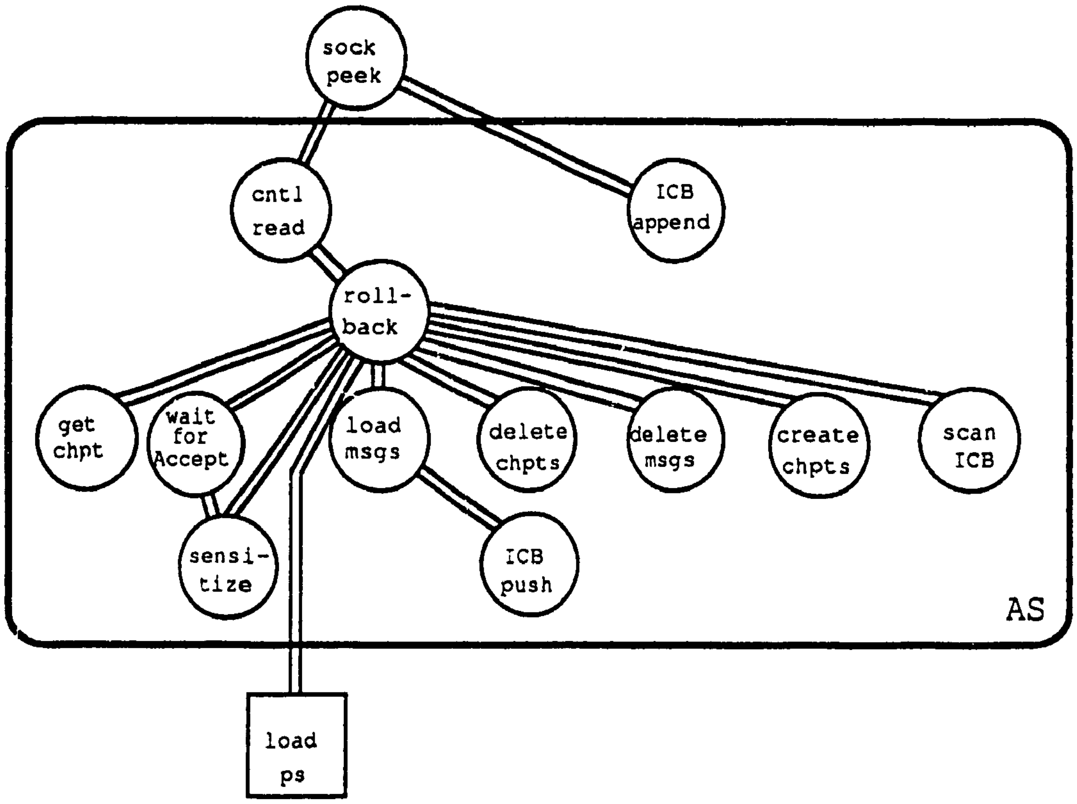


Figure 4.7b AS software in BCS rollback and recovery module

rollback actions; such as the loading of the checkpoints, inserting stored messages to the head of their ICBs, deleting obsolete checkpoints etc. There are, however, a couple of actions carried out during rollback which are unique to BCS. These are the need to wait for an Accept message, handled by Wait_for_Accept, and the augmented manner in which BCS handles the purging of pre rollback messages, which are handled by the Create_chpt and Sensitize functions. Here Create_chpt handles the creation of the hollow checkpoints.

4.3 Measurement Software

The initial goal was to collect the following measurement data with respect to execution time overhead:

D1 The overhead in taking a checkpoint within a process

D2 The overhead in coordination of checkpointing, with respect to message traffic

D3 The overhead in RC checkpointing under specified SIC and message passing environments

D4 The overhead in rollback and recovery as a function of the number of processors participating in rollback

D5 Rollback distance under specified SIC, message passing, and error latency environments, and its dependence on the process (initiator/non-initiator)

D6 Unnecessary rollback distance under specified SIC, message passing, and error latency environments

D7 The overhead due to rollback and recovery control messages with respect to message traffic

However, the measurements dealing with message traffic (D2 and D7) are not feasible in our DCS. The system is sensitive to the number of messages passed and, perhaps more importantly, the number of different sockets used during a given time span. Once these two factors combine to pass some (as yet undetermined) threshold, the system performance, in terms of execution time, becomes very erratic. The execution times fluctuate wildly from run to run. The range of measured times becomes far greater than the observed differences between RLV and BCS. This erratic behavior, we believe, is due to the bottle neck in inter-node communications mentioned in Chapter 2. For this reason, not only were the measurements for D2 and D7 not possible, but the other measurement taking had to be designed using a level of communication which did not perturb the system.

Before we introduce the software used to gather the measurement data, a brief explanation of what we mean by unnecessary rollback (D6) is required. In Fig. 4.8, we see 4 communicating processes, a recovery line, and a fault F1 detected at time T1. The rollback initiating process, P2, should roll back to a position prior to F1 in order to undo its faulty computation. In rolling back to its checkpoint, P2 undoes some non-faulty computation U2, referred to as "unnecessary rollback". Of course, in the rollback initiating process, there will always be some unnecessary rollback, the amount of which is inversely proportional to the rate of SIC checkpointing. As for the other processes involved in the rollback, P1 rolls back a distance R1, all of which is necessary since it was affected by the fault through message M2. P3 rolls back a distance R3, all of which is unnecessary since it never received a post fault message. P4 rolls back a distance R4, some of which, U4, is

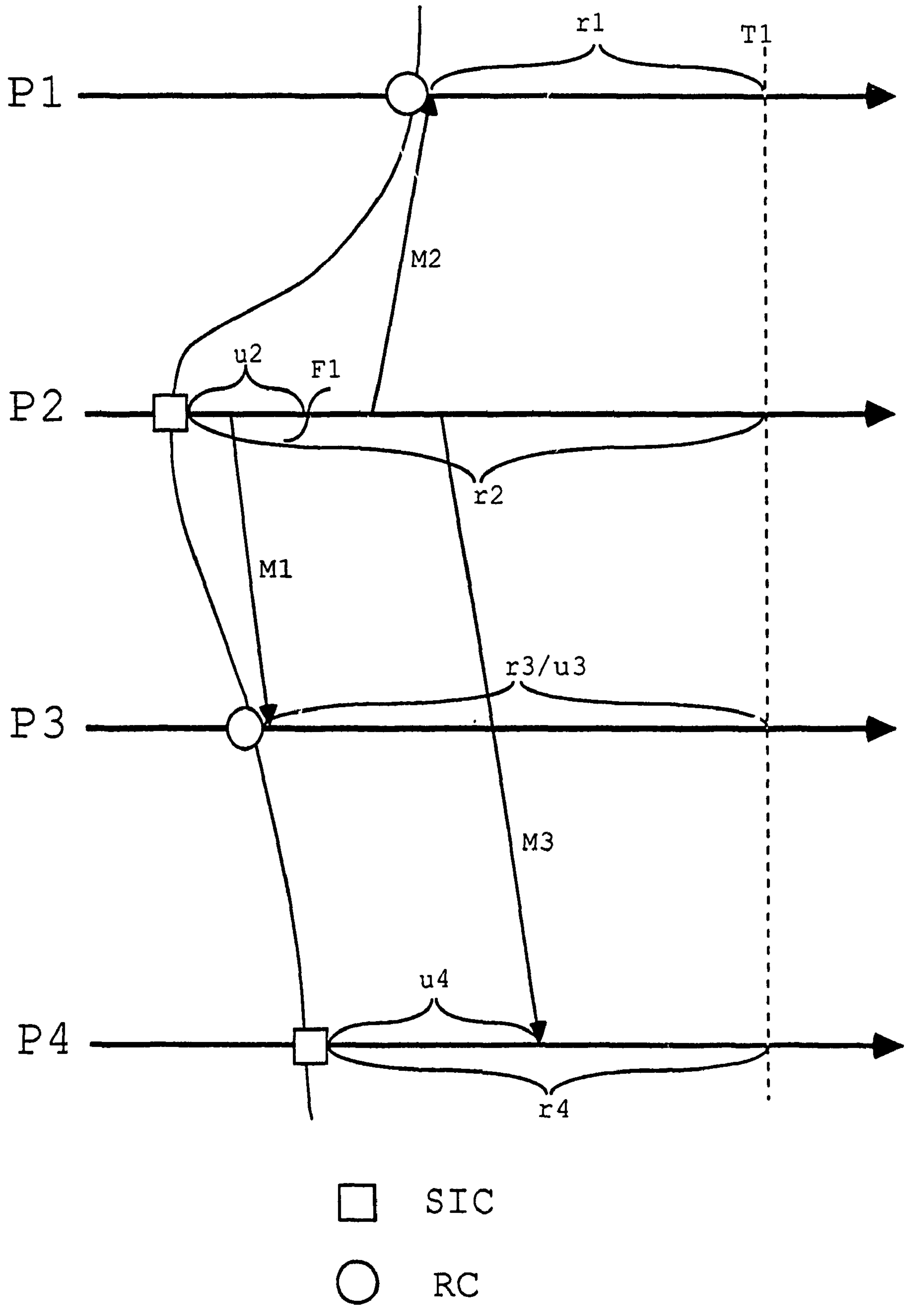


Figure 4.8 Explaining unnecessary rollback

unnecessary since the process only became affected by the fault through message M3. We will see later what effect the different rollback and recovery algorithms, RLV and BCS, have on the amount of unnecessary rollback incurred by the processes.

In order to activate the RRK (to initiate a rollback) we need to first produce the detection of a "pseudo-fault", determine the time the fault occurred, and then select a SIC which was created prior to this fault. As was mentioned in Chapter 2, the error detection was handled by a separate process, the EDM, in the testing of the RRK. This module is basically a random number generator. Once seeded and activated, it randomly signals the application/RRK process through a user-defined signal, and simulates the detection of a fault. Since the occurrence of these faults was unpredictable with respect to the RRK, this arrangement served as a suitable test environment. However, when it came to the measurements, we found that a given fault distribution could not be reliably reproduced with such an arrangement. Therefore, for the purpose of obtaining measurements, the role of the EDM was handled by specific function calls placed in the application program. In this way, we could reproduce exactly the same occurrence of faults for both the RLV and the BCS algorithms.

Once the pseudo-fault is "detected", a timestamp for this fault must be produced. This is handled by an "error handler module" or EHM, which is linked to the RRK as shown in Fig. 4.9. The EHM first produces a timestamp for the pseudo-fault, through the use of another random number generator. The random number produced, symbolizes the time elapsed since the occurrence of the fault, or "fault latency". It is, therefore, subtracted from the present time in order to produce a

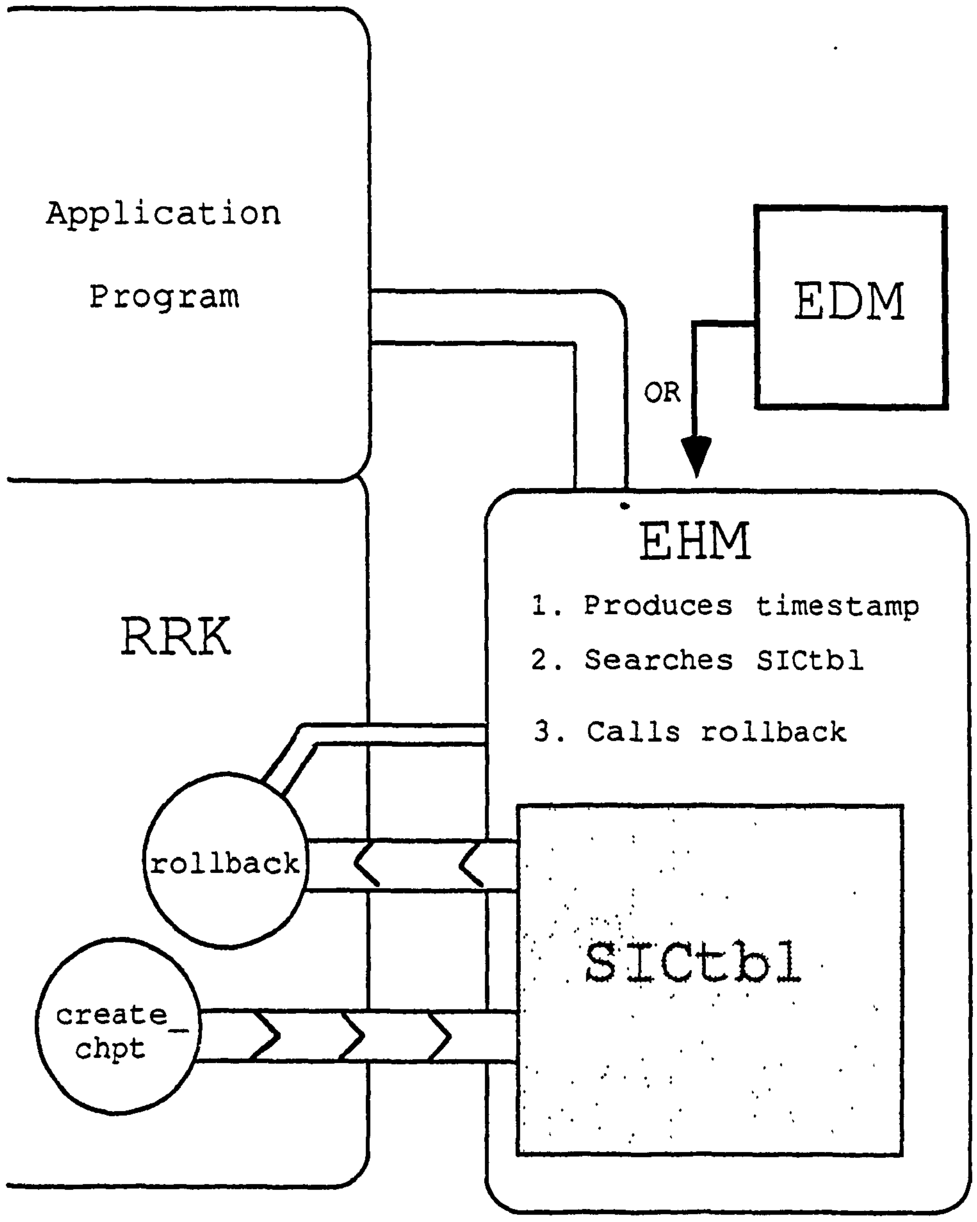


Figure 4.9 Rollback initiation software

timestamp for the pseudo-fault. When a fault latency of zero is required, the random number generator is inactivated and the EHM uses the present time as the timestamp.

Given the pseudo-fault timestamp, the next step is to find the latest SIC which precedes this time. To facilitate this, a new data structure, the SICtbl, is used. It holds a list of the IDs of all the SICs created, along with their times of creation. The data is added to the SICtbl by the EHM through calls from create_chpt. The EHM uses the timestamp created to find a suitable SIC for rollback, and then it activates the AS rollback function by passing it this SIC/recovery line ID.

Referring to the measurements to be taken, we are now able to measure D1, D3 and D4. In addition, the calculations for rollback distance (D5) and unnecessary rollback distance (D6), with respect to the rollback initiating process, can now be done by the EHM since it has access to the present time, the time of the pseudo-fault, and the time of creation of the SIC used in rollback. In order to measure D5 and D6 with respect to the other processes involved in rollback, some additional software is required. If we refer again to Fig. 4.3, we see that to get this data for P1, P3 and P4, each process would require a record of the time of each send event (i.e., those send events which may affect it), the time of each receive they execute, the time of creation for each RC they create, and the timestamp of the pseudo-fault, F1.

To enable these measurements, some additional measurement software had to be integrated into the RRK design, as is shown in Fig. 4.10. Each process P maintains a vector (TVEC) of n time counters, one counter for each of the n processes in the system. The idea is like the CCP

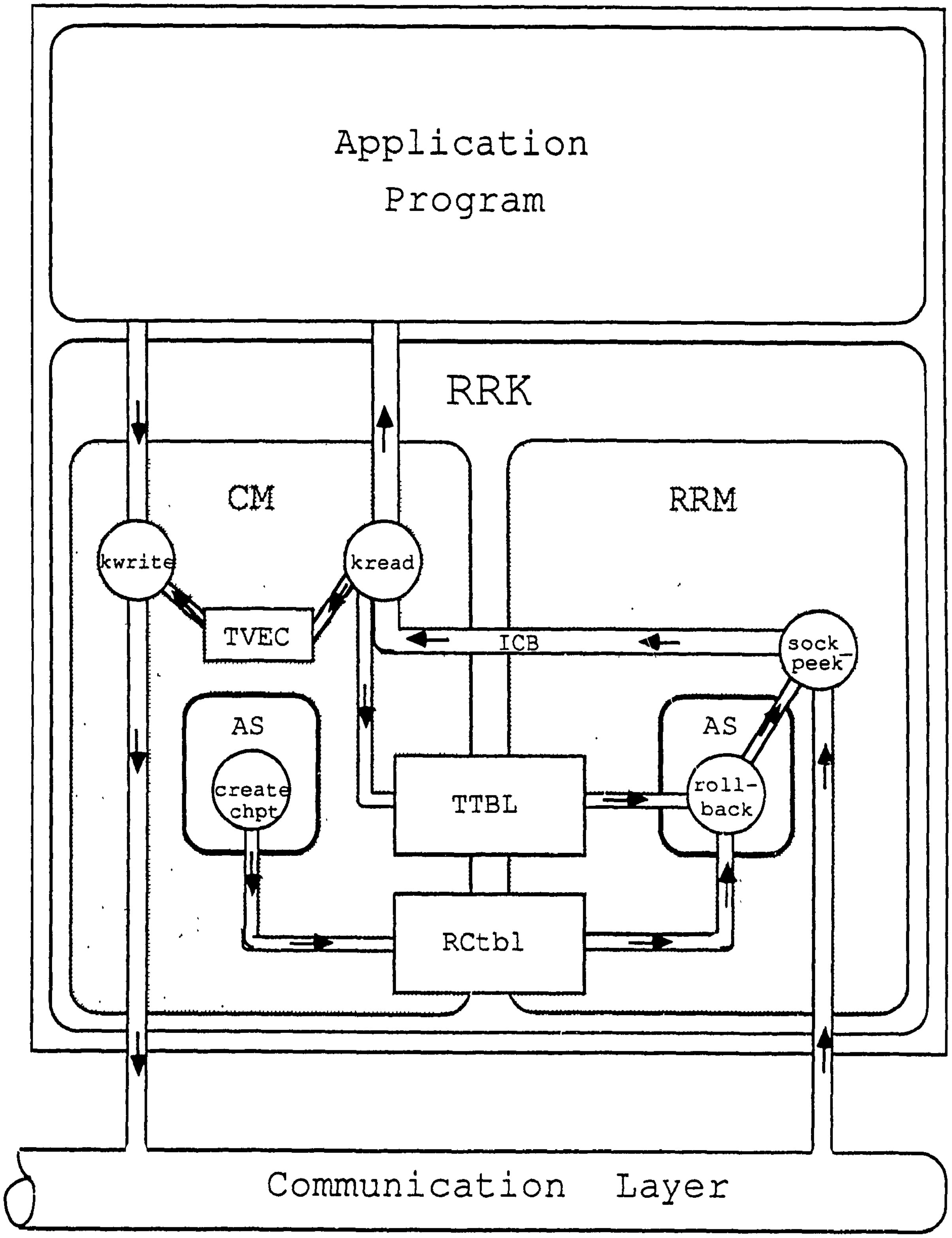


Figure 4.10 Additional measurement software

vectors of the RLV algorithm, only these counters register the time of the latest send event for each of the processes, as perceived by P. These vectors are appended to each of the application messages. The receiving process strips off the TVEC and compares it to its own copy. For each new send event which occurred on another process (including, of course, the one which sent this message), the receiving process creates a new entry in the time table or "TTBL". The table is organized into n lists of entries, one list for each sender in the system. Each entry contains the time of the send event and the time of the local receive event.

In addition, each process also maintains a "RCtbl", similar to the SICtbl of the EHM, only it contains the times that local RCs were created, along with their IDs. This table is updated through calls from create_chpt. It is used to determine the rollback distance for the processes other than the rollback initiator. The reader may recall that, in the case of the BCS algorithm, these processes may roll back to a SIC, as P4 does in Fig. 4.8. Accordingly, for BCS, the RCtbl imports all the SICtbl information in addition to collecting information on the RCs created.

Finally, during rollback, the initiator appends the time of the pseudo-fault (as determined by the EHM) to the rollback control message(s).

With this additional measurement software, we can now carry out the D5 and D6 measurements for the non-initiating rollback processes. Upon being informed of a rollback through a rollback control message, the rollback function, carries out the following actions: it strips off the timestamp for the fault; determines the time of creation of its rollback checkpoint by accessing the RCtbl; searches the appropriate list (which corresponds to the rollback initiator) in the TTBL, for the first send event, if any, which follows the pseudo-fault timestamp. With this information, and the present time, the process can determine D5 and D6.

4.4 Measurement Results and Interpretations

All the data presented in these measurements are in terms of CPU seconds, unless otherwise indicated. Test cases were run for 200 iterations of N-life. These test cases, illustrated by space-time diagrams, exhibit certain message passing and SIC checkpointing behaviors in which the two algorithms will be compared. Two N-life global matrix sizes were used, a small (10 x 20) matrix and a larger (50 x 20) matrix. In the case of the small matrix, the checkpointing action packs 1630 bytes of information, belonging to the application process, into 82 bytes before storing it into dynamically allocated memory. With the larger matrices, 8030 bytes of information were packed into 282 bytes before being stored. This large reduction was possible because the bulk of the checkpoint information is composed of simple on/off data for the individual cells of the matrix. In N-Life this data is in an integer format. The RRK packs each of these integers into a single bit.

In most tables (except Table IV) two processors, P1 and P2, are shown for the RLV and BCS algorithms. The reason for listing only two processors here is as follows: as stated earlier, the workstations do not have their own local copy of the Yellow Pages directory, and they must, therfore, access the central file server in order to carry out inter-node communications. In this environment, a given inter-node communication, may be competing with other currently executing routines, or even other communications due to the same distributed application, for the single Yellow Pages resource. In earlier attempts at collecting measurements, we used various kinds of distributed test cases. For example, in one such experiment, we used three processes which were carrying out checkpointing, and which communicated frequently, and ran them for a thousand iterations. In Fig. 4.11, we present the execution times of one of these three processes. Twelve measurements were made over about a three hour period. The execution times range from a high of about 290 seconds to a low of 230 seconds, which corresponds to more than a 20 percent variance. As we will see, BCS and RLV normally differ by only a few percent, in terms of checkpointing overhead. The wide variance in the measurement error swamps the comparison of the measurement values between RLV and BCS. To aquire meaningful data we, therefore, progressively trimmed-down our test cases to 2-process models, which communicate to a minimal degree, the results of which follows:

Checkpointing Overhead

In a system where all processes take SICs, the following things can be observed. In BCS, the maximum number of checkpoints taken by a process is bounded by the system-wide maximum number of SICs taken by a single process, times two. In RLV, the maximum number of checkpoints taken by a given process is bounded by the sum of SICs taken in the whole system. In BCS, the minimum number of checkpoints taken by a process is bounded by its own count of SICs. RLV is more reflective of the nature of the message traffic, but, if a given process receives no messages then its lower bound on checkpoint creation is also its own SIC

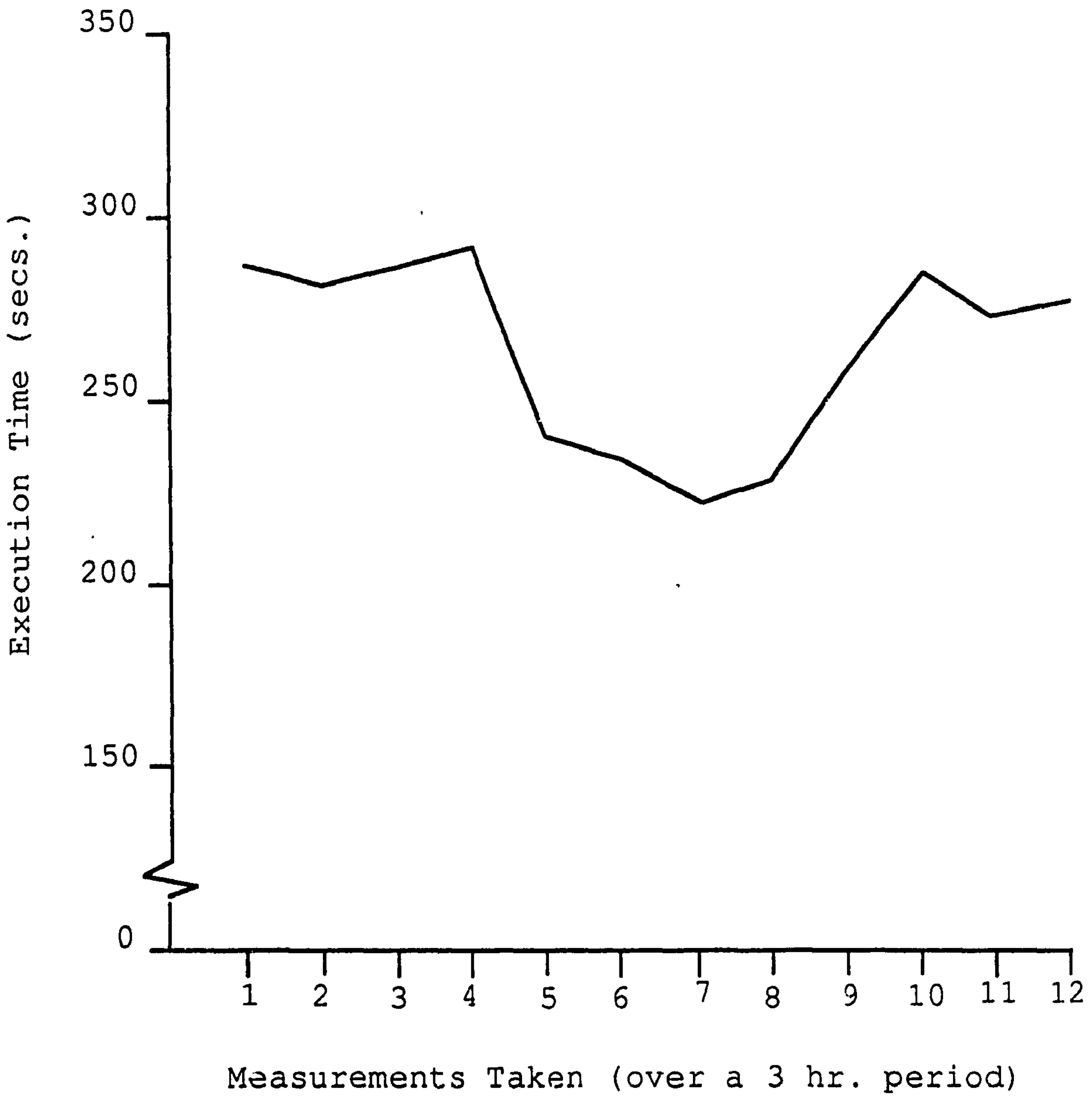


Figure 4.11 Execution Times: for one process of a 3-process, 1000 iteration, large problem size, test case.

count.

More specifically, there are three factors which affect the checkpointing comparisons between RLV and BCS. These are...

i - The relative SIC counts of processes

ii - The flow of messages between processes

iii - The rate of message passing compared with the SIC rate

BCS causes the least checkpointing overhead in an environment with uniform SIC counts between the processes in the system. Any message passing behavior, in this SIC scenario, will favor BCS. RLV, on the other hand, out performs BCS if processes exhibit the following: heterogeneous SIC counts; lopsided flow of information from the process with the higher SIC count; a message passing rate less than the SIC rate for the process with the higher SIC rate.

With these general observations, we will now examine three different checkpointing scenarios, in which the two algorithms will be compared.

Case C1:

Fig. 4.12 shows the first test case to be examined. Here, P1 creates a SIC every 4 iterations, and sends a message to P2, 1 iteration after the creation of each SIC. P2 creates no SICs and sends no messages. This is a case where both algorithms behave the same, since they both cause P2 to create a RC, upon the receipt of each message. In this environment, we can determine the overhead incurred by the two algorithms, due to single SICs and RCs. In Table 4.1, we see that the

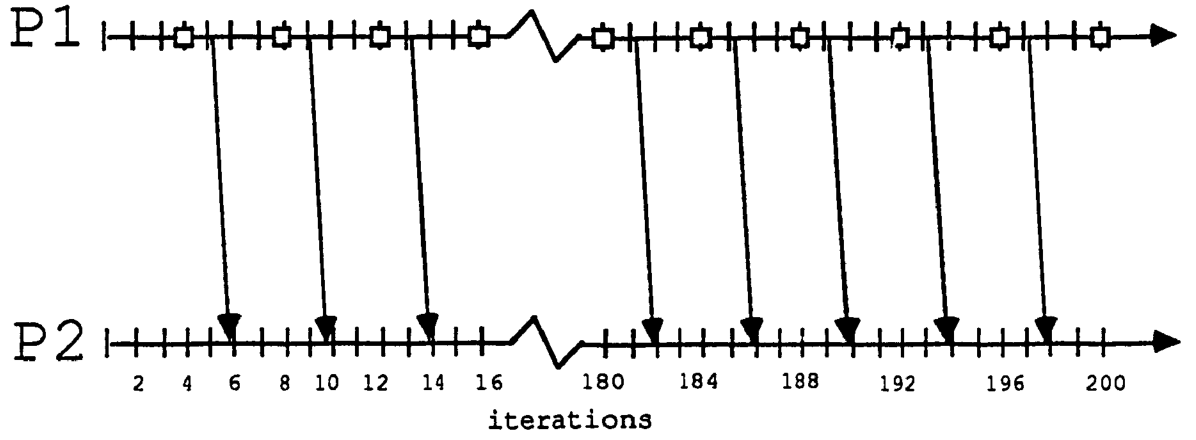


Figure 4.12 A test case for a measure of checkpoint overhead (for Table 4.1)

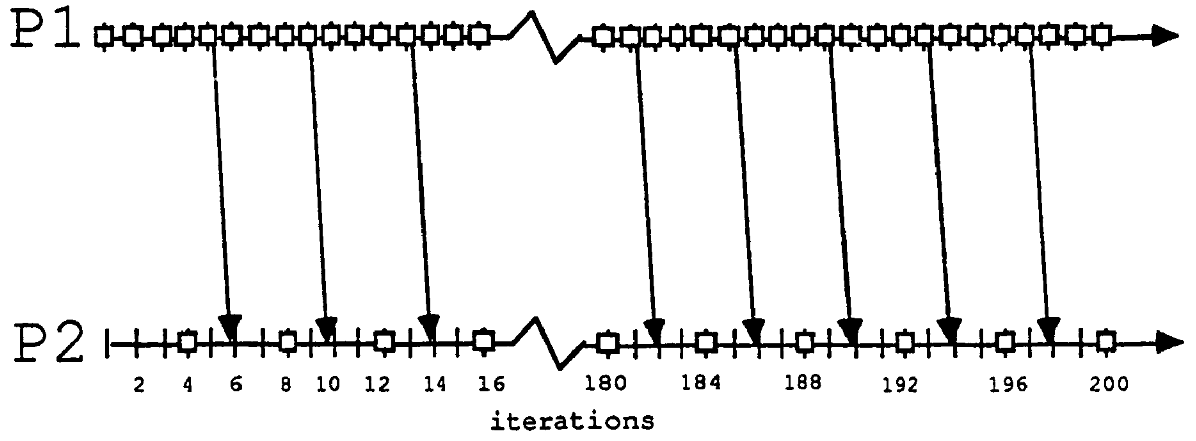


Figure 4.13 An uneven SIC environment (for Table 4.2)

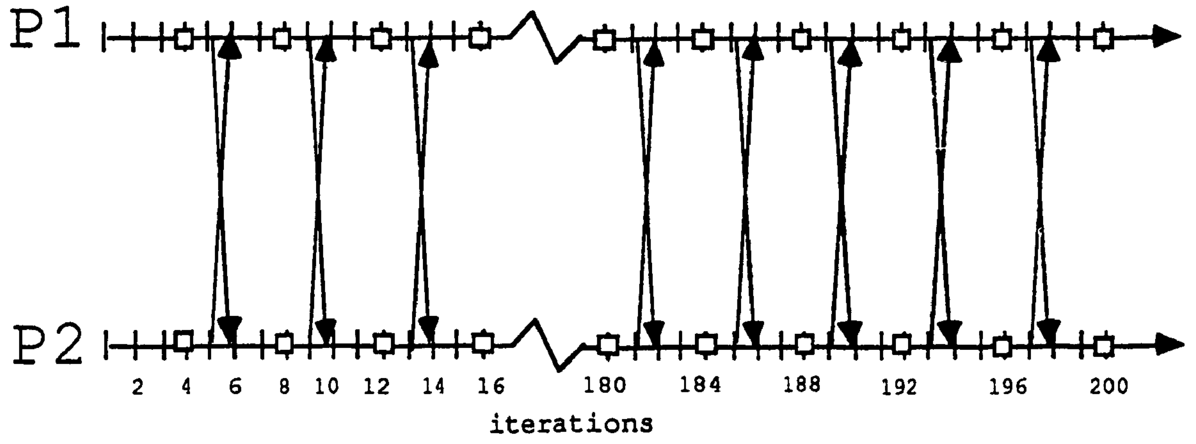


Figure 4.14 An iso-SIC environment (for Table 4.3)

Table 4.1

OVERHEAD IN TAKING SIC AND RC CHECKPOINTS

	Small Checkpoint				Large Checkpoint			
	RLV		BCS		RLV		BCS	
	P1	P2	P1	P2	P1	P2	P1	P2
1. No Chckpts.	3.89	4.09	3.88	4.08	18.50	18.66	18.48	18.67
2. Chckpts.	4.30	4.51	4.33	4.52	20.01	20.16	20.14	20.35
3. Overhead	0.41	0.42	0.45	0.44	1.51	1.50	1.66	1.68
4. SICs	50	0	50	0	50	0	50	0
5. RCs	0	49	0	49	0	49	0	49
6. Ov./Chckpt. (msec.)	8.2	8.6	9.0	9.0	30.2	30.6	33.2	34.3

Table 4.2

CHECKPOINTING OVERHEAD IN AN UNEVEN-SIC ENVIRONMENT

	Small Checkpoint				Large Checkpoint			
	RLV		BCS		RLV		BCS	
	P1	P2	P1	P2	P1	P2	P1	P2
1. No Chckpts.	3.89	4.09	3.88	4.08	18.50	18.66	18.48	18.67
2. Chckpts.	5.47	5.59	5.55	5.69	24.04	23.98	24.19	24.38
3. Overhead	1.58	1.50	1.67	1.61	5.54	5.32	5.71	5.71
4. SICs	200	50	200	50	200	50	200	50
5. RCs	0	49	0	148	0	49	0	148
6. Poll.(in 1)	-	0.04	-	0.04	-	0.07	-	0.07
7. Poll.(in 2)	-	0.65	-	0.59	-	2.22	-	2.01

overhead due to checkpointing, row 3, is calculated by subtracting the times without checkpointing (obtained with no RRK attached), row 1, from the corresponding execution times experienced with checkpointing, row 2. By dividing these overheads by the number of checkpoints taken by the process, either SICs or RCs, the overhead per checkpoint, row 6, is obtained (shown in milliseconds). We see that, for both the small and large checkpoints, SICs and RCs take about the same amount of time to create. Also, RLV seems to take a slightly lesser time in the creation of its checkpoints than does BCS. For example, in the case of the small checkpoints, RLV takes 8.2 msecs. for a SIC as compared to 9.0 msecs. for each SIC taken by BCS. This marginal difference may be due to the updating of the PEsets which BCS must carry out for each message sent and received.

Case C2:

In this test case, seen in Fig. 4.13, P1 takes SICs more often, 1 per iteration, than does P2, 1 per 4 iterations. This causes P2 to create RCs with the reception of each message. We see that, in Table 4.2, row 5, BCS causes P2 to take about 3 times as many RCs, 148, as P2 creates with RLV, 49. This is due to the need, in BCS, for each process to keep the same number of checkpoints. Which causes P2, in this case, to create two "dummy" checkpoints in addition to one "real" checkpoint, with the reception of each message, in an effort to catch up to P1. The extra overhead incurred by P2, due to the dummy checkpoints, is not as much as one might expect (e.g., in the case of small checkpoints, 1.61 secs. for BCS vs 1.50 for RLV). This is for two reasons.

First, the dummy checkpoints are implemented as hollow checkpoints;

they contain no process state of their own, but contain a pointer to the real checkpoint's state. This results in much less overhead in a dummy checkpoint than in a real checkpoint.

The second reason requires a little explanation. Observe that the overhead experienced by P2, with RLV, is much larger in Case 2 than it was in Case 1 (e.g., for the large checkpoints, 5.32 secs. in Case 2 compared with only 1.50 secs. in Case 1). Even if we account for the SICs taken by P2 and the messages it must store, the overhead seems excessive, since the same number of RCs are taken in each case. This anomaly is explainable if we look at, in Case 2, the time spent polling for messages when no checkpoints were taken, row 6, as compared to the time spent polling with checkpointing, row 7. We see that, for both algorithms, a large portion of the overhead experienced by P2 is due to polling! Using the same example of RLV and the large checkpoints, 2.22 secs. of the total of 5.32 secs. of overhead is due to polling. The reason for this excessive polling is due to the loosely-synchronous nature of N-life, discussed in Chapter 3. That is, since P2 needs the information from P1 to proceed with its portion of N-life, and since P1 is slowed down due to the large number of SICs created (200) P2 is forced to wait for P1 at each point where they communicate.

Case C3:

In this case, Fig. 4.14, an iso-SIC sample application is shown. This is an environment where BCS causes less RCs than RLV, no matter what the message passing behavior. In this particular case, RLV creates a RC with the reception of each message, while BCS creates no RCs. In addition, RLV stores each message to the preceding SIC, while

BCS does not. These extra actions cause an overhead of 1.3 secs. for RLV, in P1 with small checkpoints, vs 0.59 secs. for BCS, as seen in Table 4.3, row 3.

In conclusion, we have observed that, in general, checkpointing does not cause a large overhead, e.g., about 10 % in Case C1. RLV was observed to take slightly less time than BCS, in the creation of a single checkpoint. Also, both algorithms incur about the same overhead due to checkpointing in uneven-SIC environments, with perhaps a marginal advantage experienced by RLV. BCS causes less overhead, in the iso-SIC environment, than does RLV. Finally, the synchronization properties of the application were seen to have a large effect on the performance of the distributed application, which in turn may overshadow the relative performances of the checkpointing algorithms.

Rollback Overhead

Case R1:

In this case, we will demonstrate the relative overheads experienced by the two algorithms, due to their rollback control messages. The BCS algorithm uses a serialized form of rollback control messages which causes a momentary "freezing" of the processes during the rollback procedure, while they wait for the return of an Accept message. In an environment where the processes run on separate processors, a marked slow-down can be observed. In Fig. 4.15, we see that with the addition of processors to the rollback procedure, the rollback initiator is required to wait for longer and longer periods of time. In contrast, the RLV algorithm does not cause any freezing of the processes during

Table 4.3

CHECKPOINTING OVERHEAD IN AN ISO-SIC ENVIRONMENT

	Small Checkpoint				Large Checkpoint			
	RLV		BCS		RLV		BCS	
	P1	P2	P1	P2	P1	P2	P1	P2
1. No Chckpts.	5.17	5.18	5.19	5.18	19.86	19.86	19.72	19.70
2. Checkpts.	6.47	6.48	5.78	5.81	22.81	22.81	21.66	21.66
3. Overhead	1.30	1.30	0.59	0.63	2.95	2.95	1.94	1.96
4. SICs	50	50	50	50	50	50	50	50
5. RCs	49	49	0	0	49	49	0	0

Table 4.4

ROLLBACK OVERHEAD AS A FUNCTION OF THE NUMBER OF PROCESSORS INCLUDED IN ROLLBACK

	RLV				BCS			
Pattern #	1	2	3	4	1	2	3	4
1. No RBs.	4.20	4.52	4.52	4.54	4.22	4.54	4.55	4.55
2. 10 RBs.	7.72	8.15	8.16	8.26	7.72	9.47	9.90	10.28
3. Overhead	3.52	3.63	3.64	3.72	3.50	4.93	5.35	5.73
4. Ov./RB.	0.35	0.36	0.36	0.37	0.35	0.49	0.54	0.57

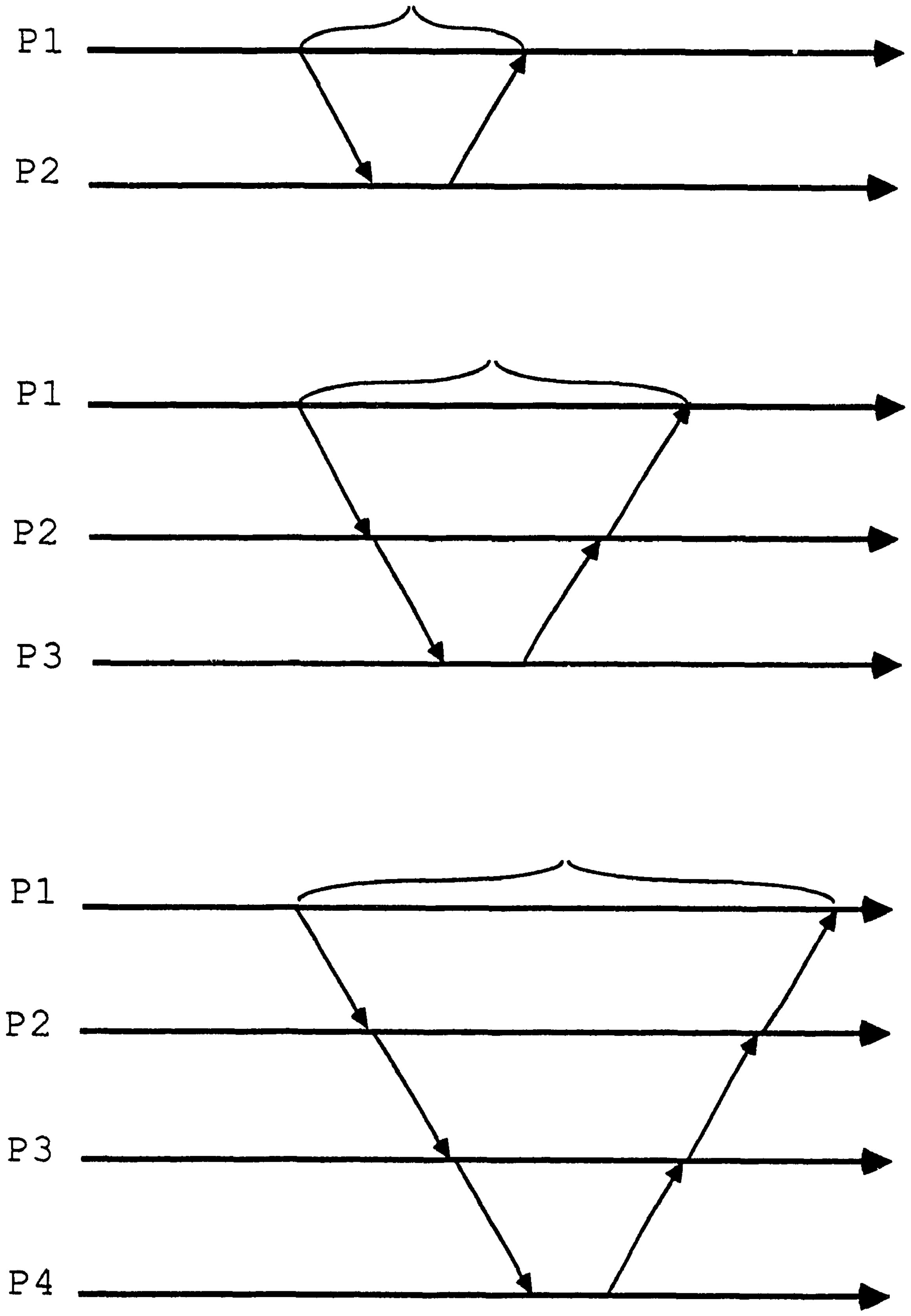


Figure 4.15 Effect of serialized control messages.

rollback; it returns control to the application program immediately after sending its rollback control messages.

To measure the overhead incurred by the serialization of rollback control messages, we have arranged the four different test patterns shown in Fig. 4.16. With these patterns, the four processes, running on separate processors, go from a non-communicating environment (pattern 1, at the top) to an environment where each process is involved in some communication (pattern 4, at the bottom). The increases of communication will cause more and more processes to be involved in the rollbacks, which are initiated by process P1.

In Table 4.4, the different execution times for process P1 are displayed. When no rollbacks are initiated the relative times for BCS and RLV are equal since the checkpointing behavior is the same, as is seen in row 1. There is a small increase of about 0.3 seconds from pattern 1 to pattern 2, which is maintained through patterns 3 and 4. This is due to the messages which P1 must send in the latter patterns. For the data in row 2, 10 uniformly distributed rollbacks have been initiated in each of the four patterns. We see that, the execution times are equal for RLV and BCS when only P1 is rolled back (pattern 1), since, in this case, BCS sends no rollback control messages. However, as more and more processes are added to the rollback procedure, P1 begins to display a marked slow-down, in the case of BCS. To more clearly illustrate this effect, we refer to row 3, which shows the difference of rows 1 and 2. The calculation removes the slight slow-down of P1 due to message passing and isolates the rollback overhead. This is illustrated in Fig. 4.17. The last row shows the overhead due to a single rollback.

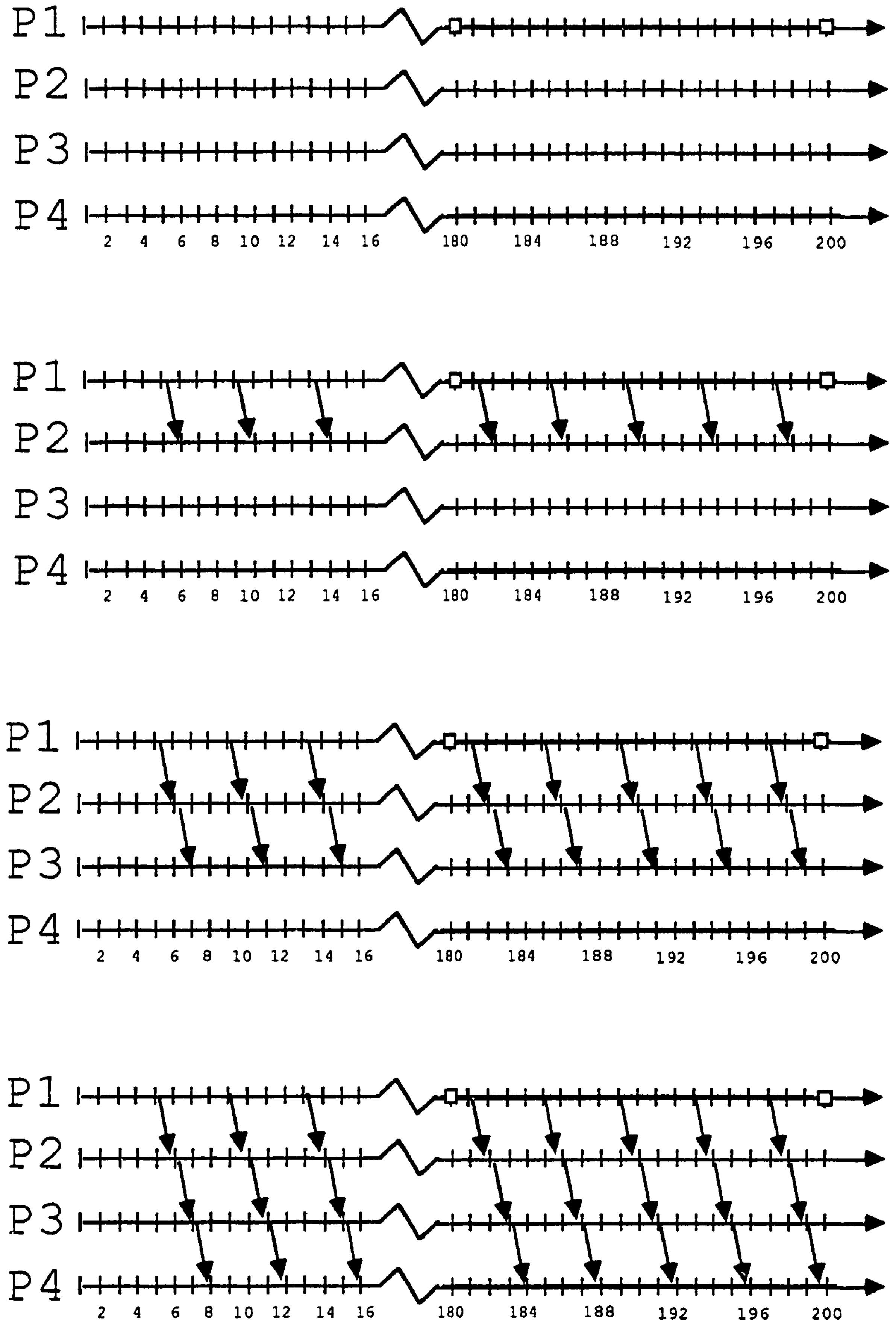


Figure 4.16 Increasing message passing in order to increase the number of processes included in rollback and recovery

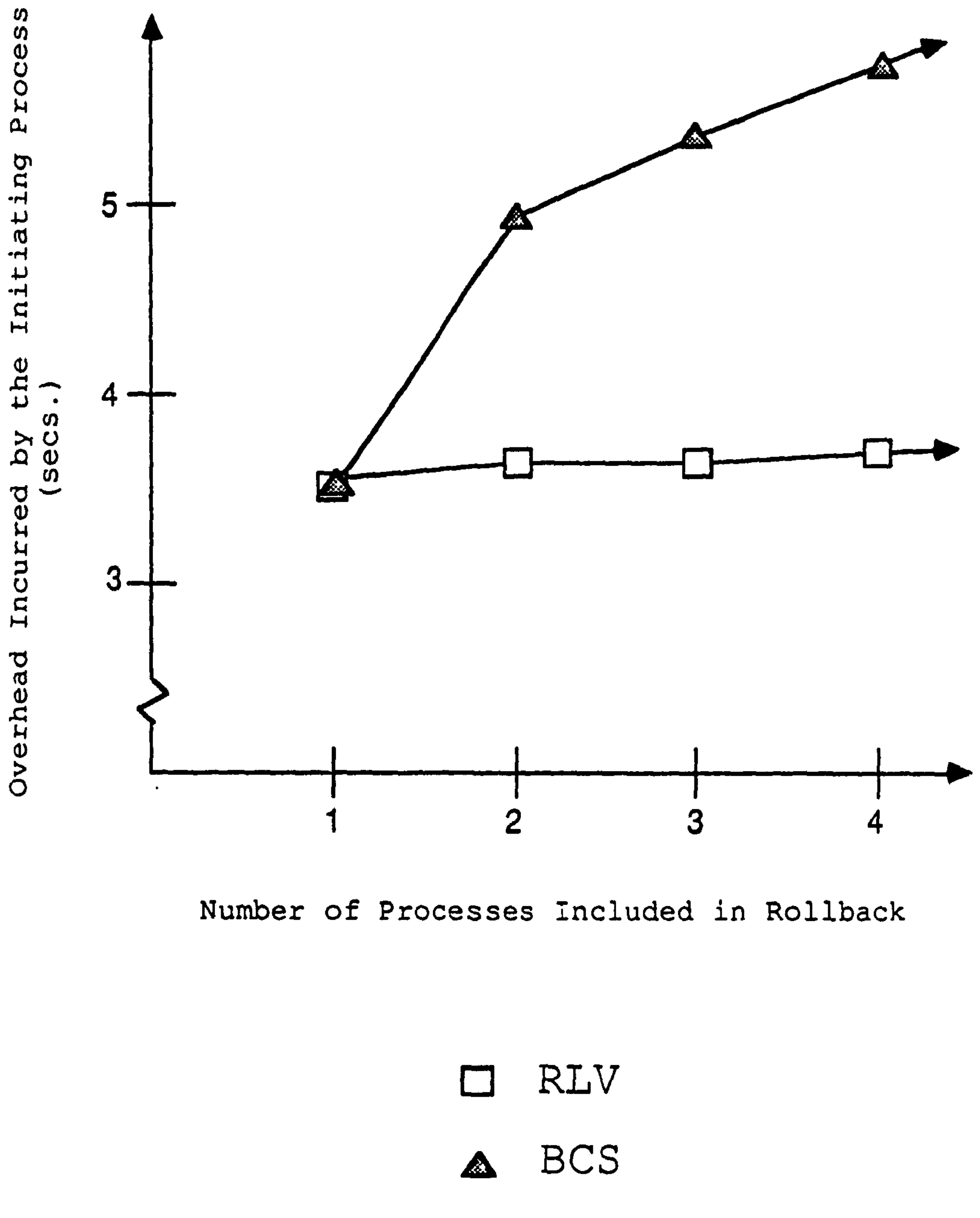


Figure 4.17 Algorithm rollback overhead

Case R2:

The factors which cause BCS to exhibit less overhead in checkpointing also cause it to incur a larger overhead in rollback and recovery. That is, since BCS does not employ dependency tracking in its creation of RCs, as does RLV, it causes the processes, other than the rollback initiator, to roll back a farther distance, in general. In the following cases, only the larger (50 x 20) matrix size was used.

In Fig. 4.18, we see a test case where BCS exhibits a lower overhead due to checkpointing and a higher overhead due to rollback and recovery, this is an iso-SIC environment. In Table 4.5a, the number of checkpoints taken along with the time of execution for checkpointing (row 1) is shown. Table 4.5b shows the rollback overhead due to 10 evenly spaced rollbacks with an error latency of zero seconds. We see that most of the overhead due to rollback, row 5, can be attributed to the rollback distance, row 2, that is, the computation which must be redone. In the case of the initiator process P1, BCS exhibits a higher overhead, mainly due to its serialized rollback control messages. In P2, BCS again exhibits a slightly higher overhead, due to the fact that it rolls back farther.

To obtain the data in Table 4.5c, an error latency, uniformly distributed between 0 and 1 second, has been used. We see that, as the error latency increases, so does the rollback distance, in both RLV and BCS. Furthermore, it appears as though the increased overhead, when compared to Table 4.5b, is directly proportional to the increase in latency (i.e., about 0.5 seconds per rollback), although further measurements should be taken to verify the linearity behavior. It would appear, then, that latency has a predictable effect on the rollback

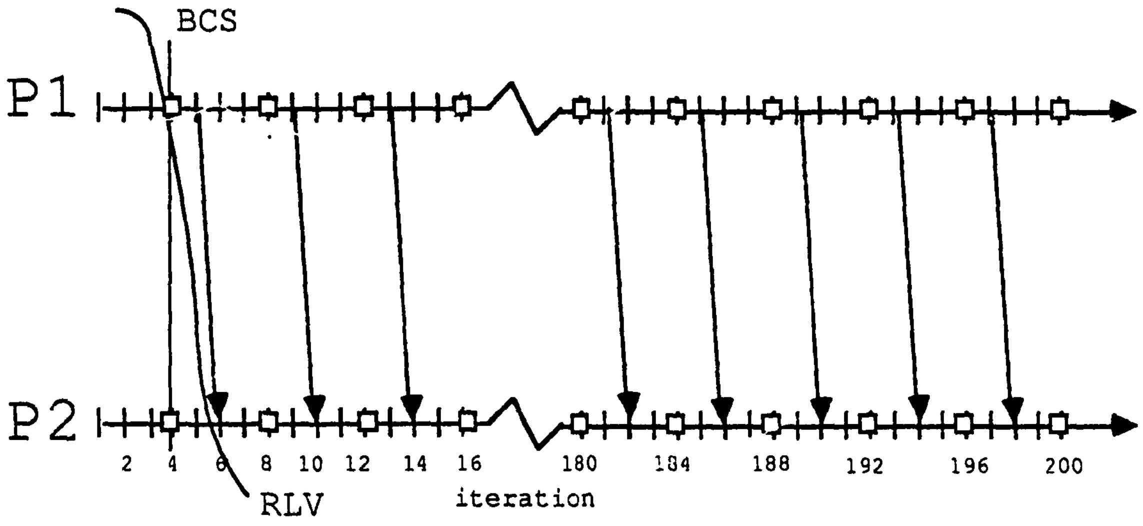


Figure 4.18 An iso-SIC environment (for Table 4.5)

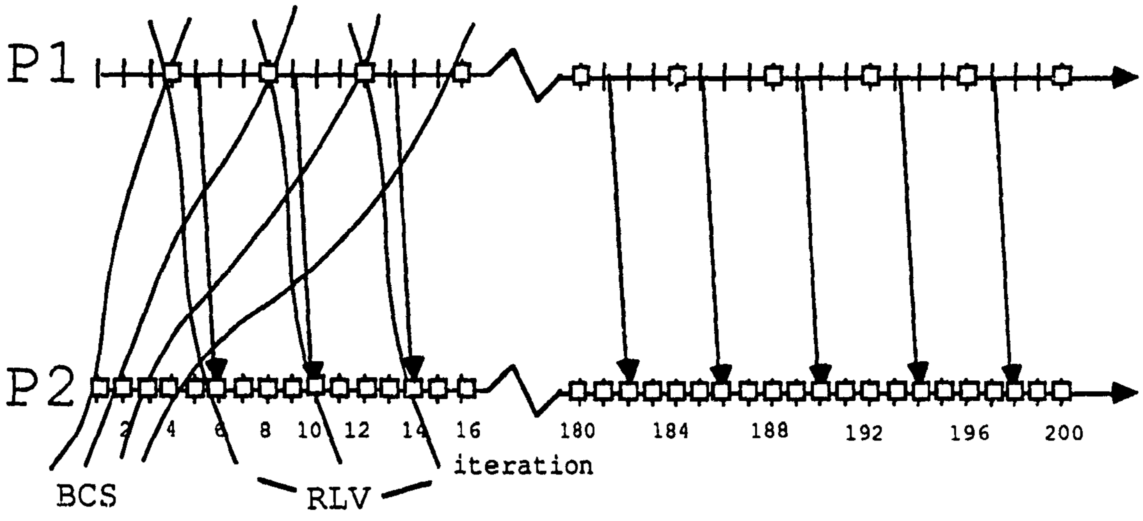


Figure 4.19 An uneven SIC environment (for Table 4.6)

Table 4.5a

CHECKPOINTING IN AN ISO-SIC ENVIRONMENT

	RLV		BCS	
	P1	P2	P1	P2
1. Control (No RBs)	20.01	21.35	20.14	20.29
2. SICs	50	50	50	50
3. RCs	0	49	0	0

Table 4.5b

ROLLBACK OVERHEAD IN AN ISO-SIC ENVIRONMENT
(0s. error latency)

	RLV		BCS	
	P1	P2	P1	P2
1. With 10 RBs.	23.48	24.98	25.05	25.15
2. RB. Distance	2.62	2.67	2.64	2.77
3. Unnecessary RB. Dist.	2.62	2.67	2.64	2.77
4. Iterations Repeated	29	29.6	29	40
5. Overhead (row1-Cntl.)	3.47	3.63	3.59	4.86

Table 4.5c

ROLLBACK OVERHEAD IN AN ISO-SIC ENVIRONMENT
(max. 1s. error latency)

	RLV		BCS	
	P1	P2	P1	P2
1. With 10 RBs.	27.78	28.80	29.56	29.68
2. RB. Distance	7.00	6.36	7.04	7.13
3. Unnecessary RB. Dist.	2.15	3.38	2.19	3.69
4. Iterations Repeated	72.2	68.8	73	83.2
5. Overhead (row1-Cntl.)	7.77	7.45	9.42	9.39

distance regardless of the rollback and recovery algorithm used.

Table 4.6 (a to c) is similar to Table 4.5, except that the message pattern and SIC pattern are unfavourable to BCS, depicted in Fig. 19. By comparing row 2 in Table 4.5b with the corresponding row in Table 4.6b, we observe that the initiator process P1 experiences about the same rollback distance. However, in the case of the non-initiator process P2, the rollback distance is much worse for BCS.

The unnecessary rollback distance (row 3 in Tables 4.5b, 4.5c, 4.6b and 4.6c) is smaller in RLV because of its way of taking RC checkpoints. When the checkpointing is uniform, as in iso-SIC (Table 4.5), BCS incurs a relatively low overhead of unnecessary rollback. However, if the message flow and SIC pattern is uneven, such as in Fig. 19 (Table 4.6), the unnecessary rollback could become substantial in the case of BCS.

4.5 Remarks and Conclusions

The storage cost for a single checkpoint depends upon the application program, the size of the problem, and the way in which the RRK packs the information for the sake of the internal storage. In the case of N-Life, for the smaller (10 x 20) matrix, 1630 bytes of information was packed into 82 bytes for each process, before being stored. Similarly, in the case of the larger (50 x 20) matrices, 8030 bytes are packed into 282 bytes of data for each checkpoint. In general, the total storage cost for checkpointing will depend upon the space required per checkpoint and the number of checkpoints taken. We have indicated, in the various tables, the number of SICs and RCs taken in our experimental study. N-Life is a simple application program and

Table 4.6a

CHECKPOINTING IN AN UNEVEN-SIC ENVIRONMENT

	RLV		BCS	
	P1	P2	P1	P2
1. Control (No RBs)	20.04	25.34	20.08	25.07
2. SICs	50	200	50	200
3. RCs	0	49	0	0

Table 4.6b

ROLLBACK OVERHEAD IN AN UNEVEN-SIC ENVIRONMENT
(0s. error latency)

	RLV		BCS	
	P1	P2	P1	P2
1. With 10 RBs.	23.38	27.16	25.26	43.61
2. RB. Distance	2.63	1.07	2.63	15.88
3. Unnecessary RB. Dist.	2.63	1.07	2.63	15.88
4. Iterations Repeated	29	13.6	29	140.2
5. Overhead (row1-Cntl.)	3.34	1.82	5.18	18.54

Table 4.6c

ROLLBACK OVERHEAD IN AN UNEVEN-SIC ENVIRONMENT
(max. 1s. error latency)

	RLV		BCS	
	P1	P2	P1	P2
1. With 10 RBs.	27.79	30.32	29.75	48.13
2. RB. Distance	7.04	3.92	7.08	19.70
3. Unnecessary RB. Dist.	2.19	2.41	2.23	19.70
4. Iterations Repeated	73	39.8	73	168.3
5. Overhead (row1-Cntl.)	7.75	4.98	9.67	23.06

it was relatively easy to decide when to take a checkpoint (the SICs are taken at the beginning of an iteration).

With respect to measurements for more complex test cases, we are in the process of adding local stable stores to the diskless workstations and modifying the communication subsystem to have a local copy of the Yellow Pages directory. Then we should be able to include several processors under all cases of message flow. At this stage, we store the checkpoints in the local RAM of the diskless nodes, which limits the size of the application program we can run. This constraint will be relaxed when local stable stores are added. However, the overall results and trends reported in this paper will not be affected.

Until the above improvements are made, we can offer suggestions on carrying out measurements in the present environment. From the graph in Fig. 4.11, we see that the execution times seem to vary with respect to some slowly changing system overhead. This overhead may be caused by system routines or other application programs. To limit the effects of this overhead, we suggest the following: design test cases which can be run in less than a couple of minutes; run the measurements in pairs (e.g., RLV vs BCS, or with checkpointing vs without checkpointing), one after the other, so they will experience about the same system overhead; express the results in terms of relative differences rather than absolute times. To limit the time of the test cases, one can reduce the number of iterations and/or reduce the problem size (matrix size). The latter should be done with care, however, since shortening the cycle time may increase the frequency of communication, and result in competing communications within the distributed application itself.

In Case C2, we saw how the loosely-synchronous nature of N-Life has a masking effect on the relative performances of the rollback and recovery algorithms, in terms of checkpointing overhead. Although no data has yet been acquired, it is expected that a similar masking effect is experienced during the rollback operation. To illustrate this, we refer to Fig. 4.20. Here, we see that P2 has rolled back a distance of d3, which corresponds (disregarding algorithm overhead) to its rollback overhead. P1, on the other hand, rolls back a distance of only d2, which would imply that P1 should experience less rollback overhead than P2. However, P1 immediately blocks for message M1, and remains blocked for period of d1 seconds. One can see that the sum of d1 and d2 is equal to d3. The conclusion which can be drawn from this, is that the completion times for the processes of a distributed application, which is either loosely-synchronous or synchronous, are bounded by the slowest process in the distributed system (in this case the rollback initiator). Furthermore, the time saved in using one rollback and recovery algorithm over another may be overshadowed by the slow-down due to the synchronization needs of the application program. An algorithm such as RLV, which minimizes the rollback distance through the use of dependency tracking, is severly hindered if the application test-bed is not well suited. In order for RLV to be of the most use, it should be used with either fully asynchronous applications, or in an environment where each node in the system supports multiple processes, in which case a blocked process would be exchanged for another process which is ready to use the CPU.

The time overhead due to the use of a rollback and recovery scheme has several components: (i) checkpointing time, (ii) time required to rollback upon the detection of a fault, (iii) rollback distance. They

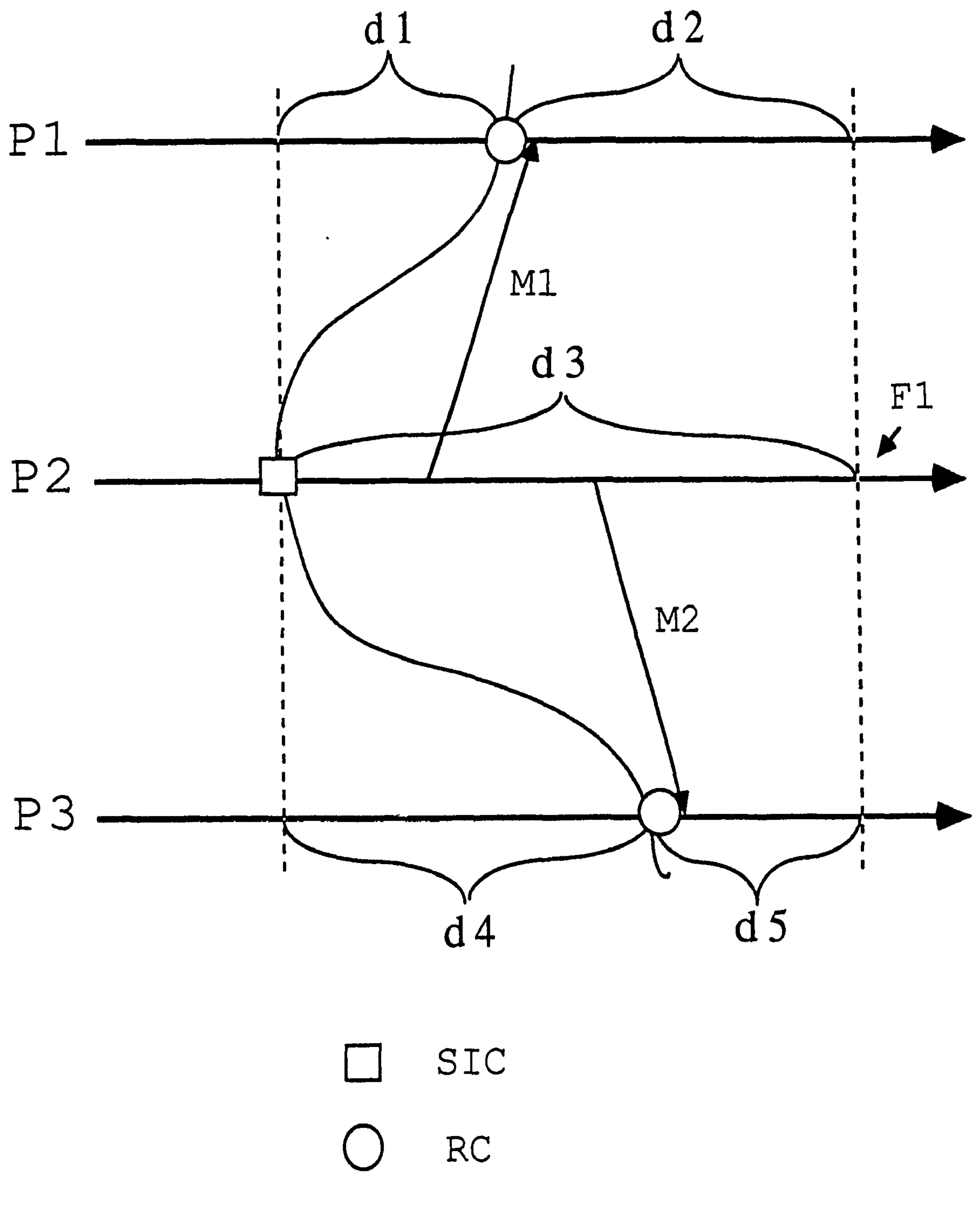


Figure 4.20 Effect of loosely-synchronous N-Life on rollback overhead.

are dependent on the application program, the distributed computing system, the rollback and recovery algorithm, and the design and implementation of the RRK software. For our experimental study the checkpointing time and the rollback distances were presented in the various tables. The average time taken for the N-Life program to rollback is found to be about 71 milliseconds for RLV and 255 milliseconds for BCS. When more processes participate in the rollback, this time will increase linearly for BCS, whereas it will stay constant for RLV. The difference is due to the way in which the two algorithms coordinate and disseminate the rollback control messages.

The question of whether such an RRK, as has been described, can be of practical value as a means of fault tolerance, can only be answered by the individual application programmer himself. To illustrate this point we use the following hypothetical example.

Let us assume that a programmer has distributed a program between four processes, and that the processes take a maximum of 100 seconds to complete 1000 iterations, in the absence of faults. The programmer wishes to use the RRK in order to make the program fault tolerant, but only if the RRK adds no more than 20 percent, i.e., 20 seconds, to the execution time of his program (disregarding any overhead incurred due to an error detection software). He predicts that a maximum of 2 faults might occur during one run, and that the maximum latency of these faults could be 0.9 seconds. He is told that, each checkpoint takes 30 milliseconds to create, and that, due to the liberal message passing of his program, each SIC taken by a process is sure to cause the creation of an RC on the other three nodes (here we assume RLV as the RRK algorithm). Finally, he is told that, upon detection of a fault, the

RRK takes 100 milliseconds to perform a rollback. Can he use the RRK? If so, what SIC checkpointing rate should he employ?

We assume that each process creates the same number of SICs and that a constant SIC checkpointing rate is used. Furthermore, we calculate the overhead with respect to the rollback initiator, i.e., the process which should experience the largest rollback overhead. The problem can be solved as follows:

(i) Checkpointing Time:

Let n = the number of SICs created. Therefore, the number of checkpoints taken by each process is 4n, i.e., 1 SIC + 3 RCs.

=> Checkpointing Time = 4n x 0.03 seconds.

(ii) Algorithm Rollback Time:

We assume the maximum of two faults.

=> Algorithm Rollback Time = 2 x 0.1 seconds.

(iii) Rollback Distance:

This is made up of the sum of the error latency, and the distance between the fault and the rollback checkpoint. Assume a maximum error latency, i.e., 0.9 seconds. Furthermore, assume that both faults occur just prior to the taking of a SIC, which causes the maximum rollback distance, e.g., to the prior SIC. Therefore, the distance between the fault and the rollback checkpoint is the inter-checkpoint distance, i.e., the total execution time divided by the number of SICs taken.

=> Rollback Distance = (2 x (100/n) + 2 x 0.9) seconds.

We see that..

i. $(4n \times 0.03) + (2 \times 0.9) + (2 \times 0.1) + (2 \times 100/n) <= 20$

ii. $0.12n + 2 + 200/n <= 20$

iii. $0.12n - 18 + 200/n <= 0$

iv. $0.12n^2 - 18n + 200 <= 0$

Solving for this quadratic equation, we get.. $n = [\ 12.08\ ..\ 137.89\ ]$.

Therefore, the programmer can indeed make use of the RRK, and, by creating anywhere from 13 to 137 SICs on each process, he is assurred of a fault tolerance overhead of no more than 20 percent. If the rollback distance is not crucial to the application (it is in the real-time environment) then the particular rate of SIC checkpointing chosen depends on whether the programmer is more interested in throughput or uniformity of performance. The lower the number of SICs, the lower the average time taken per run, assuming faults don't often occur, and the higher the throughput. The higher the number of SICs, the smaller the rollback distance upon the occurrence of a fault, and the more uniform the performance of the system, with or without faults.

Based on our experience with this experimental project, we wish to state the following for the benefit of other experimentalists: The proper selection of a distributed application program is important, especially when it is used as a means for the evaluation of rollback and recovery algorithms. The limitations and flexibilities of the communication suk :ystem play an important role. The error latency is one of the important factors that will determine the practical usefulness of a rollback and recovery scheme.

Chapter 5

Towards the Design of a Real-Time RRK

In the preceding chapters, we outlined the design of an RRK, based on a certain category of non-real-time rollback and recovery algorithms. In this chapter, we examine the unique aspects of the real-time environment which must be addressed in the design of a real-time RRK. Since, as we have seen, the design of the RRK depends on the nature of the rollback and recovery algorithms used, we must restrict our design to a particular type of algorithm, or model. In this thesis, we introduce the "Coexistence Model" for real-time fault tolerance.

The chapter begins with a look at the properties of real-time applications. Then, we will discuss the difficulties of designing reliable software for this environment and look at some approaches to real-time fault tolerance. After which, we will outline the proposed coexistence model and see how it may affect the design of the RRK.

5.1 Properties of Real-Time Applications

By "real-time" we mean those applications which deal with ongoing physical processes of the real, or natural, world. In the world of computers, time is parceled in the currency of CPU seconds; we conveniently allocate it, divide it, share it, stop it, even roll it back. When the interruptable digitized time of the computer world must interface with the non-interruptable continuous time of the real world, some compromises must be made, we must adapt the computers to function in real-time.

In this thesis, we deal with closed-loop process control systems of the hard real-time environment. The term "closed-loop" refers to a system, as seen in Fig. 5.1, with an unbroken flow of information between the controlled object, i.e., the physical process, and the controlling system, e.g., the Real-Time DCS, or RTDCS. As we see in the figure, some input from a human operator may also be received by the controlling system. The controlling system monitors the behavior of the controlled object through input from sensors such as thermocouples, optical scanners and contact probes, and effects physical changes on the controlled object through actuators like heating elements, servo-motors and valves. Applications where the input from the sensors or the output to the actuators must be carried out according to strict real-time limits are referred to as "hard real-time" applications or tasks. Hard real-time tasks are often safety-critical tasks, e.g., avionic control.

For the sake of clarity, we may refer to the real-time application, i.e., the controlled object, as the "physical process(es)", and the controlling computer system as the "software process(es)". It should be understood that the controlling system is only the computer itself, any

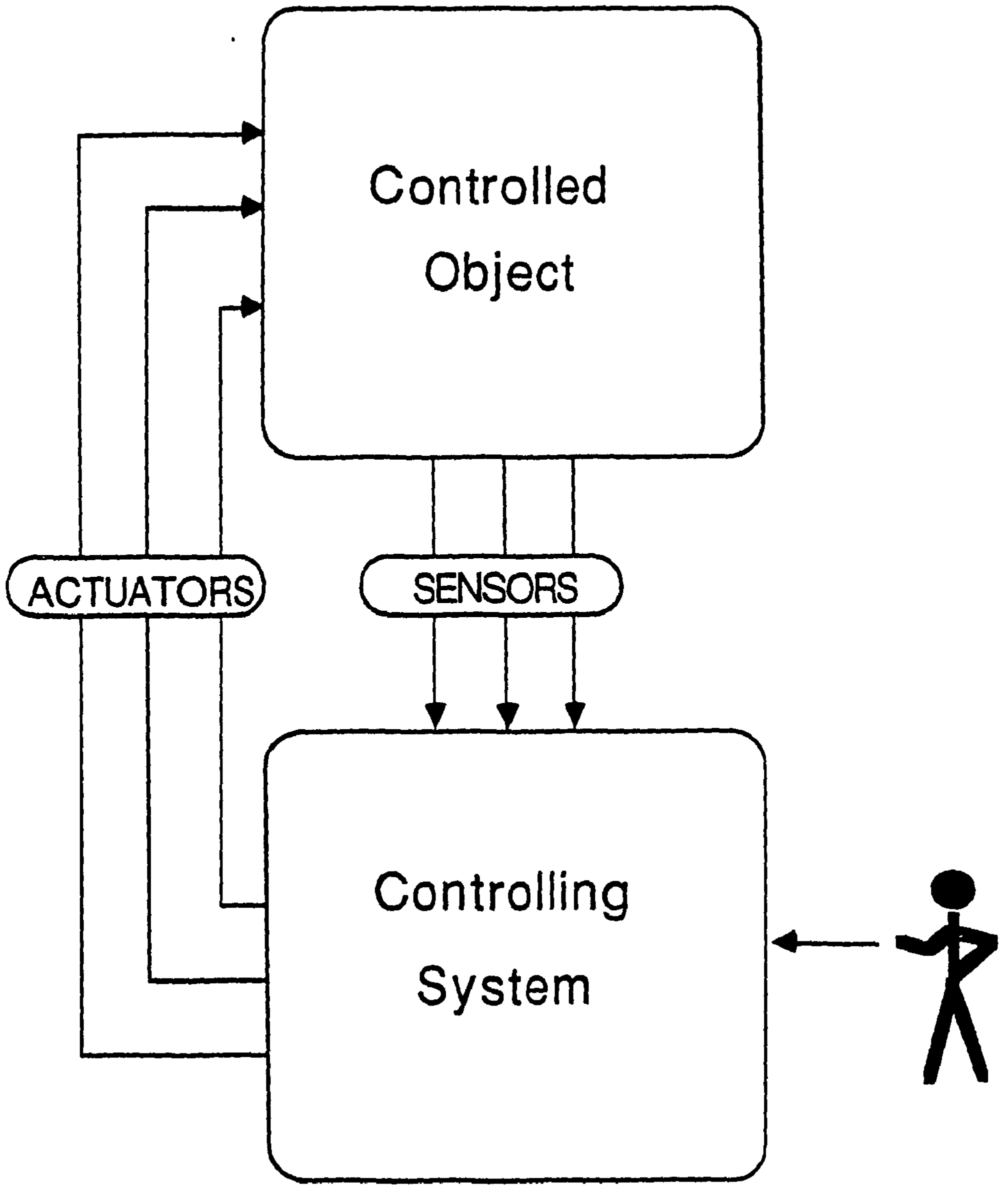


Figure 5.1 A closed-loop process control system

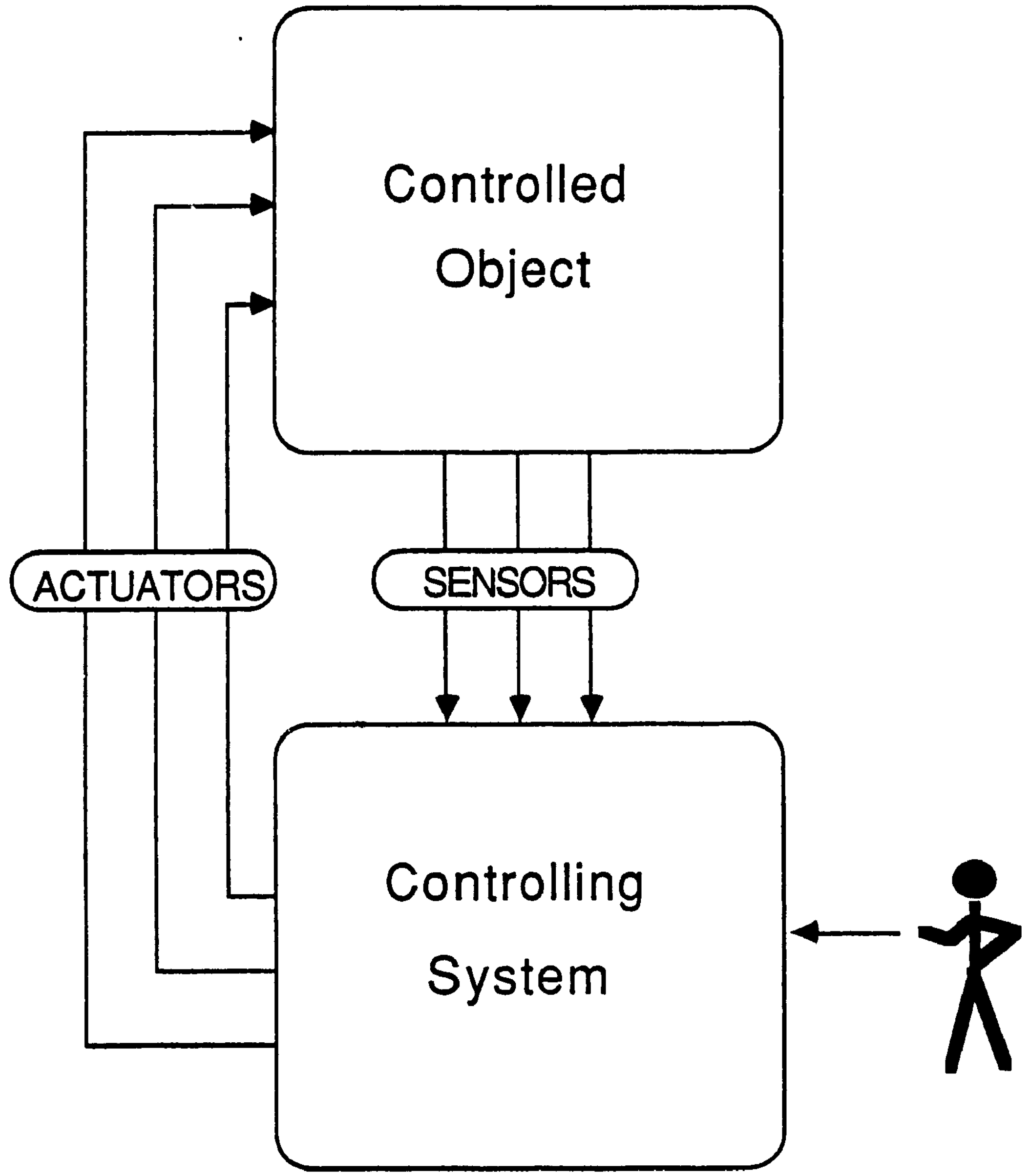


Figure 5.1 A closed-loop process control system

mechanical apparatus which interacts with the computer is viewed as part of the controlled object.

In Chapter 2, we saw how the application program and the RRK could be viewed as separate modules in the non-real-time environment. This enabled the design of a more general RRK which could be used with different applications. Fault intolerance, in the application program, and fault tolerance, in the RRK, could be handled separately. In real-time, the time constraint imposed on the computer system makes it more difficult for them to be treated as separate concerns. Time taken by the application program and the fault tolerant software must both be accounted for in order to meet deadline requirements. To facilitate the design procedure, the application program and the RRK should be designed concurrently rather than in sequence. To do that, we must first understand the nature, or properties, of the real-time physical processes in order to see how a RTDCS can be used to control it.

Obviously, the most important property of real-time systems is their timing constraint, some examples of which are borrowed from [19] and shown in Table 5.1; but what other properties do real-time applications have?

A common property of real-time applications is that the physical processes are made up of several, geographically or physically separate, related components. For example, in avionic control, the global task of flying the plane is made up of several separate components such as engine control, wing control, control of landing gear, and cockpit data display. Each separate site or component is, in turn, composed of one or more physical processes which may or may not be directly related to each other [9].

Table 5.1

EXAMPLES OF TYPICAL REAL-TIME CONSTRAINTS

Application	Required Response Times	Time Intervals Between Sensor Inputs
Radar Scanning	0.3 - 0.6 msec.	0.3 - 0.6 msec.
Jet Engine Test Bed	-	5 msec.
Missile Impact Prediction	100 msec.	50 msec.
Scientific Data Collection	up to 1 sec.	-
Production Control System	1 - 50 sec.	10 sec. upwards

For example, in the wing of the plane, there are such tasks as the movement of the control surfaces of the wing as well as the monitoring of the wing for excessive stress and the checking of the level of the fuel. With respect to the controlling system, the RTDCS is made up of a network of interconnected computers (processors). Each computer is called a node. On each node there can be one or more software processes. They are the controlling processes. For the closed-loop system, the simplest case, with respect to realizing processor schedules which guarantee time deadlines, would be to run each software process on its own dedicated node. Unfortunately, cost, weight, and space limitations may make it impracticable to support many processors at each physically separate site. More often, a single processor might have to handle the execution of several software processes, each controlling a separate physical process. For example, a single microprocessor may be required to handle the tasks to be carried out in the wing of the airplane, described above.

The behavior of the physical processes cited above can be summarized in terms of states and events. The state of a physical process usually changes in a regular and predictable manner, due to the inertia of the physical process and its inherent laws of nature. These state changes may proceed in one of two ways: periodic (cyclic change), and aperiodic (start-to-finish smooth-curve change), as shown in Fig. 5.2a and Fig. 5.2b respectively. Examples of physical processes which exhibit cyclic change are disk memory access, radar scanning, and heart-lung machine control. Smooth-curve change is seen in such physical processes as chemical reactions (e.g., temperature or pressure), and missile guidance (e.g., altitude or velocity).

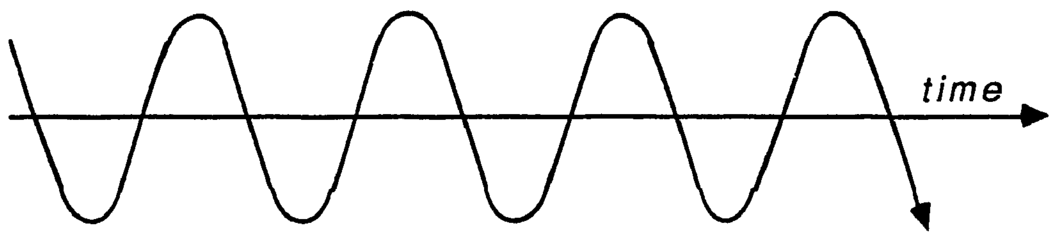


Figure 5.2a Predictable cyclic change of state.

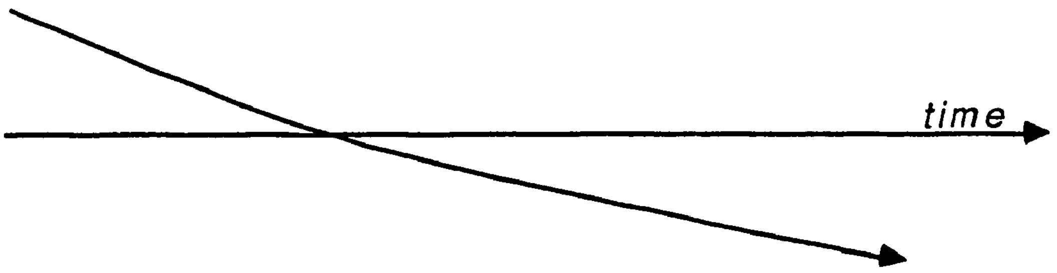


Figure 5.2b Predictable smooth-curve change of state.

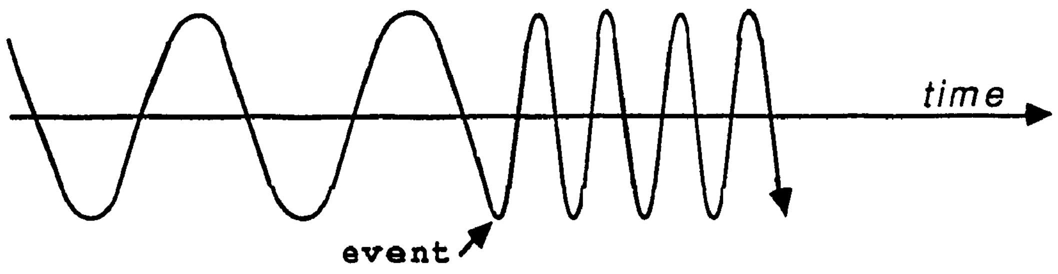


Figure 5.2c Effect of an event on cyclic behavior.

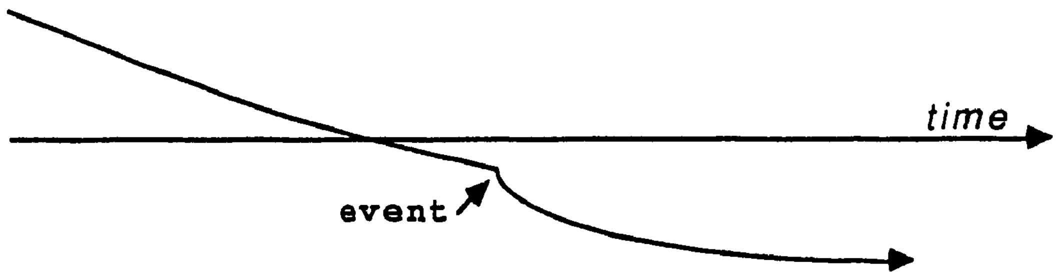


Figure 5.2d Effect of an event on smooth-curve behavior.

The events can also be classified into two categories: predictable and non-predictable, or sporadic. Events are predictable if we know when they will occur. For example, the ejection of a stage of a rocket. As for sporadic events, we do not know when they will occur or, even if they will occur. Examples of sporadic events are: a crack formation in the cooling module of a nuclear power plant which causes coolant to be lost and a critical temperature to be reached; lightning strikes a jet engine and causes it to fail; a worker, wanting to go for a coffee, pushes the 'off' button on his automated punch press.

The occurrence of an event may alter the nature of the process' state behavior (state transition), as are illustrated in Fig. 5.2c and Fig. 5.2d. For example, in the nuclear power plant, the loss of a functional cooling module may cause the remaining cooling modules to work harder in order to keep the core temperature safe. In this case, the frequency of the engine cycles of these cooling modules may change in the manner depicted in Fig. 5.2c.

Before we see how the behavior of the physical processes, outlined above, is handled by the coexistence model, we will first look at some of the existing approaches to the design of reliable real-time software.

5.2 Design of Reliable Real-Time Software

Some researchers are introducing more formalism into the design of real-time programs in order to create fault-free software. ADA supports such modern language concepts as (i) data abstraction, (ii) modularity, and (iii) machine independence, and introduces the "rendezvous" concept which handles both the synchronization of and communication between parallel software processes [13]. Wirth [26] suggested using separate processors for each software process of a real-time system in order to respect time deadlines. Mok [20] uses a type of process-based model, the "graph-based model", as a high-level scheduling tool in an environment where a processor must handle more than one software process. Zave [28] uses a more general view of the real-time system where both the physical processes and the software processes are modeled as interacting "asynchronous" entities. In her model, human beings are considered as part of the system. Faulk and Parnas [9] describe a two-level approach to synchronization in hard real-time by separating the low-level concerns, the "extended computer", from the high-level concerns, the "state transition event" (STE) variables. Considering the complexity of the task of designing fault-free real-time software, absolute success is virtually impossible no matter what the approach, and a robust fault tolerance mechanism must be added to real-time software systems.

In Chapter 2, we saw that the non-real-time RRK was required to reproduce consistent cause-effect relationships between the distributed send and receive events, upon detection of a fault. Although the overhead due to the RRK was considered, and efforts were definitely made to minimize it, the criteria for determining whether the RRK performed quickly enough were not set by any controlled object. In hard real-time, the fault tolerance mechanism must perform its duties within strict time limits determined by laws of nature rather than man.

A common approach to real-time fault tolerance is N-version programming [7]. Its desirability stems from its uniform time of computation, whether faults occur or not, but, as was discussed in Chapter 1, this approach is hindered by the difficulty in designing unique versions of software. Furthermore, some real-time applications, e.g., the on-board CPUs of missiles, may not be able to support the extra hardware, needed in N-version programming, because of restrictions on space and weight.

If rollback and recovery is the chosen means of fault tolerance, the system must be able to detect a fault and reexecute the computation before a response is required by the controlled object. An erroneous output sent to the environment cannot be rolled back. An elegant solution to these problems is supplied by Anderson and Knight in [2]. Their approach is based on a variant of Randell's "conversation" scheme [22].

Randell proposed that communicating parallel processes be grouped into conversations in order to synchronize the creation and the discarding of checkpoints. Processes involved in a conversation may communicate freely among themselves, but cannot communicate with other processes in the system. Processes may enter a conversation at different times, but must all exit at the same time. The information regarding which processes may be permitted to participate in a given conversation can be defined at a pre-run time, so that all the processes may have the ability to accept or reject any message, based on the sender's ID. Upon entering a conversation each process takes a checkpoint. Before exiting, each process performs an acceptance test on the results of its computation. Only if all the processes pass their respective acceptance tests are they allowed to discard their checkpoints and exit the conversation. If a fault is detected in any of the processes belonging to a conversation, they are all forced to roll back their computations to the beginning of the conversation.

By further restricting the interprocess communication, Anderson and Knight group processes into fault tolerant "exchanges". A process enters an exchange as it is initiated and exits when all the processes of the exchange, including itself, terminate. Therefore a process, throughout its execution, participates in a single exchange and cannot communicate with processes belonging to any other exchanges. This differs from the conversation model, where processes may participate in more than one conversation, i.e., either in sequential conversations or in nested conversations. Anderson and Knight argue that, although conversations are more flexible, they would require more synchronization points on each of the participating processes, and that the basically iterative nature of real-time systems can be easily modeled through exchanges if one views an iteration as being equivalent to a process of an exchange.

In both the conversation and the exchange models, the fault tolerance is based on the "recovery block" model [14], discussed in Chapter 1. In order for it to be possible for the processes to meet their time deadlines, the alternate software which is used upon rollback must be of a simple enough design so as to be able to run in a fraction of the time taken by the primary version. Due to this simple design, the alternates can only offer a degraded service. A trivial example of an alternate software is the so called "skip-frame" strategy, where the results of the previous iteration are used for the iteration in which the error occurred. An error history is maintained to determine the severity of the fault. Basically, if a fault appears in consecutive iterations the primary module is swapped for a stand-by primary version.

One should note that the design of the real-time recovery block need not be limited to the primary module and one alternate. Two, or more, alternates may be incorporated into the design as long as there is sufficient time, between the start of the recovery block and the time deadline, for the primary module and all the alternates to be executed.

Also in Anderson's and Knight's paper, the distributed computations are modelled in the form of "Synchronization Graphs", an example of which is borrowed from their paper and shown in Fig. 5.3. These are acyclic directed graphs with two types of nodes: those representing processes, and those representing timing constraints. Each process node is connected to two timing nodes, by arcs. A process must be initiated at, or after, the time indicated by the earlier of the two timing nodes, to which it is connected, and terminate at, or before, the time indicated by the latter of the two timing nodes. By modelling the system in this way, we see that a single timing node can serve as a common point of initiations and terminations of several processes, which greatly simplifies the logistics of processor scheduling.

A limitation of Anderson's and Knight's model, which is shared by virtually all real-time software design methodologies, lies in the treatment of sporadic events. To explain this, we must digress. Regarding the properties of real-time applications, we recall that their behavior is made up of predictable changes of state along with the occurrence of predictable and non-predictable, or sporadic, events. Typically, real-time software systems handle the predictable behavior through the use of iterative processes. The period of the iterations depends on the nature of the physical processes' behavior as well as the priority of the tasks. For example, in Fig. 5.4a, we see that, for the controlling system to keep pace with the controlled object, the period

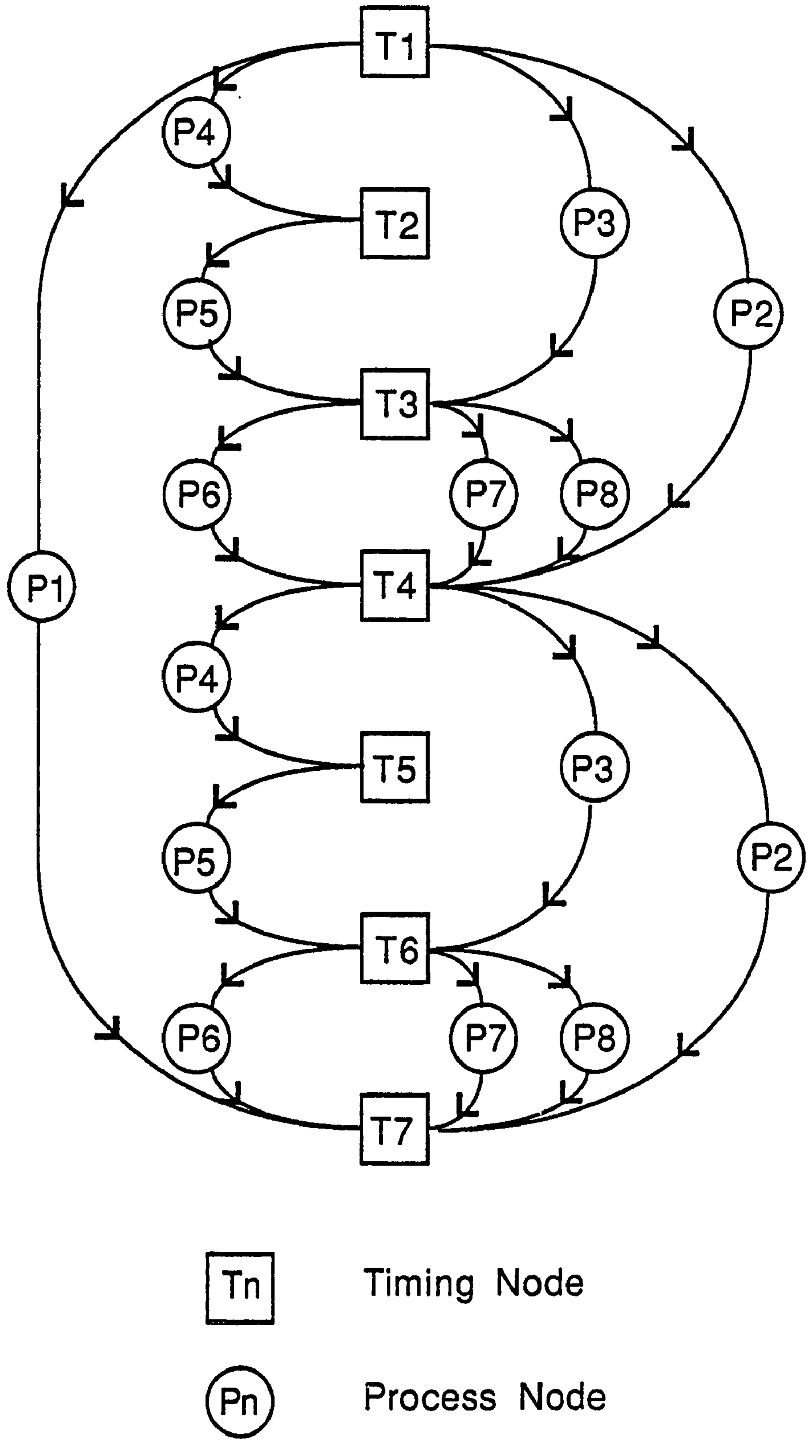


Figure 5.3 An example of a synchronization graph.

of a cyclic physical process, Tc, determines the period of the software process, Ts, i.e., Ts = F(Tc), where F is a function determined by the system designer. In Fig. 5.4b a smooth-curve change of state is shown. In this case the period of the software process would depend on the critical nature of the task as well as the rate of change of the curve, e.g., if the task is of a low priority or the rate of change is slow, frequent interaction may not be required. In avionic control, for example, the level of fuel in the wing may be seen as a smooth curve state change, which changes quite slowly. To monitor this change, a microprocessor may check the fuel sensor at a rate which is determined by the rate of fuel consumption, e.g., maybe once every 2 minutes.

In the literature, the sporadic events are either handled through periodic monitoring of sensors, or they are assumed to be of a lower priority and are handled on a time-left-over basis [20]. The first approach, periodic monitoring, is at best inefficient, since a process is allocated CPU time even if an event never occurs, and may even prove to be infeasible in an environment where several sporadic events must be accounted for in a system with limited processing power. Either approach, periodic monitoring or handling on a time-left-over basis, may not be quick to respond to the occurrence of a critical sporadic event. Referring to the same avionic control example, if there is a sudden dramatic loss of fuel, due to a ruptured wing structure for instance, it may go undetected for up to two minutes!

5.3 Coexistence Approach to Real-Time Fault Tolerance

A more general approach to real-time fault tolerance is needed to handle sporadic events, especially in an environment with strict

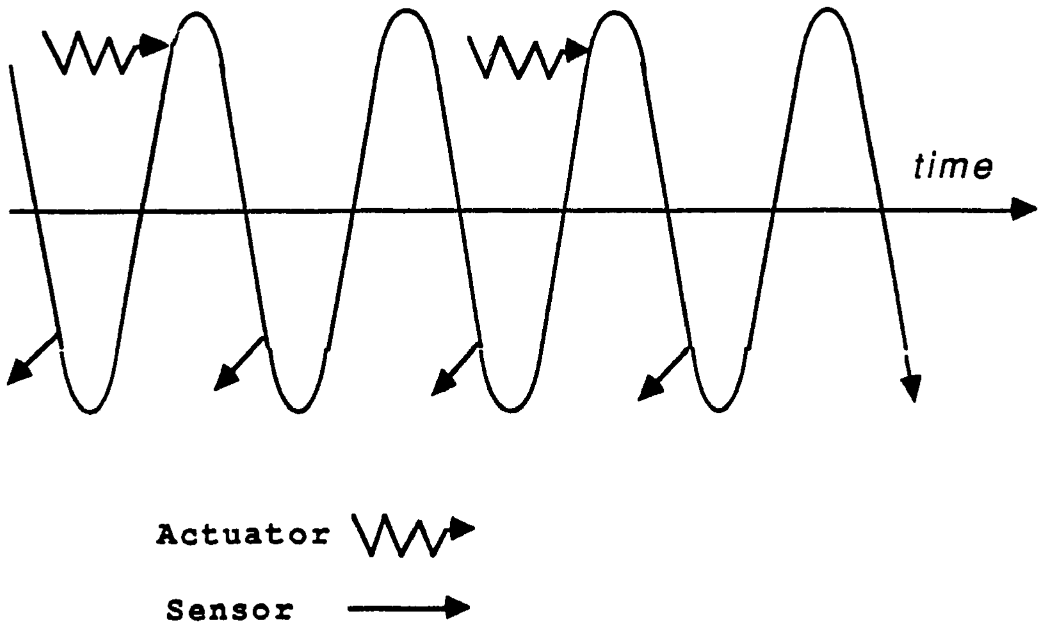


Figure 5.4a Control of a cyclic physical process..

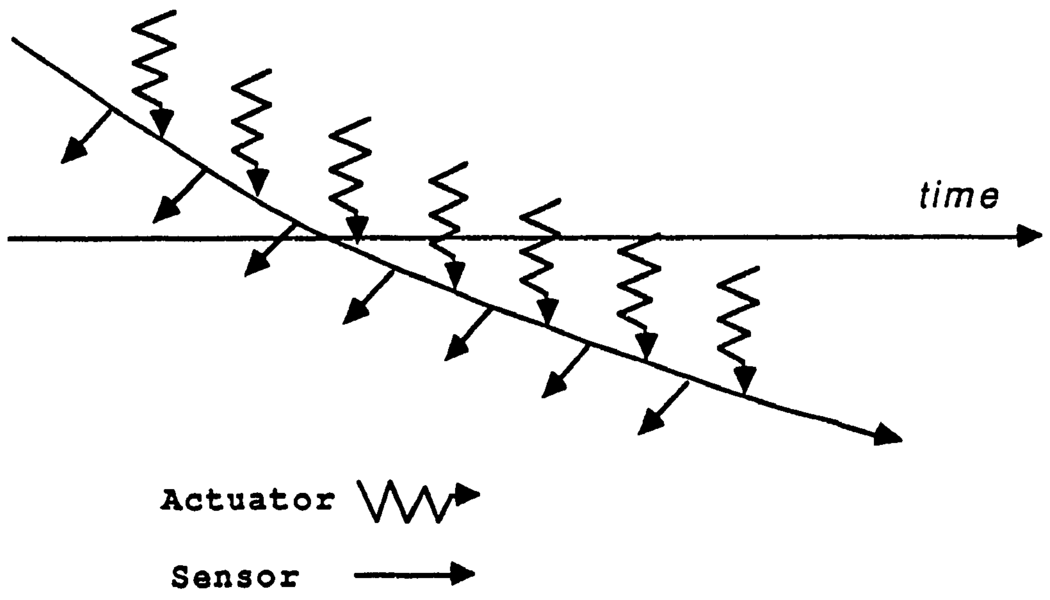


Figure 5.4b Control of a smooth-curve physical process..

limitations on the allowable space and weight of the RTDCS hardware. We introduce the coexistence model for real-time fault tolerance. The objective of this model is to handle the sporadic events in a way that: (i) is more efficient than periodic monitoring, i.e., the amount of special action needed to handle sporadic events should be proportional to the frequency of their occurrence; (ii) is more responsive than either periodic monitoring or handling on a time-left-over basis, i.e., the response to sporadic events, especially critical ones, should be almost instantaneous.

In the coexistence model, all predictable physical process behaviors are handled through Anderson's exchange/synchronization graph approach, while the handling of sporadic events is facilitated through the use of aperiodic interrupt-handlers and a rollback and recovery scheme which is similar to the RLV algorithm described earlier. The particulars of Anderson's approach have already been briefly described. In this section, we will first outline the approach to the control of sporadic events. Then, we will describe how the two approaches can work together, i.e., coexist, to form a more comprehensive approach to real-time software fault tolerance.

The sporadic events may or may not be critical events. The more critical the event the more severe the effect on the changing state behaviors of the physical process. The sporadic events are handled by aperiodic processes. Even though the occurrence of the sporadic events is unpredictable, the manner in which they are handled by these aperiodic processes can be well thought-out and known a priori. Although part of the duties of these processes may be to produce some responses to be output directly to the environment, their main function is to alter the arrangement of the periodic processes and, thereby, affect the controlled object. The aperiodic processes must properly diagnose the sporadic event and then update the schedule of the periodic processes that follow. We can, therefore, view the aperiodic processes as a state of flux in the controlling system, which acts as a sort of coarse control of system behavior, while the periodic processes carry out the ongoing fine tuning of this behavior. Since these aperiodic processes, like the periodic processes used for the predictable behavior, should produce results within a certain time and must have a quick means of detecting errors, the software is arranged in recovery blocks. The complete aperiodic action could be made up of several recovery blocks in the distributed system, as is shown in Fig. 5.5. The determination of the size of these recovery blocks will be addressed shortly.

The respective tasks of the distributed aperiodic processes are initiated, i.e., the nodes become aware of the sporadic event, through one of the three following means: (i) critical timer alarm; (ii) asynchronous stimulus from the environment; (iii) asynchronous message.

The first option, critical timer alarm, occurs when a periodic routine has failed, i.e., passed its deadline without producing a satisfactory result. In this way, the coexistence model can offer an extra layer of fault tolerance over the model proposed by Anderson and Knight. They point out that often the system can tolerate belated or erroneous outputs for a cycle, since the inertia of the physical equipement may absorb the temporary failure of the software. However, in their model, chronic failure could be disasterous. In the

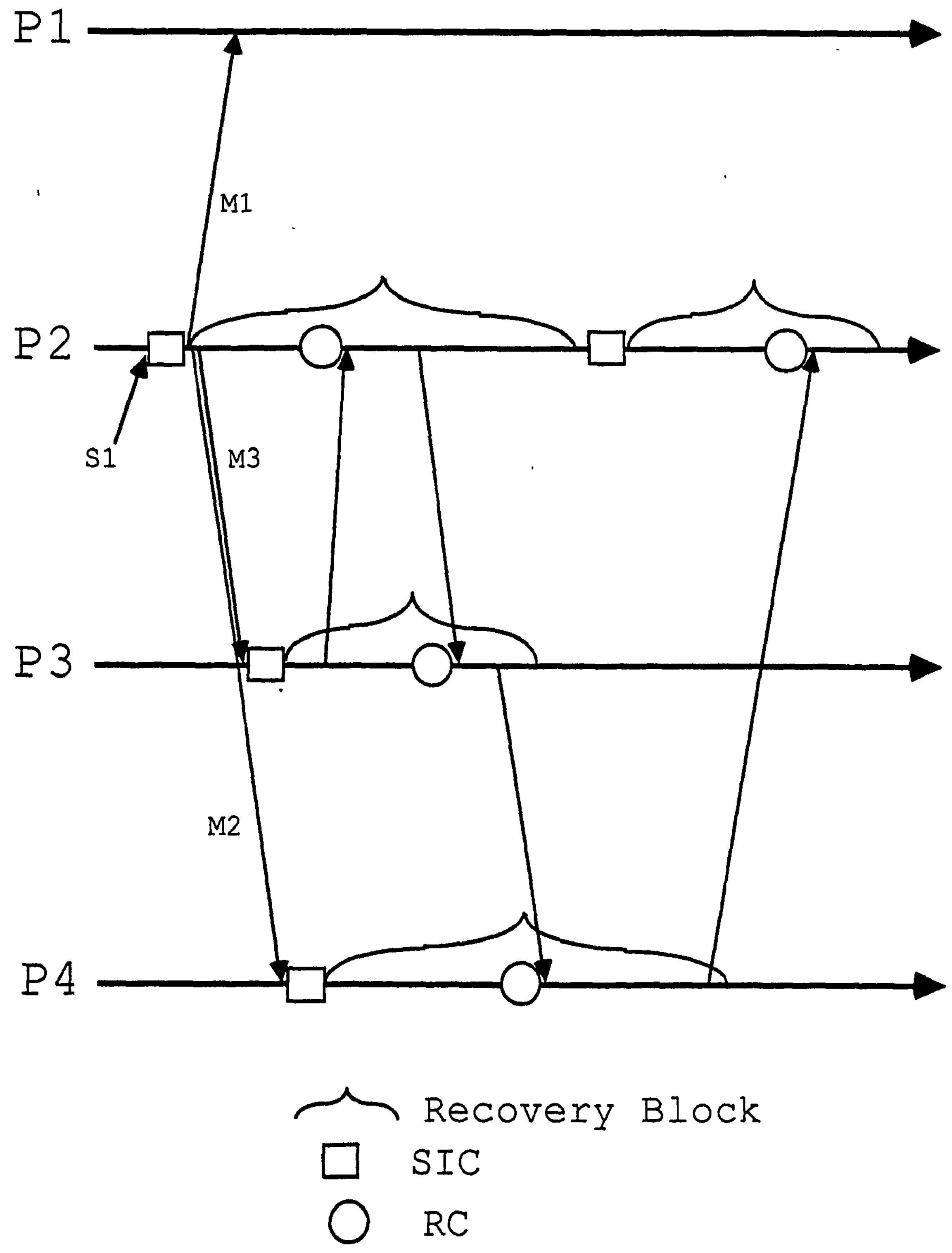


Figure 5.5 Aperiodic processes of the coexistence model.

coexistence model, if the task is critical or if the malfunction persists, the inclusion of an aperiodic routine could act as a "fail-safe" mechanism.

The second option, asynchronous stimulus, is the more common type of sporadic event described throughout this chapter. It occurs when a sensor is activated due to a predefined threshold being exceeded by the physical process.

The third option, asynchronous message, happens when a process learns of the sporadic event from another process in the system. To enable a speedy response to these messages, their implementation may require some sort of out-of-bound message passing, perhaps with special channels dedicated to asynchronous messages. Also, to more easily inform the receiving process of the nature of the sporadic event, these asynchronous messages may be "typed" according to event.

In the RTDCS, the first process initiated due to the sporadic event is labelled the "Response Initiator" or RI. The RI becomes initiated through either a critical timer alarm or an asynchronous stimulus. Whether the identity of the RI, for a particular sporadic event, is predetermined or not, is not critical to this design. The RI is responsible for at least the first stage of event diagnosis, that is, the broad classification of the event to one type or another. Subsequently, the RI may require the help of other processes to more completely diagnose the event and come up with a schedule for the periodic processes. As we see in Fig. 5.5, processor P2, the RI, becomes aware of the sporadic event through an unexpected input from a sensor via stimulus S1. The eventual responses needed to control the controlled object may be produced by the RI itself or another participating process. The first action taken by the RI is to send an asynchronous message on all its outgoing channels to inform other nodes about the particular ̣:oradic event which has occurred, so that they may begin their respective duties, if any. Referring again to Fig. 5.5, we see that process P2 informs the other nodes in the system by sending out the asynchronous messages, M1, M2 and M3. In this case, processor P1 has no duties to perform for such an event so it carries on with whatever it was doing, that is, the execution of periodic processes. Processors P3 and P4, on the other hand, respond by initiating their own aperiodic processes.

In Table 5.1, we saw that the I/O between the controlling system and the controlled object often occurs quite frequently, e.g., a few times per second. Not surprisingly, the periodic processes designed to handle this predictable behavior usually take a fraction of a second per iteration. On the other hand, the aperiodic processes may have to carry out detailed event diagnosis and processor scheduling, requiring extensive computation which may be infeasible to roll back in its entirety. Furthermore, since the processes are not iterative, trivial alternative software modules, such as skip-frame, can not be as easily designed. For these reasons, the exchange model proposed by Anderson and Knight would not be suitable as a means for fault tolerance of the aperiodic processes. A more appropriate model would enable a minimization of the rollback distance, perhaps through dependency tracking and the creation of timely RCs, i.e., a dynamic checkpointing algorithm like RLV.

In the coexistence model, each aperiodic process first takes a checkpoint, an SIC, before beginning their respective tasks. These SICs mark the beginning of their first recovery blocks. To handle the logistics of the creation and discarding of the SICs and the subsequent RCs, we refer to Kim's "PTC/LCN" scheme [16]. This algorithm, a dynamic checkpointing algorithm, is very similar to the RLV algorithm in terms of its creation of RCs and recovery lines. Unlike RLV, however, Kim's algorithm is based on the recovery block mechanism, which is seen as being more appropriate for the real-time environment since it can place a bound on the execution time of, and the error detection for, a process. Kim, himself, does not view his algorithm in the real-time context. More detailed analysis of the suitability of the PTC/LCN scheme should be carried out in this sense. In this algorithm, RCs may be formed when a process receives a message within its recovery block. Before exiting its recovery block, and discarding its SIC, each process must pass an acceptance test. A SIC can be discarded if the recovery block acceptance test is passed and either no RCs were established within the recovery block or all the RCs established have been discarded by their owners. A more complete discussion of the PTC/LCN algorithm can be found in [16].

The size of the recovery blocks depends on two things: (i) whether a response is directly output to the environment (in which case it would force the execution of an acceptance test just prior to the response, since the response cannot be rolled back); and (ii) the amount of "slack time" between the deadline for the completion of the process and the execution time of the primary algorithm. In the latter case, the sum of the executions of the alternate modules should never exceed the slack time.

The periodic processes of Anderson's scheme must coexist with the aperiodic processes described above, whether faults occur or not. We recall, from section 5.1, that the occurrence of events may cause a change in the otherwise predictable behavior of the controlled object. Accordingly, we can classify the aperiodic tasks into groups, according to the degree to which they affect the behavior of the controlling system. For example, low priority, intermediate priority, and high priority tasks. The low priority tasks are those that either represent sporadic events which are not too important, or there is no action that can be taken to remedy the situation and the tasks carried out are merely for book-keeping purposes, i.e., to record the occurrence of the event. In either case, the tasks can be handled on a time-left-over basis and need not interrupt, nor affect, the execution of the periodic processes. The intermediate priority tasks are those which may alter the behavior of the system, but not at the expense of turning off the ongoing periodic tasks. These tasks are of a higher priority than the primary algorithm of the periodic tasks, that is, they either interrupt it or prevent it from beginning, but are of a lower priority than the alternate algorithms and, therefore, cannot affect them. The high priority tasks are those which will so radically affect the behavior of the system that further periodic execution is both wasteful and redundant. These tasks interrupt and/or prevent the execution of the periodic routines. Finally, the priority of a given aperiodic routine may vary, that is, it may be regarded as a high priority task in relation to some unimportant periodic routines, but only as intermediate priority when compared to more crucial periodic tasks.

To achieve fault tolerance, we view the two types of processes, periodic and aperiodic, as separate atomic actions. To enforce this view, the aperiodic processes should be designed so that they do not directly communicate with the periodic processes (aperiodic processes may set the initial state of newly arranged periodic processes, however). Although this may at first appear to be a rather severe restriction, it is believed that it is quite naturally realized, since the functions of the two types of processes are quite different. If these restrictions can be enforced, then, as Campbell et.al. point out [6], "If a fault, resulting error propagation, and subsequent successful error recovery all occur within a single atomic action they will not affect any other system activities". Therefore, in the coexistence model the fault tolerance of one type of process, e.g., periodic, can be treated completely separately from the other. To illustrate this we refer to Fig. 5.6. In this figure, we see that a given processor, during a given time frame, may have an ongoing aperiodic routine which has interleaved within it one or more periodic routines. In this example, should any of the periodic routines roll back due to the detection of a fault they would have no effect on the aperiodic routine. If, on the other hand, the aperiodic routine should roll back, its recomputation may alter the behavior of future periodic routines, but it cannot force a periodic routine to roll back.

With the coexistence model, a host of aperiodic routines can be defined without affecting the performance of the system, since a given aperiodic routine is only activated upon the occurrence of a particular sporadic event. Also, depending on the critical nature of the event, an aperiodic routine can respond right away.

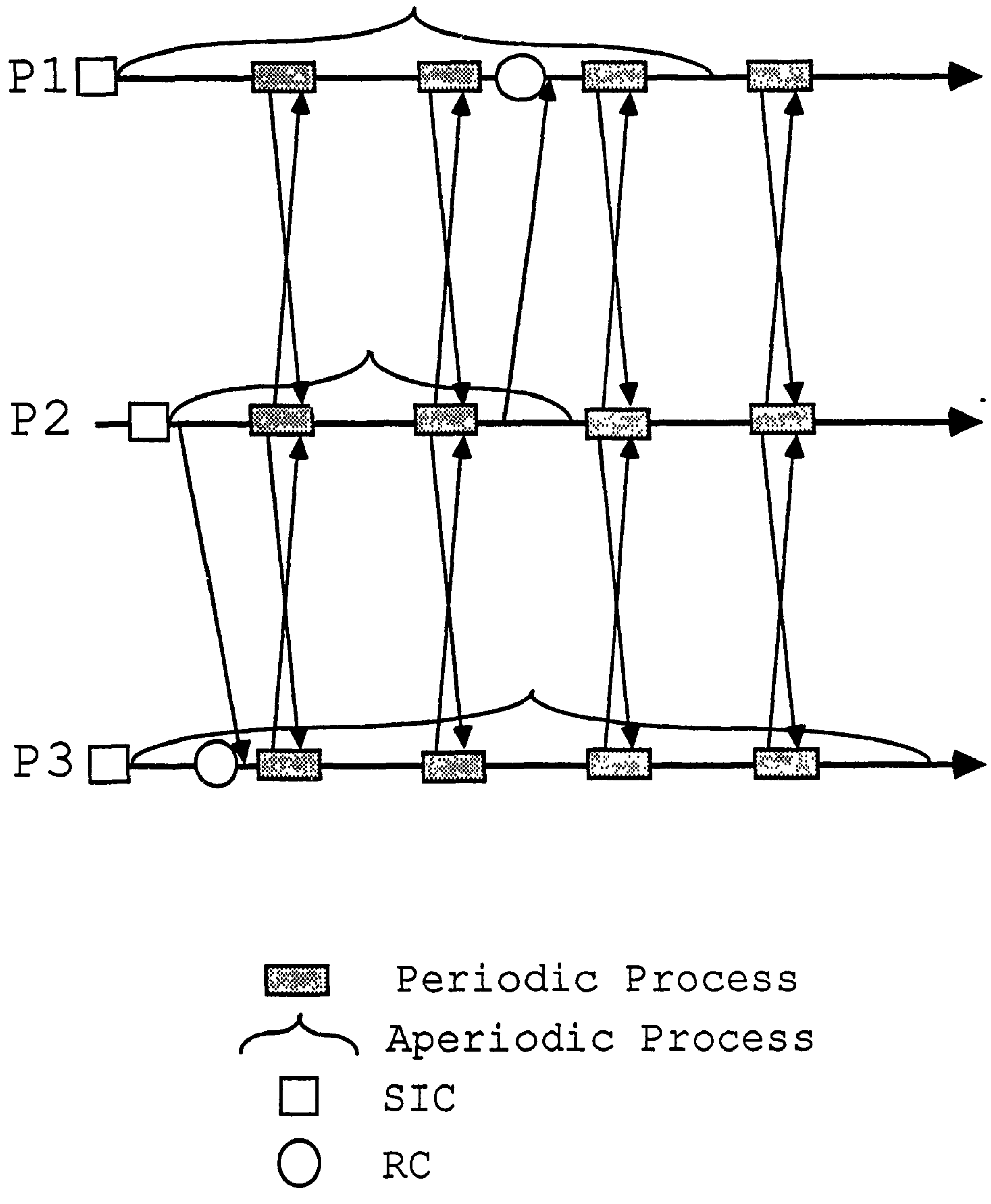


Figure 5.6 Interleaved processes in the coexistence model.

5.4 Considerations for a Real-Time RRK

The following points should be considered in the design of a real-time RRK:

(i) In the real-time process control environment, each node in the RTDCS may have to support several software processes. Although these processes may or may not be directly related [9], they are all indirectly linked by their common controlled object. Therefore, unlike the RRK described in Chapter 2, the real-time RRK must support more than one application process at the same time.

(ii) The Real-Time RRK (RTRRK) should be divided into two separate modules, one which handles the rollback and recovery for the periodic jobs and one which handles the rollback and recovery for the aperiodic jobs. This coincides with their treatement as separate entities. Obviously, the RTRRK could not be appended to an individual process, even for the purpose of performance analysis, as we had done for the non-real-time RRK, in this thesis. It would have to exist as part of the OS software.

(iii) In Chapter 2, the design of the RRK, the application program, and the OS software were treated as separate concerns. In the real-time domain, these three softwares must be designed together. It is not feasible to allow the application programmer to design his program as he wishes, without paying attention to the design of the other softwares.

(iv) On each node in the system, there must be a "watch-dog" timer which keeps track of the passage of real time. This timer would be responsible for ensuring that each process respects its time limits. If a process should pass a deadline the watch-dog timer will alert the appropriate RRK module.

In summary, we find that sporadic events seem to be handled ineffectively in most existing real-time software systems. An approach to real-time software fault tolerance has been described which appears to offer a more efficient and responsive means of handling sporadic events. This approach, the coexistence model, also incorporates a means for allowing for more immediate responses to the occurrence of the more critical events.

Chapter 6

Conclusions and Suggestions for Future Work

In this thesis, we examined some of the important issues in the practical implementation of rollback and recovery in the distributed computing environment. In particular, we were interested in a specific type of rollback and recovery algorithm, the dynamic checkpointing algorithms. These algorithms employ preplanned recovery, are non-intrusive checkpointing algorithms, and place few constraints on the behavior of the application program.

In Chapter 2, we discussed some of the aspects to be considered in the design of a Rollback and Recovery Kernel (RRK). This RRK was meant to provide a practical test-bed environment for the implementation and testing of two dynamic checkpointing algorithms, RLV and BCS. Also, an effort was made to design an RRK which would be easy to use by future application programmers. It is felt that the resulting design, made up of an application invariant part and an application dependent part, is one which allows for a relatively easy link-up with application programs while retaining a rollback and recovery software which can perform quite efficiently. To support this claim, future work should include the use of the RRK with other applications other than N-Life. Also, the RRK should be integrated with the OS software, as described in

Chapter 2.

While implementing the RRK, we had to choose an appropriate distributed test-bed application program. Since we could find little literature on the subject, we evolved some of our own criteria for the characteristics of a such an application program. Our own particular choice was a distributed version of J. H. Conway's "Game of Life", "N-Life", which was developed by us. Although N-Life was found to be a very flexible and easy-to-use testing and debugging tool, its synchronization properties (loosely synchronous) were concluded not to be ideal for our purposes, i.e., the performance measurement of dynamic checkpointing algorithms, which are essentially asynchronous. A worthy subject for future research would be a more rigorous examination of the differrent synchronization characteristics which may be exhibited by different classes of application programs, and the development of a totally asynchronous test-bed application program.

Once the RRK was implemented, we proceeded with taking measurements on the two algorithms mentioned above. Unfortunately, we discovered that the DCS architecture we had chosen harboured a bottle-neck in communication due to its use of a central file server. Because of this, we limited most of our measurements to a two-process model. When the bottle-neck is resolved, future work may include more elaborate measurement examples. No absolute conclusions could be reached on the algorithms, either individually or as a class, since the overhead they incur is based on such application dependent characteristics as rate of checkpointing, size of checkpoints, message passing behavior, and error latency. However, we did observe that the overhead due to checkpointing, at least with N-Life, was not much (often less than

10%). In most cases BCS caused less overhead in checkpointing, although the difference between the algorithms, in this regard, was negligible. With respect to the overhead due to rollback and recovery, RLV was shown to rollback a shorter distance than BCS in most cases, although, once again, this depended on the particular message passing behavior of the application. Since the amount of error latency was seen to cause a proportional increase in the rollback distance, it was concluded that, ultimately, the error latency in a particuler system may determine the feasibility of using rollback and recovery as a means of fault tolerance. Future research may include methods of speeding-up the actions of saving and loading checkpoints, and perhaps distinguishing between the actions taken for SICs vs. RCs. Since SICs are more or less predictable, some sort of economization may be implemented.

The real-time computing environment is one in which software fault tolerance is most crucial, due to the critical nature of the tasks. However, it is also an environment where the implementation of fault tolerance through rollback and recovery is very difficult, due mainly to the inherent timing constraints imposed by the controlled object. In Chapter 5, we discussed some of the characteristics of real-time applications and summarized some approaches to the design of reliable software for is environment. We also introduced a composite model for software fault tolerance, the "Coexistence Model". The impetus for the latter was, what was seen as, a weakness in the treatment of sporadic events by existing approaches.

However, several important factors remain to be addressed in order to realize a complete fault tolerant model. Firstly, the reason previous approaches have avoided such a dynamic software model is that periodic jobs are more predictable, in terms of their resource requirements, than aperiodic jobs. Consequently, a simpler scheduling scheme can be used if all the processes are periodic. Faulk and Parnas point out [9] that a more effective approach to real-time scheduling can be achieved by using both a pre-run-time and a run-time scheduler. The pre-run-time scheduler can map out the predictable periodic scheduling, which forms the bulk of hard real-time computation, while the run-time scheduler is only used upon the occurrence of unpredictable external events. They describe how, even for the run-time scheduler, the overhead can be further reduced by having the "worst-case" scheduling decisions made before run-time and saving the resulting schedules in tables. The computation of the run-time scheduler is, then, merely one of selecting a schedule from a list of alternatives. In the coexistence model, the task of run-time scheduling must be considered as part of the overhead due to the aperiodic routines.

Each time a process preempts the execution of another process a context switch must be made. If the number of processes running on a node in the system is large or transfer of control takes place often, the time overhead due to context switching as well as the memory space needed to save interrupted images may be too costly. Faulk and Parnas describe how a scheduling program can consider the information on the timing constraints and sequencing of the individual processes, and then output a single object code made up of the different tasks, which has embedded within it the schedule of task execution. Other researchers [8,21,25] describe the use of software interrupt-handlers, which can be used to handle the occurrence of asynchronous communications. The ongoing periodic routines could be defined within the main body of the program code, and the aperiodic routines defined in the interrupt handlers. Since such an approach would not require the use of separate processes for the aperiodic tasks it would also result in a decrease in the overhead due to context switching.

The limited utility of the acceptance test approach to fault detection must also be addressed. Jackson showed how the addition of other forms of fault detection to that of the acceptance test increased the number of faults detected [15]. Anderson also declared that the "positive acceptability checks", i.e., acceptance tests, should be used to supplement, not replace, the more rigorous specific malfunction, or "negative", checks. However, he does not go on to outline how the ongoing error detection routines, necessary for malfunction tests, can be incorporated into the exchange model. For his part, Kim describes a variant of the method described earlier. In this algorithm, PTC/LCN-2, the fault detection mechanism of a message-receiving process can accuse the sender of having exported faulty information, in the case where the sender's acceptance test failed to catch the fault. However, since the additional means of fault detection is again an acceptance test, it is doubtful whether there would be a significant increase in the number of faults detected. In order to realize a truly fault tolerant real-time system, some sort of ongoing error detection mechanism, one which need not be encapsulated within the recovery blocks, must be added. Undoubtedly, such an error detection mechanism must ensure a low error latency in order to be feasible in the real-time domain. For, as we saw in Chapter 4, an increase in error latency results in a proportional increase in the overhead due to computation reexecution.

Finally, the models used throughout the thesis, e.g., the Space

Time Models, are process-based models. In any field of research, the use of a single type of model, to represent a problem, may result in a rather restricted perspective of the task at hand. Future work should investigate the use of alternative modelling schemes, e.g., object-based models. Especially with respect to research involving real-time distributed computing, a more formal model must be developed.

References

1. M.W. Alford, J.P. Ansart, G. Hommel, L. Lamport, B. Liskov, G.P. Mullery, and F.B. Schneider, "Distributed Systems - Methods and tools for specification - An advanced course", Lecture notes in Computer Sci. No. 190, Springer-Verlag 1985.

2. T. Anderson and J.C. Knight, "A framework for software fault tolerance in real-time systems", IEEE Trans. Soft. Eng., SE-9, No. 3, 1983, pp. 355-364.

3. A. Avizienis, "Fault tolerant systems", IEEE Trans. Comput., Vol. C25, Dec. 1976, pp. 1304-1312.

4. E.R. Berlekamp, J.H. Conway, and R.K. Guy, "What is Life?", Winning Ways for your Mathematical Plays Vol. 2, Chapter 25, Academic Press 1982, pp. 817-850.

5. D. Briatico, A. Ciuffoletti, and L. Simoncini, "A distributed domino-effect free recovery algorithm", Proc. Symposium on Reliability in Distributed Software and Data Base systems, Oct. 1984, pp. 207-215.

6. R.H. Campbell, T. Anderson, and B. Randell, "Practical fault tolerant software for asynchronous systems", IFAC Safecomp, 1983, pp. 59-65.

7. L. Chen and A. Avizienis, "N-Version Programming: A fault tolerance approach to reliability of software operation", FTCS-8, June 1978, pp. 3-9.

8. A.N. Copper III and J.P. Kearns, "Real-time distributed control with asynchronous message reception", Real-Time Sys. Symp., 1985, pp. 67-75.

9. S.R. Faulk and D.L. Parnas, "On synchronization in hard-real-time systems", Comm. ACM, Vol. 31, No. 3, March 1988, pp. 274-287.

10. J.P. Fishburn, "Analysis of speedup in distributed algorithms", Ph.D Thesis, University of Wisconsin, Madison, May 1981.

11. M. Gardner, "The fantastic combinations of John Conway's new solitaire game 'Life'", Scientific American, Oct. 1970, pp. 120-123.

12. R.L. Glass, "Real-Time: the 'Lost World' of software debugging and testing", Real-Time Software, Ed. R.L. Glass, Prentice-Hall 1983, pp. 159-172.

13. M.E. Gordon and W.B. Robinson, "Using preliminary ADA in a process control application", Real-Time Software, Ed. R.L. Glass, Prentice-Hall 1983, pp. 159-172.

14. J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery", Proc. Conf. Operating Systems: Theoretical and Practical Aspects, IRIA, Apr. 23-25, 1974, pp. 177-193.

15. P.R. Jackson and B.A. White, "The application of fault tolerant techniques to a real time system", IFAC Safecomp, 1983, pp. 75-82.

16. K.H. Kim, J.H. You, and A. Aboulenaga, "A scheme for coordinated execution of independently designed recoverable distributed processes", Digest FTCS-16, 1986, pp. 130-135.

17. J.C. Knight, N.G. Leveson, and L.D. St. Jean, "A large scale experiment in N-version programming", FTCS-15, 1985, pp. 135-139.

18. J.C. Knight and N.G. Leveson, "An empirical study of failure probabilities in multi-version software", FTCS-16, 1986, pp. 165-170.

19. J. Martin, "The range of real-time systems", Real-Time Software, Ed. R.L. Glass, Prentice-Hall 1983, pp. 3-12.

20. A.K. Mok, "Fundamental design problems of distributed systems for the hard real-time environment", Ph.D Thesis, MIT,May 1983.

21. U. Petermann and A. Szalas, "A note on PCI: distributed processes communicating by interrupts", SIGPLAN Notices, Vol. 20, No. 3, 1985, pp. 37-46.

22. B. Randell, "System structure for software fault tolerance", IEEE Trans. Soft. Eng., SE-1, No. 2, June 1975, pp. 220-232.

23. K. Venkatesh, "Global States of Distributed Systems: Classification and applications", Ph.D Thesis, Concordia University, Montreal, Jan. 1988.

24. P.T. Ward and S.J. Mellor, "Structured development for real-time systems" Vol. 1, Introduction, Yourdon Press 1985.

25. H. Wettstein and G. Merbeth, "The concept of asynchronization", SIGOPS Review, 1980, pp. 50-70.

26. N. Wirth, "Toward a discipline of real-time programming", Real-Time Software, Ed. R.L. Glass, Prentice-Hall 1983, pp. 128-142.

27. W.G. Wood, "Recovery control of communicating processes in a distributed system", Reliable Computer Systems, Ed. S.K. Shrivastava, Springer-Verlag 1985, pp. 448-484.

28. P. Zave and W. Schell, "Salient features of an executable specification language and its environment", IEEE Trans. Soft. Eng., SE-12, No. 2, Feb. 1986, pp. 312-325.

Appendix A

Setjmp/Longjmp Design Restriction

The Setjmp/Longjmp constructs introduce a design restriction. Only one return pointer can exist at any one time. This has two ramifications in the use of these C-language constructs in a rollback and recovery kernel.

Firstly, since limiting the user to a solitary checkpoint would obviate the usefulness of a RRK, the application program is required to be of an iterative nature with, at most, one call to the Chckpnt function per cycle. As was explained in Chapter 2, this occurs at the beginning of an iteration. Since the multiple iterations are the result of a single segment of code, all the SICs created are the result of a single occurrence of "Chckpnt();" in the program code. This allows a lone return pointer to service many checkpoints.

Secondly, the reader may recall that, along with explicit Chckpnt calls, i.e., for the creation of SICs, there also exists implicit checkpoint creations in response to received messages, i.e., the creation of RCs. Since we only have the use of a single return pointer, a rollback to the latter type would result in a Longjmp to the unique return pointer location, already used for SICs, rather than to the place in the code where the Kread, which caused the creation of the RC, was actually executed. To remedy this, a "fall-through" mechanism was employed. In this mechanism, certain program variables, e.g., indices used in for-loops, while-loops etc., which indicate the extent of the computation at the time of checkpoint creation, are saved with the other checkpoint process state information. During rollback, these variables are reloaded and control is returned to the position in the code where the call to Chckpnt occurs, that is, at the top of an iteration. If the checkpoint in question is a RC, then the control would "fall-through", as it were, to a place within the code which corresponds to the value of the reloaded variables. This is ensured by the existing programming language logic.

The arrangement, however, is not ideal since, for example, the granularity of the "fall-through" mechanism is limited by the size of the innermost loop construct. A rollback to a RC causes the repetition of any instructions located between the Kread instruction and the top of the innermost loop construct in which it is situated. Future code updates should include a more general scheme with low-level access to the system's process state information.