Universitat Autònoma de Barcelona

Escola Tècnica Superior d'Enginyeria
Departament d'Arquitectura de Computadors
i Sistemes Operatius

RADIC II: a Fault Tolerant Architecture with Flexible Dynamic Redundancy

Master thesis submitted by Guna Alexander S. dos Santos in fulfillment of the requirements for the degree of Master per la Universitat Autònoma de Barcelona.

Bellaterra, July 2007

Dr. Dolores Isabel Rexachs del Rosario, from the Computer Architecture and Operating Systems Department at Universitat Autònoma de Barcelona,

HEREBY CERTIFY:

That the work entitled "**RADIC II: a Fault Tolerant Architecture with Flexible Dynamic Redundancy**" has been written under my supervision by Guna Alexander Silva dos Santos, and that it constitutes the experimental work carried out within the Postgraduate Program for Computer Architecture and Parallel Processing.

Bellaterra, July 10, 2007.

Dolores Isabel Rexachs del Rosario

To everyone who is reading this work.

Acknowledgments

Almost two years ago, I was just arriving here: a foreign guy in a foreign land facing a great challenge in his life. Today, I am here completing the middle step of my challenge, and I must to thanks a lot of people for that.

First, I would to thank God and the Spirituality, They did help me and guide me through this work, being present in the best and the worst moments.

Of course, I would like to thank my parents and family. Without them I would not be here, without them, I would not acquire the knowledge necessary to be here and without them, I would not be strong enough to facing this challenge.

I am very grateful to my great friend Eduardo Argollo, the man that made possible my dream came true. Thank you by all of the tips about the doctorate, research, etc. Thank you by the pleasant time playing “puein, puein” matches. Thanks for has presented me this new world.

I can consider myself as a very lucky person. I had two incredible persons advising me in this work. Indeed, this dissertation is result of their work too. I would to thank you Emilio by all knowledge and expertise. Lola, as I said before, you did overcome the advisor tasks, I would to thank you by your words of wisdom, care and smiles.

Special thanks to my friend Angelo, I am so grateful by your RADIC lessons, entertainment moments, kebab meetings and so. Thank you by has left me the future of your work. Other special thanks to my friend and housemate Genaro, our “guru”. Thanks for all tips about C programming, Linux concepts, ideas, etc. Thank you too by to be an excellent housemate with your peculiar sense of humor. To all my friends, the new ones, the old ones, here or in Brazil, I would to tell you as grateful I am because all of your support, through words of incentive, positive thought and help.

There are two persons who I am in debt: Dani Ruiz and Jordi Valls. Thank you guys by all technical support, solving all my problems with the cluster, BLCR and so.

“The last ones will be the first ones”. I have no words to thank a special person, whom was besides me through very bad moments, helping me, supporting me and caring me. My life here became better and happier after to know her. Thank you Natasha for that you it is.

Guna Alexander

Bellaterra, July 2007

Resumen

La demanda de computadores más veloces ha provocado el incremento del área de computación de altas prestaciones, generalmente representado por el uso de sistemas distribuidos como los *clusters* de computadores ejecutando aplicaciones paralelas. En esta área, la tolerancia a fallos juega un papel muy importante a la hora de proveer alta disponibilidad, aislando los efectos causados por los fallos.

Prestaciones y disponibilidad componen un binomio indisociable para algunos tipos de aplicaciones. Por eso, las soluciones de tolerancia a fallos deben tener en consideración estas dos restricciones desde el momento de su diseño.

En esta disertación, presentamos algunos efectos colaterales que se puede presentar en ciertas soluciones tolerantes a fallos cuando recuperan un proceso fallado. Estos efectos pueden causar una degradación del sistema, afectando las prestaciones y disponibilidad finales.

Presentamos RADIC-II, una arquitectura tolerante a fallos para paso de mensajes basada en la arquitectura RADIC (*Redundant Array of Distributed Independent Fault Tolerance Controllers*). RADIC-II mantiene al máximo posible las características de transparencia, descentralización, flexibilidad y escalabilidad existentes en RADIC, e incorpora una flexible funcionalidad de redundancia dinámica, que permite mitigar o evitar algunos efectos colaterales en la recuperación.

RADICMPI es el prototipo de RADIC que implementa un conjunto de las funciones del patrón MPI-1. Este prototipo ha sido adaptado para realizar las tareas de RADIC-II y ha sido usado para validar y evaluar la solución propuesta en varios escenarios a través del uso de *log* de depuración y de un sistema de inyección de fallos. También hemos expandido el conjunto de funciones disponibles en el prototipo, incorporando las funciones no-bloqueantes de MPI, lo que nos ha permitido ejecutar otros tipos de aplicaciones.

Los resultados han mostrado que RADIC-II opera correctamente y es una buena opción para proveer alta disponibilidad a las aplicaciones paralelas sin sufrir la

degradación del sistema en ejecuciones pos-recuperación, re-estableciendo las prestaciones originales del sistema.

Abstract

The demand for computational power has been leading the improvement of the High Performance Computing (HPC) area, generally represented by the use of distributed systems like clusters of computers running parallel applications. In this area, fault tolerance plays an important role in order to provide high availability isolating the application from the faults effects.

Performance and availability form an undissociable binomial for some kind of applications. Therefore, the fault tolerant solutions must take into consideration these two constraints when it has been designed.

In this dissertation, we present a few side-effects that some fault tolerant solutions may presents when recovering a failed process. These effects may causes degradation of the system, affecting mainly the overall performance and availability.

We introduce RADIC-II, a fault tolerant architecture for message passing based on RADIC (Redundant Array of Distributed Independent Fault Tolerance Controllers) architecture. RADIC-II keeps as maximum as possible the RADIC features of transparency, decentralization, flexibility and scalability, incorporating a flexible dynamic redundancy feature, allowing to mitigate or to avoid some recovery side-effects.

RADICMPI is a RADIC prototype implementing a set of functions of the MPI-1 standard. This prototype was adapted to perform the RADIC-II tasks and used to validate and evaluate RADIC-II in several scenarios through using a debug log and a fault injection system. We also expanded the set of functions available in the prototype incorporating the MPI non-blocking functions, which allows executing other kind of applications.

The results has shown that RADIC-II operates correctly and becomes itself as a good approach to provide high availability to the parallel applications without suffer a system degradation in post-recovery execution, reestablishing the original system performance.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Since its creation, computers play an important and increasing role solving complex problems. Following the computers evolution, new and more complex problems can be solved each day. Indeed, it seems that despite how much more powerful are the computers, always will be more applications needing long periods of time to be executed.

This demand for computational power has been leading the improvement of the High Performance Computing (HPC) area, generally represented by the use of distributed systems like clusters of computers running parallel applications. Following there are typical examples of applications areas commonly executed in computer clusters.

- *Fluid-flow simulation*. Consists in simulate the interaction of large three dimensional cells assemblage between themselves, like the weather and climate modelling.
- *Natural behaviour simulation*. A notoriously complex area, that makes computers simulate the real world and its interactions. Good examples are the simulation of forest fire and simulation of individuals.
- *Medicine research*. Studies like of protein folding require *petaflops* of computing power in order to predict the structure of the protein complete from a known sequence of the protein, being applied in many disease treatments.
- *.Astronomy.* Simulation of N bodies under the influence of physical forces, usually gravity and sometimes other forces. Normally used in cosmology to study the process of galaxy cluster formation.

For those applications, the correctly finish and the spent time of their executions become major issues when planning to perform their tasks in computer based solutions. Therefore, it is reasonable to say that those applications commonly have two basic constraints: performance and availability (also known as *performability* [Meyer, 1980]*)*.

However, the complexity of the actual computers, particularly the cluster of computers, left them more susceptible to occurrence of failures. How much components they have, more probable that one of components will fail. This perception is even worse when applied in distributed system like the computer clusters. Each individual fault probability increases the susceptibility of failures in the whole system.

Since the failures affect directly the system availability and indirectly the system performance, these two major requirements may be quantified basing on the mean time to failure (MTTF) and the mean time to repair (MTTR) [Nagajara *et al*. 2005]. A computer cluster configures itself as a system formed by a combination of independent components, and generally needs all of them to produce desirable results, therefore the MTTF of such system will be smaller that each component, in this case the computer nodes.

Nowadays, the supercomputers usually have more than 1000 nodes and at least dozen of them have more than 10000 processors [Top500.Org, 2006], dramatically increasing the fault probability. In order to avoid or mitigate the effects related with the MTTF/MTTR, fault management plays an important task in those systems, providing ways to allow the system tolerates some kind of failures in certain levels.

Many researches proposing fault tolerant solutions for parallel systems has been presented involving different techniques to detect and recovery from faults. Some proposals allows to perform these activities automatically (transparent) like MPICH-V [Bouteiller*, et al.*, 2006], which consists in a communication library developed for the MPICH MPI implementation, or the recovery solution with hierarchical storage proposed in ScoreD [Kondo, M., et al. 2003; Gao *et al,* 2005]. Other solutions try combine recovery and failures prediction like the approach of FT-PRO [Li and Lan, 2006]. Moreover, some works base in manual operating, needing some user intervention like LAM/MPI [Burns *et al,* 1994; Squyres and Lumsdaine, 2003], this

solution uses a coordinated checkpoint scheme activated either by the user application or by an external controller. Since increasing the MTTF is a hard task, these solutions commonly deal with MTTR reduction, making essential a transparent fault management.

Some fault tolerant solutions, in order to assure the correct application ending, may generate system configuration changes during the recovery process. This behavior happens because those ones manage the faults just using the own active cluster resources, in other words, the application continues executing with one less node, but keeping the number of processes, causing a unplanned process distribution.

RADIC (Redundant Array of Distributed Independent Fault Tolerance Controllers) [Duarte, 2007] is a transparent architecture that provides fault tolerance to message passing based parallel systems. RADIC acts as a layer that isolates an application from the possible cluster failures. This architecture does not demand extra resources to provide fault tolerance, because of this, after a failure the controller recovers the faulty process in some existent node of the cluster. As mentioned before, this behavior may leads to system performance degradation in the post-recovery execution.

RADIC has two kinds of processes working together in order to perform the fault tolerance tasks: Observers and Protectors. Each cluster's node has one Protector, and each application's process has one Observer attached. Each Observer communicates with at least one Protector, and each Protector may attend to several Observers. The Observers are in charge of: a) manage the inter-process message passing communication, b) send the log of received messages and c) take and transmit the associated process checkpoints. While the Protectors perform failure detection, diagnosis and creation of a fully distributed stable storage virtual device.

1.2 Goals

Parallel systems are designed intending to achieve certain performance level. In order to satisfy this performance level, the process distribution through the nodes takes in consideration factors like load balance, CPU power, or memory availability.

Whenever this distribution is changed, may lead to system performance degradation, due to processes having an unplanned sharing of the computing power of a node.

All kind of applications cited in the initial of this chapter demands as more computing power as possible, and may not tolerate performance slowdown. In systems ruled by time constraints, it is so critical finish correctly the application as it is finish it before a time limit, which may invalidate the results of the execution. Moreover, the "never stop" systems generally requires high cost special devices, because they cannot support a continuous reduction of the nodes quantity caused by failures, but this requirement may left impracticable the use of those systems maintaining high availability.

In this work we developed RADIC II, incorporating a dynamic redundancy [Koren and Krishna, 2007, p. 25; Hariri, *et al.*, 1992] feature that enables RADIC, via a spare nodes use, to protect the system configuration from the changes that a recovery process may generate. Furthermore, this feature may be also used to restore the initial system process distribution and for maintenance purposes too.

The major premise of RADIC II is to keep as maximum as possible all the features of RADIC referred before. Thus, the dynamic redundancy provided by RADIC II is transparent, decentralized, flexible and scalable too.

The RADIC transparency must be maintained in RADIC II allowing us to manage the entire process of request and use of spare nodes without need any user intervention, or application's code changes. It must be able to find automatically a spare if exists, to discover its state, to request use and to send all information needed to recovery and re-spawn the process in this node.

In order to keep the fully distributed operational mode from RADIC, RADIC II has to remain all information about the spares decentralized. All nodes should to work independently, exchanging information as needed just with a few neighbor nodes. The spare nodes presence must be spread using techniques that do not compromise the RADIC scalability, keeping the low overhead produced by RADIC.

RADIC II needs to preserve the flexibility of RADIC from the point of view of allowing different structures and relationships between its operational elements and of the RADIC parameters tuning. Moreover, RADIC II aims to be more flexible through allowing dynamic insertion of spare nodes, including during the application execution without need to stop it. The dynamic redundancy must be flexible in order to enable the original process distribution reestablishment by faulty nodes replacement. Finally, the flexibility of RADIC II should allow us to try to avoid failures by permitting to use the dynamic redundancy as a mechanism to perform cluster's nodes maintaining tasks, replacing fault-probable nodes for healthy ones.

Others challenges to RADIC II are: a) to impose a negligible overhead in relation to RADIC during failure-free executions. b) To provide a quick recovery process must be when applying spare nodes.

RADIC II, in the same way than RADIC, bases on rollback-recovery techniques, implementing an uncoordinated checkpoint with pessimistic **event log**. This choice was one of the RADIC keys to maintain the independence between processes. Thus, RADIC II also does not need any coordination process that may increase the overhead in large clusters. It should be noted that RADIC II implements an **event log**, instead adopts the usual message log simplification, which may be not enough to assure a correct recovery process.

RADICMPI is the RADIC prototype developed for test the architecture, implementing a set of functions of MPI standard [MPI Forum, 2006]. As secondary goal of this work, we extended the set of implemented MPI functions of the RADICMPI. We included the set of non-blocking functions and all tasks needed to provide fault tolerance to these functions. As we will see in this dissertation, these functions allow us to enlarge the set of possible applications to be executed in RADICMPI, i.e. new benchmark programs. We also used RADICMPI to test the spare nodes functionality and the MPI functions implemented too.

We performed several experiments with RADIC II in order to validate its functionality and to evaluate its appliance in different scenarios. We used the debug log provided by the RADICMPI original implementation for validate the RADIC II

operation, inserting new events related with the spare nodes usage. For this validation we used a ping-pong program due to its simplicity. We applied the same approach in order to validate the non-blocking function.

The evaluation of our solution was made comparing the effects of recovery having or not available spare nodes. These experiments observed two measures: the overall execution time, and the throughput of an application. We applied different approaches for a matrix product algorithm, using a static distributed Master/Worker and a SPMD approach implementing a Cannon algorithm and we executed a N-Body particle simulation using a pipeline paradigm.

1.3 Organization of this dissertation

This dissertation contains eight chapters. In the next chapter, we discuss theoretical concepts about fault tolerance, including availability, usual strategies for provide fault tolerance in message-passing systems and current research in this area.

Chapter 3 presents the concepts and describes the operation and the elements of the RADIC Architecture. Chapter 4 talks about possible side effects that some fault tolerant architectures may cause that affect the post-recovery execution of applications and explains the RADIC II proposal for the system protection, the methods to achieve that protection, concepts related, and changes made in the RADIC architecture.

In chapter 6, we talk about RADICMPI and the new functions implemented, including considerations about to implement MPI non-blocking functions under the fault tolerance concept.

Chapter 7 presents the experiments conducted with RADICMPI in order to perform a functional validation and evaluation of the solution. Finally, in Chapter 7 we state our conclusions and remained open lines.

Chapter 2

Fault Tolerance

In this chapter, we will discuss theoretical concepts involving fault tolerance, its appliance in message passing systems and current implementations.

2.1 What is fault?

Before starts to discuss about fault tolerance concepts, it is important to define what exactly a fault is. Generally, the terms fault, error and failure are mentioned interchangeably. By definition, failure is the perception of undesirable behavior of a system, meaning that the system do not produce the expected results, as example a software abnormal ending. An error is the generating event which leads to a subsequent failure, unless it exists some corrective actions, as example an programming error leads to an abnormal end except if this error was caught and treated avoiding the failure. Finally, fault it is a system defect with potential to generate errors. Thus, a fault may cause an error, which may cause a failure.

Fault tolerance may be defined as the ability to avoid failures despite existence of errors generated by some fault. Fault tolerance has two basic goals: To increase the overall reliability of a system (despite individual faults of its components) and to increase the system availability [Jalote, 1994, p. 30].

2.2 Availability

Availability is one of the major requirements when using parallel computers. Any user of the applications exemplified in the Chapter 1 expects to have the system available during the entire execution of its work. The equation 1 represents mathematically the meaning of availability [Koren and Krishna, 2007, p. 5].

$$A = \frac{MTTF}{MTTF + MTTR} \qquad 1$$

According with the equation 1, the availability is a ratio between the component mean time to failure (MTTF) and the MTTF itself adding the mean time to repair the component (MTTR). In actual literature, there are some definitions about the MTTF or MTBF (mean time between failures) metrics, according with each need. We decide consider the approach given by [Koren and Krishna, 2007, p. 5] that is the same given by [Jalote, 1994, p. 38] and [Nagajara]. Indeed, [Koren and Krishna, 2007, p. 5] consider that MTBF includes the time needed to repair the component, resulting MTBF = MTTF+MTTR.

The MTTF metrics commonly assigns a reliability measure. From the equation 1 we can deduce that there are two ways to increase the availability of a system: either by increasing the reliability of its components or by decreasing the time for repair. To increase the components reliability generally implies to use high cost equipments, which sometimes makes not viable its implementation. Therefore, fault tolerance plays its role by reducing the MTTR. Indeed, we only achieve a theoretical 100% availability by having MTTR to zero, once a component with infinite MTTF is unachievable by now.

2.2.1 Availability Metrics

In order to measure the availability, the industry generally adopts the “nines” concept. This approach quantifies the availability by the uptime percentage. As we see in the Table 2-1, many “nines” means lower downtimes in a specific period of time.

Table 2-1: Availability Metrics

Percentage Uptime	Downtime per Year	Downtime per week
98%	7.3 days	3.37 hours
99%	3.65 days	1.67 hours
99.9%	8.75 hours	10.09 minutes
99.99%	52.5 minutes	1 minute
99.999%	5.25 minutes	6 seconds
99.9999%	31.5 seconds	0.6 seconds

Most of fault tolerant systems aims achieve a “five nines” availability level, been considered as high availability, although the notion about high availability be relative. In order to achieve referred level, just playing with the MTTR reduction, it is imperative that the fault tolerant system can automatically detect, diagnose and repair the fault.

2.3 Fault Tolerance Applied in Message Passing Systems

Message passing in a strict sense is a common technique used in parallel computers in order to provide communication between concurrent processes. This technique takes the following assumptions:

- The processes have and only access its own local memory
- All communications between the process are sending and receiving messages

- The data interchange requires cooperative actions in each process, meaning that a message sending needs a correspondent receive in the other process.

With these simple assumptions, message passing is widely used for parallel computing because fits well in cluster of workstations or supercomputers, which are interconnected by a network. The figure 2-1 exemplifies the functioning of a simple message passing system with three processes sending and receiving messages (diagonal arrows) trough the timeline (horizontal arrows)

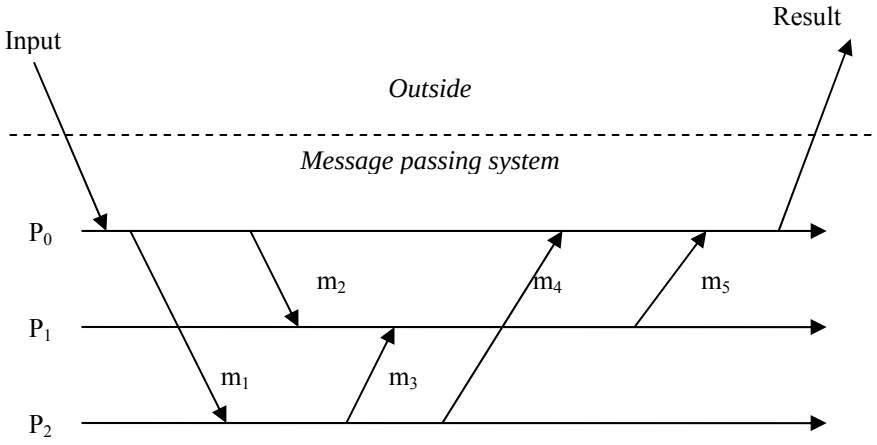

Figure 2-1: A message passing with three processes interchanging messages.

Parallel computers using message passing are more susceptible to the effects of a failure. In these architectures, a fault may occur in a node or either in the communication network. If the fault occurs in the network, the behavior of the systems depends if the implementation provides or not mechanisms like timeout and if the fault is transient or not. When a node fails, the processing assigned to it will be lost and may incur in the inaccurate, useless or incorrect result of the parallel application.

There are many techniques developed to increase the overall reliability and provide high availability for message passing distributed systems including

replication protocols, self-stabilizing protocols and rollback-recovery protocols [Bouteiller, *et al.*, 2006]. Rollback-recovery is widely studied and used in order to provide fault tolerance for message passing systems.

2.4 Rollback-Recovery

Rollback-recovery is a protocol or technique to provide fault tolerance basing in backs the program execution to a point before the failure and in some ways, retry the computation. According Shooman [Shooman, 2002, p.270] there are four basic types of rollback-recovery techniques:

Reboot/restart techniques – It is the simplest recovery technique, but the weakest too. This approach consists in restart the system or the application from the beginning. It is acceptable when the time spent in computation is still small and the time needed to restart the system or application is satisfactory. When the restart procedure is made automatically, this technique is generally referred as *recovery*.

Journaling techniques – It bases in periodically stores all inputs to the system. In failures cases, the processing may be repeated automatically. This technique is a usual feature in most word processors.

Retry techniques – This technique is more complex and supposes that the failure is transient and in a subsequent moment, the system can operates normally. It bases on stay performing the action repeatedly until achieve a maximum of attempts or achieve a correct result. Disk controllers are a good example of retry use.

Checkpoint techniques – It can be said that the checkpoint technique is a improvement of the reboot one. In this approach, the system state is saved periodically, so the application or the system just needs to back to most recent checkpoint before the failure.

Due to the characteristics of the applications running in parallel systems, usually executing during a long time, the checkpoint approach becomes more suitable for these systems. Performing checkpoint is a more difficult task in distributed systems compared with centralized ones [Kalaiselvi and Rajaraman, 2000]. This difficult is

because distributed systems are compound by a set of independent processors with individual lines of execution, and there is not a global synchronized clock between them, which allows starting a checkpoint at same time, saving the global state of the parallel application.

2.4.1 Basic Concepts

Before continuing discussing about rollback-recovery and checkpoint, we should introduce some important concepts involving the rollback-recovery in distributed systems. These concepts will be useful to understand how works our solution.

Checkpoint

Chekpoints, also known as recovery points, may be considered as the state saving of a process. In this procedure, all information needed to re-spawn the process is stored in a stable storage. This information is compounded by variable and register values, control point, thread states, etc. In failure case, the fault tolerant system use this saved state to recover the process. In single machines, the checkpoint process is not a complex issue, but when applied in a distributed context it is not quite simple. As the processes communicate between themselves, each checkpoint must to reflect all relevant communication exchanged.

Stable Storage

The use of checkpoints to perform rollback-recovery generally requires that system state must be available after the failure. In order to provide this feature the fault tolerance techniques suppose the existence of a stable storage, which survives to any failures occurred in the system, when all system will be saved. Although stable storage is usually confused with physical disk storage, it is just an abstract concept [Elnozahy, *et al.*, 2002]. A stable storage may be implemented in different ways:

a) It may be a disk array using RAID, allowing tolerates any number of non-transient failures;

b) If using a distributed system, a stable storage may be performed by the memory of a neighbor node;

c) If just it needs tolerate transient faults, a stable storage may be implemented using a disk in the local machine.

Consistent System State

The major goal of a rollback-recovery protocol is bring back the system working and producing the expected results. Rollback-recovery is a quite simple to implement in a single process application, but in distributed systems, with many processes executing parallel, it becomes a hard task. In the parallel applications using message passing, the state of the system comprises the state of each process running in different nodes and are communicating between them. Therefore, take a checkpoint of a process individually may not represent a snapshot of the overall system.

Hence, we can define consistent system state as one which each process state reflects all interdependences with the other processes, in other words, if a process accuses a message receipt, the sender process must be accuses the message sending too. We can say that during a failure-free execution, any global state taken is a consistent system state.

Domino Effect

The *domino effect* [Koren and Krishna, 2007] may occur when the processes of a distributed application take their checkpoints in an uncoordinated manner. When a failed process rollbacks to its most recent checkpoint, its state may not reflect a communication with other processes, forcing these processes to roll back to checkpoint prior this communication. This situation may continue happening until reach the initial of the execution. Following, we exemplify this happening by the situation depicted in Figure 2-2 that shows an execution in which processes take their checkpoints (represented by blue circles) without coordinating with each other.

We consider the process starts as an initial checkpoint. Suppose that process P_0 fails and rolls back to checkpoint A. The rollback of P_0 invalidates the sending of message m_6, and so P_1 must roll back to checkpoint B in order to "invalidate" the receipt of the message m_6. Thus, the invalidation of message m_6 propagates the rollback of process P_0 to the process P_1, which in turn invalidates message m_5 and

forces P_2 to roll back as well. Because of the rollback of process P_2, process P_3 must also rollback to invalidate the reception of m_4. Those cascaded rollbacks may continue and eventually may lead to the *domino effect*, which forces the system to roll back to the beginning of the computation, in spite of all saved checkpoints.

The amount of rollback depends on the message pattern and the relation between the checkpoint placements and message events. Typically, the system restarts since the last recovery line. However, depending on the interaction between the message pattern and the checkpoint pattern, the only bound for the system rollback is the initial state, causing the loss of all the work done by all processes. The dashed line shown in Figure 2-2 represents the recovery line of the system in case of a failure in P_0.

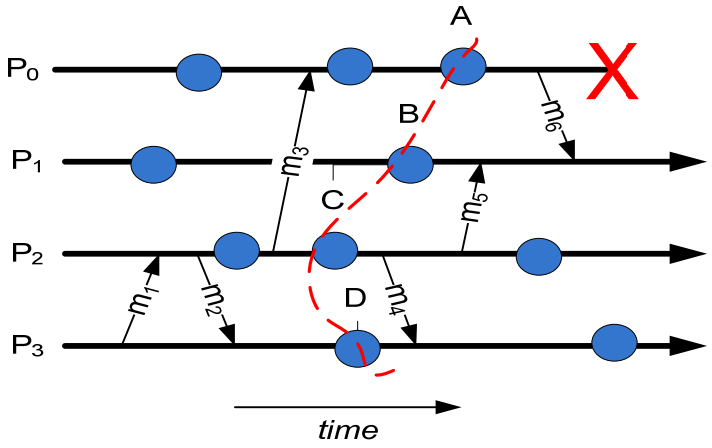

Figure 2-2: Domino effect

In-transit messages

A message that is in the state of the sender but is not yet in the state of the receiver is an example of an in-transit message. A message that appears in the receiver state but not in the sender state is an orphan message. The in-transit message generally is not a problem. If the model presumes a reliable communication channel, this one guarantees the delivery of all messages. However, in systems that do not provide a reliable communication, the rollback-recovery relies that the application been executed provides the mechanisms in order to guarantee the message delivery.

Logging Protocols

Log-based rollback recovery is a strategy used to avoid the domino effect caused by uncoordinated checkpoints. Logging protocols is a set of protocols whose take message logs besides checkpoints. Such protocols base on the *piecewise deterministic* (PWD) assumption [Strom and Yemini, 1985]. Under this assumption, the rollback recovery protocol can identify all the nondeterministic events executed by each process. For each nondeterministic event, the protocol logs a determinant that contains all needed information to replay the event should it be necessary during recovery. If the PWD assumption holds, a log-based rollback-recovery protocol can recover a failed process and replay the determinants as if they have occurred before the failure.

The log-based protocols require only that the failed processes roll back. During the recovery, the messages that were lost because of the failure are "resent" to the recovered process in the correct order using the message logs. Therefore, log-based rollback-recovery protocols force the execution of the system to be identical to the one that occurred before the failure. The system always recovers to a state that is consistent with the input and output interactions that occurred up to the fault.

2.5 Checkpoint Based Protocols

The goal of rollback-recovery protocols based on checkpoint is to restore the system to the most recent consistent global state of the system, in other words, the most recent recovery line. Since such protocols do not rely on the PWD assumption, they do not care about nondeterministic events, that it means, they do not need to detect, log or replay nondeterministic events. Therefore, checkpoint-based protocols are simpler to implement and less restrictive than message-log methods.

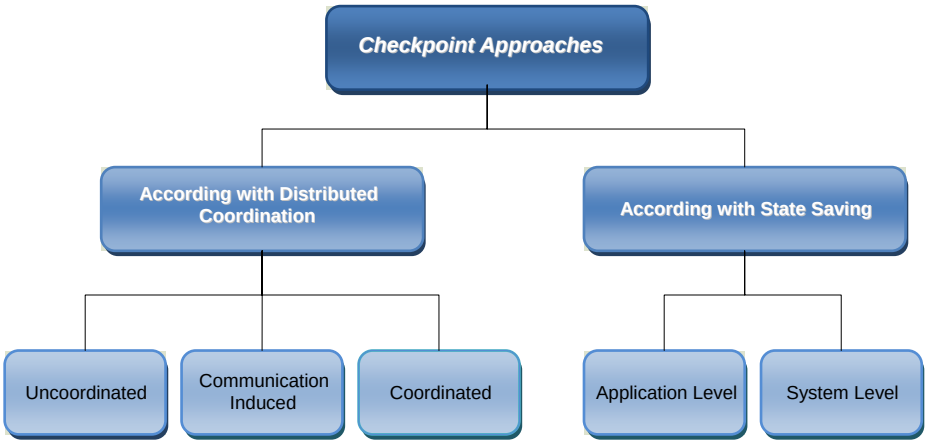

Figure 2-3: Different checkpoint approaches.

The Figure 2-3 shows a classification scheme for checkpoint approaches, basing on where is performed, application level or system level, or in the coordination strategy, uncoordinated, communication induced or coordinated. The next topics explain the three categories of the checkpointing strategies used by the checkpoint-based protocols: *uncoordinated*, *coordinated* and *communication-induced*.

2.5.1 Uncoordinated checkpointing

In this method, each process has total autonomy for making its own checkpoints. Therefore, each process chooses to take a checkpoint when it is more convenient to it (for instance, when the process's state is small) and does not care about the checkpoints of the other processes. Zambonelli [Zambonelli, 1998] makes an evaluation of several uncoordinated checkpoint strategies.

The uncoordinated strategy simplifies the checkpoint mechanism of the rollback-recovery protocol because it gives independence for each process manage its checkpoint without any negotiation with the other processes. However, such independence of each process comes under a cost expressed as follows:

a) There is the possibility of domino effect and all its consequences;

b) A process can take useless checkpoint since it cannot guarantee by itself that a checkpoint is part of a global consistent-state. These checkpoint will overhead the system but will not contribute to advance the recovery line.

c) It is necessary to use garbage collection algorithm to free the space used by checkpoints that are not useful anymore.

d) It is necessary a global coordination to compute the recovery line, what can be very expensive in application with frequent output commit.

2.5.2 Coordinated Checkpointing

In this approach, the processes must synchronize their checkpoint in order to create a consistent global state. A faulty process always will restart from its most recent checkpoint, so the recovery is simplified and the domino effect avoided. Furthermore, as each process only needs to maintain one checkpoint in stable storage, there is no the need of a garbage collection scheme and the storage overhead is reduced.

The main disadvantage is the high latency involved when operating with large systems. Because of this, the coordinated checkpoint protocol is barely applicable to large systems.

Although straightforward, this scheme can yield in a large overhead. An alternative approach is to use a non-blocking checkpoint scheme like the proposed in [Chandy and Lamport, 1985] and in [Elnozahy, *et al.*, 1992]. However, non-blocking schemes must prevent the processes from receiving application messages that make the checkpoint inconsistent.

The scalability of coordinated checkpointing is weak because all processes must to participate in every checkpoint and transmits their checkpoints to a stable storage that generally is centralized, this activity may cause a communication bottleneck.

2.5.3 Communication-Induced Checkpointing (CIC)

The communication-induced checkpointing protocols do not require that all checkpoints be coordinated and do avoid the domino effect. There are two kinds of checkpoints for each process: local checkpoints that occur independently and forced checkpoints that must occur in order to guarantee the eventual progress of the recovery line. The CIC protocols take forced checkpoints to prevent the creation of useless checkpoints, that is, checkpoints that will never be part of a consistent global state (and so they will never contribute to the recovery of the system from failures) although they consume resources and cause performance overhead.

As opposed to coordinated checkpointing, CIC protocols do not exchange any special coordination messages to determine when forced checkpoints should occur; instead, they piggyback protocol specific information on each application message. The receiver then uses this information to decide if it should take a forced checkpoint. The algorithm to decide about forced checkpoints relies on the notions of *Z-path* and *Z-cycle* [Alvisi, *et al.*, 1999]. For CIC protocols, one can prove that a checkpoint is useless if and only if it is part of a *Z-cycle*.

Two types of CIC protocols exist: indexed-based coordination protocols and model-based checkpointing protocols. It has been shown that both are fundamentally equivalent [Helary, *et al.*, 1997a], although in practice they have some differences [Alvisi, *et al.*, 1999].

Indexed-based coordination protocols

These protocols assign timestamps to local and forced checkpoints such that checkpoints with the same timestamp at all processes form a consistent state. The timestamps are piggybacked on application messages to help receivers decide when they should force a checkpoint [Elnozahy, *et al.*, 2002].

In CIC, each process has a considerable autonomy in taking checkpoint. Therefore, the use of efficient policies in order to decide when to take checkpoints can lead to a small overhead in the system. Since these protocols do not require processes

to participate in a globally coordinated checkpoint, they can, in theory, scale up well in systems with a large number processes [Elnozahy, *et al.*, 2002].

Model-based protocols

These schemes prevent useless checkpoint using structures that avoid patterns of communications and checkpoints that could lead to useless checkpoints or Z-cycles. They use a heuristic in order to define a model for detecting the possibility that such patterns occur in the system. The patterns are detected locally using information piggybacked on application messages. If such a pattern is detected, the process forces a checkpoint to prevent that the pattern occurs [Elnozahy, *et al.*, 2002].

Model-based protocols are always conservative because they force more checkpoints than could be necessary, once each process does not have information about the global system state because there is no explicit coordination between the application processes.

2.5.4 Comparing the checkpoint protocols

It is reasonable to say that the major source of overhead in checkpointing schemes is the stable storage latency. Communication overhead becomes a minor source of overhead as the latency of network communication decreases. In this scenario, the coordinated checkpoint becomes worthy since it requires less accesses to stable storage than uncoordinated checkpoints. Furthermore, in practice, the low overhead gain of uncoordinated checkpointing do not justify neither the complexities of finding the recovery line after failure and performing the garbage collection nor the high demand for storage space caused by multiple checkpoints of each process [Elnozahy, *et al.*, 2002].

CIC protocol, in turn, does not scale well as the number of process increases. The required amount of storage space is also difficult to predict because the occurrence of forced checkpoints at random points of the application execution.

2.6 Log-based protocols

These protocols require that only the failed process to roll back. During normal computation, the processes log the messages into a stable storage. If a process fails, it will recover from a previous state and the system will lose the consistency since there may be missed messages or orphan messages related to the recovered process[Elnozahy and Zwaenepoel, 1994]. During the process's recovery, the logged messages will be recovered properly from the message log, so the process can resume its normal operation and the system will reach a consistent state again [Jalote, 1994].

Log-based protocols consider that a parallel-distributed application is a sequence of deterministic state intervals, each starting with the execution of a nondeterministic event [Jalote, 1994]. Each nondeterministic event relates to a unique *determinant*. In distributed systems, the typical nondeterministic event that occurs to a process is the receipt of a message from another process (*message logging* protocol is the other name for these protocols.) Sending a message, however, is a deterministic event. For example, in Figure 2-2, the execution of process P_3 is a sequence of three deterministic intervals. The first one is the process' creation and the other two starts with the receipt of m_2 and m_4. The initial state of the process P_3 is the unique determinant for sending m_1.

During failure-free operation, each process logs the determinants of all the received messages onto stable storage. Additionally, each process also takes checkpoints to reduce the extent of rollback during recovery. After a failure occurs, the failed processes recover by using the checkpoints and logged determinants to replay the corresponding nondeterministic events precisely as they occurred during the pre-failure execution. Because the execution within each deterministic interval depends only on the sequence of received messages that preceded the interval's beginning, the recovery procedure reconstructs the pre-failure execution of a failed process up to the first received message that have a no logged determinant.

Log-based protocols guarantee that upon recovery of all failed processes, the system does not contain any orphan process. A process is orphan when it does not fail and its state depends on the execution of a nondeterministic event whose determinant

cannot be recovered from stable storage or from the volatile memory of a surviving process [Elnozahy, *et al.*, 2002].

The way a specific protocol implements the no-orphan message condition affects the protocol's failure-free performance overhead, the latency of output commit, and the simplicity of recovery and garbage collection schemes, as well as its potential for rolling back correct processes. These differences lead to three classes of log-based protocols: *pessimistic*, *optimistic* and *causal*.

2.6.1 Pessimistic log-based protocols

Despite all efforts in order to provide fault tolerance, in reality, failures are rare. Although this, these protocols assume a pessimistic behavior, supposing that a failure may occur after any nondeterministic event in the computation. In their most simple form, pessimistic protocols log the determinant of each received message before the message influences in the computation. Pessimistic protocols implement a property often referred to as *synchronous logging*, i.e., if an event has not been logged on stable storage, then no process can depend on it [Elnozahy, *et al.*, 2002]. Such condition assures that orphan processes will never exist in systems using pessimistic log-based protocol.

Processes also take periodic checkpoints in order to limit the amount of work that the faulty process has to repeat during recovery. If a failure occurs, the process restarts from its most recent checkpoint. During the recovering procedure, the process uses the logged determinants to recreate the pre-failure execution, without needing any synchronization between the processes. The checkpoint period implies directly in the overhead imposed by fault tolerance, creating a dilemma: if checkpoints is taken in short periods, it will cause greater overhead during a failure-free execution, but less *expensive* will be the recovery process.

Synchronous logging enables that the observable state of each process is always recoverable. This property leads to four advantages at the expense of a high computational overhead penalty [Elnozahy, *et al.*, 2002]:

e) Recovery is simple because the effects of a failure influences only the processes that fails.

f) Garbage collection is simple because the process can discard older checkpoints and determinants of received messages that are before the most recent checkpoint.

g) Upon a failure, the failed process restarts from its most recent checkpoint what limits the extent of lost computation.

h) There is no need of a special protocol to send messages to outside world.

Due to the synchronism, the log mechanism may enlarge the message latency perceived by the sender process, because it has to wait until the stable storage confirms the message log writing in order to consider the message as delivered. In order to reduce the overhead caused by the synchronous logging, the fault tolerance system may applies a *Sender Based Message Logging* model that stores the log in the volatile memory of the message sender, supposing as a reliable device. In this case, the recovery process is more complex, needing to involve each machine that has communicated with the failed process.

2.6.2 Optimistic log-based protocols

In opposition, these protocols suppose that failures occurs rarely, relaxing the event log, but allowing the orphans processes appearing caused by failures in order to reduce the failure-free performance overhead. However, the possibility of appearing orphans processes lefts the recovery process more complex, garbage collection and output commit [Jalote, 1994]. In optimistic protocols as in pessimistic protocols, every process take checkpoint and message log asynchronously [Alvisi and Marzullo, 1998]. Furthermore, a volatile log maintains each determinant meanwhile the application processes continue their execution. There is no concern if the log is in the stable storage or in the volatile memory. The protocol assumes that logging to stable storage will complete before a failure occurs (thence its optimism).

If a process fails, the determinants in its volatile log will be lost, and the state intervals started by the nondeterministic events corresponding to these determinants

are unrecoverable. Furthermore, if the failed process sent a message during any of the state intervals that cannot be recovered, the receiver of the message becomes an orphan process and must roll back to undo the effects of receiving the message. To perform these rollbacks correctly, optimistic logging protocols track causal *dependencies* during failure-free execution [Elnozahy, *et al.*, 2002; Jalote, 1994]. Upon a failure, the dependency information is used to calculate and recover the latest global state of the pre-failure execution in which no process is in an orphan. Since there is now a dependency between processes, optimistic protocols need to keep multiple checkpoints what complicates the garbage collection policy.

The recovery mechanism in optimistic protocol can be either *synchronous* or *asynchronous*. Each one is explained bellow [Elnozahy, *et al.*, 2002] and detailed bellow:

Synchronous recovery

During failure free operation, each process updates a state interval index when a new state interval begins. The indexes serve to track the dependency between processes using two distinct strategies: direct or transitive. In synchronous recovery, all processes use this dependency information and the logged information to calculate the maximum recovery line. Then, each process uses the calculated recovery line to decide if it must roll back.

In direct tracking strategy, each outgoing message contains the state interval index of the sender (piggybacked in the message) in order to allow the receiver to record the dependency directly caused by the message. At recovery time, each process assemblies its dependencies to obtain the complete dependency information.

In transitive tracking, each process maintains a size-N vector V, where $V[i]$ is the current state interval index of the process P_i itself, and $V[j]$, $j \neq i$, records the highest index of any state interval of a process P_j on which P_i depends. Transitive dependency tracking generally incurs a higher failure-free overhead because of piggybacking and maintaining the dependency vectors, but allows faster output commit and recovery.

Asynchronous recovery

In this scheme, a recovery process broadcasts a rollback announcement to start a new incarnation. Every process that receives a rollback announcement checks if it has become an orphan because of the announcement and then, if necessary, it rolls back and broadcasts its own rollback announcement.

Asynchronous recovery can produce a situation called exponential rollbacks. Exponential rollbacks occur when a process rolls back an exponential number of times because of a single failure. The asynchronous protocol eliminates exponential rollbacks by either distinguishing failure announcements from rollback announcements or piggybacking the original rollback announcement from the failed process on every subsequent rollback announcement that it broadcasts.

2.6.3 Causal log-based protocols

These protocols avoid the creation of orphan processes by ensuring that the determinant of each received message, which causally precedes a process's state, either is in stable storage or is available locally to that process [Elnozahy, *et al.*, 2002]. Such protocols dispense synchronous logging, which is the main disadvantage of pessimistic protocols, while maintaining their benefits (isolation of failed processes, rollback extent limitation and no apparition of orphan processes). However, causal protocols have a complex recovery scheme.

In order to track causality, each process piggybacks the non-stable determinants that are in its volatile log on the messages it sends to other processes. On receiving a message, a process first adds any piggybacked determinant to its volatile determinant log and then delivers the message to the application.

2.7 Comparing the rollback-recovery protocols

Table 2-2 summarizes the differences among the rollback-recover protocols. The decision about which one is best suited for a given system than another is not straightforward. It depends on diverse factors like probability of failures, message pattern among application processes, the resources consumed, etc.

Using the four basic requirements as reference (scalability, transparency, decentralization and flexibility,) we compared the protocols described in Table 2-2 in order to choose the ones that could best attend to these requirements. We immediately discarded the uncoordinated, the CIC and the optimistic protocol because they allow the creation of orphan processes.

We defined that, in order to be scalable, the number of computational elements of the parallel computer must not influence the operation of the protocol. To satisfy such requirement, the recovery mechanism must be independent of the number of elements present in the system. For this, it is necessary that the process recovering rest only on local information, i.e, it cannot rests on the information about other process.

Looking again at Table 2-2, one can see that the only protocol that allows local decision during the recovery phase is the pessimist message-log. This protocol also increases the efficiency in terms of storage space because each process only needs to store its last checkpoint in order to recover. Additionally, this feature greatly simplifies the implementation of the garbage collection mechanism.

The pessimistic rollback-recovery protocol does not restrict the other features. It may operate in the system level so the application is not aware about it (transparency). It has an intrinsic decentralization because each process only needs local information to recover from faults.

Table 2-2: Comparison between rollback recovery protocols [Elnozahy, *et al.*, 2002]

	Checkpointing			Message logging		
	Uncoord.	Coordinated	CIC	Pessimistic	Optimistic	Causal
PWD assumed	No	No	No	Yes	Yes	Yes
Checkpoint per process	Several	1	Several	1	Several	1
Domino effect	Possible	No	No	No	No	No
Orphan processes	Possible	No	Possible	No	Possible	No

Rollback extent	Unbounded	Last global checkpoint	Possibly several local checkpoints	Last checkpoint	Possibly several checkpoints	Last checkpoint
Recovery data	Distributed	Distributed	Distributed	Distributed or Local	Distributed or local	Distributed
Recovery protocol	Distributed	Distributed	Distributed	Local	Distributed	Distributed
Output commit	Not possible	Global coordination required	Global coordination required	Local decision	Global coordination required	Local decision

Finally, the pessimistic message log protocol is very flexible because the operation of the fault tolerance mechanism is restricted to each process, allowing the building of several different arrangements in order to attend to the performance or efficiency requirements of the system. For example, each process may have its own checkpoint interval in order to reduce the overall cost of the checkpoint procedure.

2.8 Current Researches

Fault tolerance becomes a major issue in the high performance computing area. Hence, many works has been developed in order to provide fault tolerance for parallel systems. Following, there are some of the current researches in this area.

MPICH-V [Bouteiller, *et al.*, 2006] is a framework that aims to compare different approaches for fault tolerance over the MPICH-1 implementation of MPI. It is an evolution of other three implementations. This framework is a MPICH channel library implementation associated with a runtime environment. The MPICH-V runtime environment is formed by some components: Dispatcher, Channel memories, Checkpoint servers, and Computing/Communicating nodes. The dispatcher is the responsible to launch the entire runtime environment, and performs a fault detection task by monitoring the runtime execution. Channel Memories are dedicated nodes providing a service of tunneling and repository. The architecture assumes neither central control nor global snapshots. The fault tolerance bases on an uncoordinated checkpoint protocol that uses centralized checkpoint servers to store communication context and computations independently.

FT-Pro [Li and Lan, 2006] is a fault tolerance solution that bases on a combination of rollback-recovery and failure prediction to take some action at each decision point. Using this approach, this solution aims to keeps the system performance avoiding excessive checkpoints. Currently support three different preventive actions: Process migration, coordinated checkpoint using central checkpoint storages and no action. Each preventive action is selected dynamically in an adaptive way intending to reduce the overhead of fault tolerance. FT-Pro works an initially determined and static number of spare nodes.

Score-D [Kondo, M., et al. 2003]. The Score-D checkpoint solution is a fault tolerance solution used in the Score cluster implementing a distributed coordinated checkpoint system. In Score's checkpointing algorithm, each node stores its checkpoint data into the local disk in parallel. In addition, it saves redundant data to ensure the reliability for non-transient failures. A server is in charge to send a heartbeat to each node in order to detect failures. This redundancy is achieved through parity generation. In the recovery task, this system uses the parity data distributed over the nodes, in order to reconstitute the checkpoint image and restart the process in a spare node allocated statically at the program start. The initial solution has a clear bottleneck caused by disk writing, so Gao [Gao *et al,* 2005] proposed an optimization using a hierarchical storage approach combined with a diskless checkpointing for transient failures tolerance.

MPICH-V2 [Bouteiller*, et al.*, 2003a] is a improvement in the previous version, implementing the sender based pessimistic log (the computing node now keeps the message-log), is well suited for homogeneous network large-scale computing. Unlike its antecessor, it requires a few number of stable components to reach good performance on a cluster. MPICH-V2 replaced the channel memories concept by event loggers assuring the correct replace of messages during recovers.

MPICH-VCL [Bouteiller*, et al.*, 2003b] is designed for extra low latency dependent applications. It uses coordinated checkpoint scheme based on the Chandy-Lamport algorithm [Chandy and Lamport, 1985] in order to eliminate overheads during fault free execution. However, it requires restarting all nodes (even non-

crashed ones) in the case of a single fault. Consequently, it is less fault resilient than message logging protocols, and is only suited for medium scale clusters.

LAM/MPI [Squyres and Lumsdaine, 2003; Burns *et al, 1994*]. This implementation uses a component architecture called System Services Interface (SSI) that allows checkpoint an MPI application using a coordinated checkpoint approach. This feature is not automatic, needing a back-end checkpoint system. In case of failure, all applications nodes stop and a restart command is needed. LAM/MPI demands a faulty node replacement. This procedure is neither automatic, nor transparent.

MPICH-V1 [Bosilca, *et al.*, 2002] is the first implementation of MPICH-V. This version has a good appliance in very large scale computing using heterogeneous networks. Its fault tolerant protocol uses uncoordinated checkpoint and remote pessimistic message logging. MPICH-V1 well suited for Desktop Grids and Global computing as it can support a very high rate of faults. As this solution requires a central stable storage, it requires a large bandwidth that becomes the major drawback for this implementation.

Starfish [Agbaria and Friedman, 1999] provides failure detection and recovery at the runtime level for dynamic and static MPI-2 programs. Starfish allows the user to control checkpoint and recovery by an user level API allowing the user to control checkpoint and recovery. Both coordinated and uncoordinated checkpoints strategies may be applied by the user choice. Coordinated checkpoint relies on the Chandy-Lamport's algorithm. For an uncoordinated checkpoint, the environment sends to all surviving processes a notification of the failure. The application may take decision and corrective operations to continue execution.

Table 2-3: A comparison of some fault-tolerant MPI solutions based on five relevant features

Solution	Scalable	Fully Decentralized	Transparent	Flexible	Dynamic Redundancy
Cocheck			User		Not informed
Starfish					Not informed
Score-D			User	Yes	Statically Allocated
FT-PRO			User	Yes	Statically Allocated
MPICH-V1			User and Admin	Yes	Statically Allocated

MPICH-V2	Yes		User and Admin	Yes	Statically Allocated
MPICH-VCL			User and Admin		Statically Allocated
LAM/MPI			User		Not Applicable
MPICH-V	Yes		User and Admin	Yes	Statically Allocated

Cocheck [Stellner, 1996] was one of the firsts solutions for fault tolerance in MPI. It works as an independent application making a MPI parallel application fault tolerant. It is implemented at the runtime level (but its implementation on top of tuMPI required some modification of the tuMPI code), on top of a message passing library and a portable process checkpoint mechanism. Cocheck coordinates the application processes checkpoints and flushes the communication channels of the target applications using a Chandy-Lamport's algorithm. A centralized coordinator manages the checkpoint and rollback procedures.

In Table 2-3 we present a summary of these solutions, comparing according five relevant features. We considered a solution scalable when it has not any characteristic that may affect the scalability, i.e. bottlenecks or checkpoint /recovery strategy needing coordination between all nodes. The fully decentralized feature means that in any moment of the fault-tolerance process, including recovery, the solution does not need any central elements. The transparency was analyzed by two points of view: from the user, a programmer, who does not need to change its program code; and from the system administrator, who does not need take care of the recovery activities. We considered flexible, the which ones allowing adjusting some kind of parameter. Finally, we analyzed the presence of the dynamic redundancy feature, considering as flexible, when allows a dynamic insertion of spare nodes and statically allocated when it have a fixed and pre-determined number of spares.

Chapter 3

The RADIC Architecture

This chapter discusses characteristics and behavior of the architecture chosen as basis of our work. In his work, Duarte [Duarte, 2007] introduces a new fault tolerance architecture called RADIC, an acronym for Redundant Array of Independent Fault Tolerance Controllers. As RADIC was intended not uses extra resources, the recovery process causes a system degradation due to the node losses. Hence, we need to supply RADIC with some more features, allowing reducing or avoiding this degradation by protecting the system or allowing preventing the faults.

3.1 RADIC architecture model

Table 3-1: The key features of RADIC

Feature	How it is achieved
Transparency	– No change in the application code – No administrator intervention is required to manage the failure
Decentralization	– No central or fully dedicated resource is required. All nodes may be simultaneously used for computation and protection
Scalability	– The RADIC operation is not affected by the number of nodes in the parallel computer
Flexibility	– Fault tolerance parameters may be adjusted according to application requirements – The fault-tolerant architecture may change for better adapting to the parallel computer structure and to the fault pattern

RADIC establishes an architecture model that defines the interaction of the fault-tolerant architecture and the parallel computer's structure. Figure 3-1 depicts how the RADIC architecture interacts with the structure of the parallel computer (in the lower level) and with the parallel application's structure (in the higher level). RADIC implements two levels between the MESSAGE-PASSING level and the computer

structure. The lower level implement the fault tolerance mechanism and the higher level implements the fault masking and message delivering mechanism.

The core of the RADIC architecture is a fully distributed controller for fault tolerance that automatically handles faults in the cluster structure. Such controller shares the parallel computers resources used in the execution of the parallel application. The controller is also capable to handle its structure in order to survive to failures.

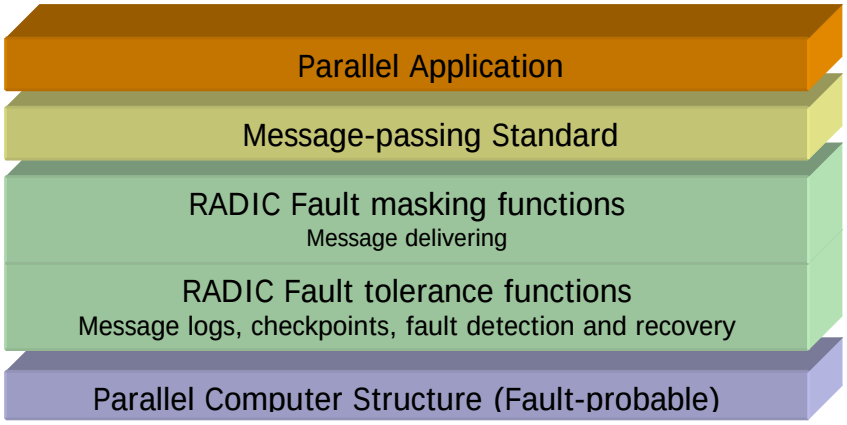

Figure 3-1: The RADIC levels in a parallel system

3.1.1 Failure pattern

We assume that the probability of failures in the nodes follows a Poisson distribution. This assumption is accurate if we consider that:

- the chance that a failure occurs in a time interval is proportional to the interval size;
- the probabilities of failure of each node are independent;
- the probability of multiple failures in a given interval is much smaller than the probability of a single failure.

Basing on these assumptions, we establish that if a node fails, all elements involved in the recovery of the failed processes will survive until the end of the

recovery procedure. In other words, if two or more failures occur concurrently, none of them affects the elements implicated in the recovery of the other failures while the recovering procedure occurs.

Similarly, any number of failures may occur if each failure does not affect an element implicated in the recovery of a previous failure.

3.2 RADIC functional elements

The structure of the RADIC architecture uses a group of processes that collaborate in order to create a distributed controller for fault tolerance. There are two classes of processes: *protectors* and *observers*. Every node of the parallel computer has a dedicated protector and there is a dedicated observer attached to every parallel application's process.

3.2.1 Protectors

There is a protector process in each node of the parallel computer. Each protector communicates with two protectors assumed as neighbors: an antecessor and a successor. Therefore, all protectors establish a protection system throughout the nodes of the parallel computer. In Figure 3-2, we depict a simple cluster built using nine nodes (N_0-N_8) and a possible connection of the respective protectors of each node (T_0-T_8). The arrows indicate the antecessor←successor relationship.

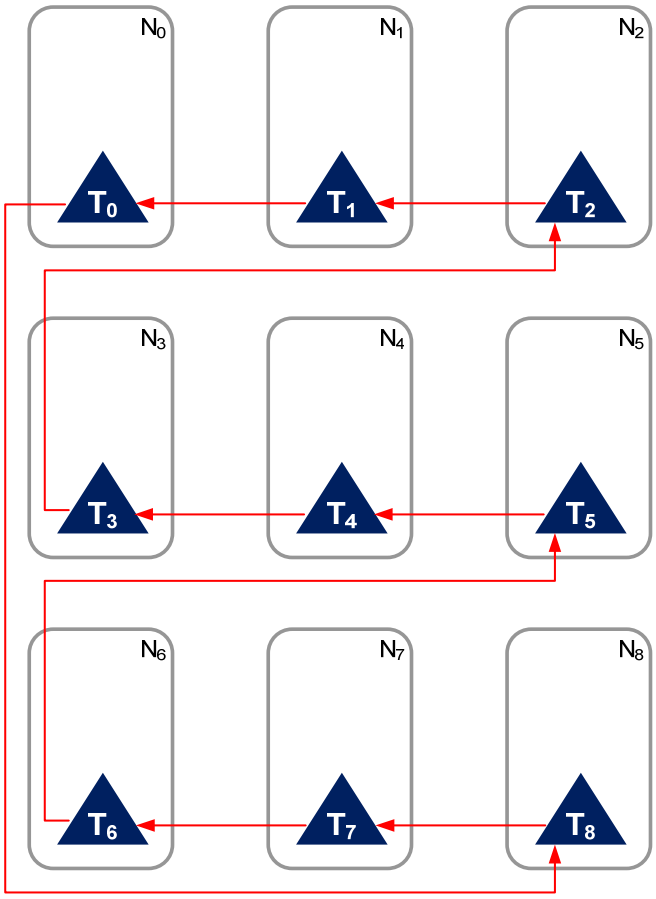

Figure 3-2: An example of Protectors (T_0-T_8) in a cluster with nine nodes. Green arrows indicate the antecessor←successor communication.

The relationship between neighbor protectors exists because the fault detection procedure. There is a heartbeat/watchdog mechanism between two neighbor protectors: one has the watchdog and receives heartbeats from the other. By definition, the protector who has the watchdog is the antecessor and the protector who sends the heartbeats is the successor.

The arrows in Figure 3-2 indicate the orientation of the heartbeat signals from the successor to the antecessor. Actually, each successor has a double identity because it acts simultaneously as a successor for a neighbor and as an antecessor for the other

neighbor. For example, in Figure 3-2, the protector T_7 is the antecessor of the protector T_8 and the successor of the protector T_6.

Each protector executes the following tasks related to the operation of the rollback-recovery protocol:

a) It stores checkpoints and message-logs from the application processes those are running in its successor node;

b) It monitors its neighbors in order to detect failures via a heartbeat/watchdog scheme;

c) It reestablishes the monitoring mechanism with a new neighbor after a failure in one of its current neighbors, i.e., it reestablishes the protection chain;

d) It implements the recovery mechanism.

3.2.2 Observers

Observers are RADIC processes attached to each application processes. From the RADIC operational point-of-view, an observer and its application process compose an inseparable pair.

The group of observers implements the message-passing mechanism for the parallel application. Furthermore, each observer executes the following tasks related to fault tolerance:

a) It takes checkpoints and event logs of its application process and send them to a protector running in another node, namely the antecessor protector;

b) It detects communication failures with another processes and with its protector;

c) In the recovering phase, it manages the messages from the message log of its application process and establishes a new protector;

d) It maintains a mapping table, called *radictable*, indicating the location of all application processes and their respective protectors and updates this table in order to mask faults.

3.2.3 The RADIC controller for fault tolerance

The collaboration between protectors and observers allows the execution of the tasks of the RADIC controller. Figure 3-3 depicts the same cluster of Figure 3-2 with all elements of RADIC, as well as their relationships. The arrows in the figure represent only the communications between the fault-tolerance elements. The communications between the application processes does not appear in the figure because they relate to the application behavior.

Each observer has an arrow that connects it to a protector, to whom it sends checkpoints and message logs of its application process. Such protector is the antecessor of the local protector. Therefore, by asking to the local protector who is the antecessor protector, an observer can always know who its protector is.

Each protector has an arrow that connects it to an antecessor protector. Similarly, it receives a connection from its successor. A protector only communicates with their immediate neighbors. For example, in Figure 3-3, the protector T_5 communicate only with T_4 and T_6. It will never communicate with T_3, unless T_4 fails and T_3 becomes its new immediate neighbor.

The RADIC controller uses the receiver-based pessimistic log rollback-recovery protocol to handle the faults in order to satisfy the scalability requirement. As explained in the item 2.6.1, this protocol is the only one in which the recover mechanism does not demand synchronization between the in-recovering process and the processes not affected by the fault. Such feature avoids that the scalability suffer with the operation of the fault tolerance mechanism.

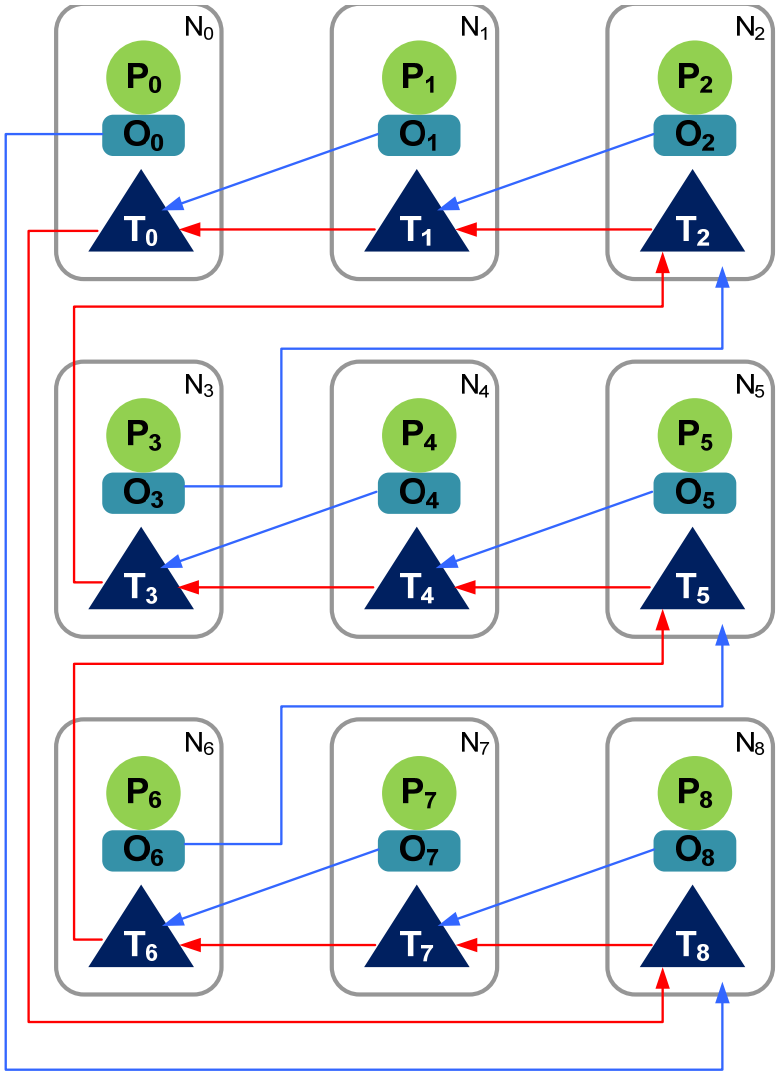

Figure 3-3: A cluster using the RADIC architecture. P_0-P_8 are application process. O_0-O_8 are observers and T_0-T_8 are protectors. O→T arrows represent the relationship between observers and protector and T→T arrows the relationship between protectors.

Besides the fault tolerance activities, the observers are responsible to manage the message-passing mechanism. This activity rests on a mapping table that contains all information required to the correct delivery of a message between two processes. Protectors do not participate directly in the message-passing mechanism, only performing the message log storing.

3.3 RADIC operation

As we seen, the RADIC distributed controller concurrently executes a set of activities related to the fault tolerance. Besides these fault tolerance activities, the controller also implements the message-passing mechanism for the application processes. Following we explain how these mechanism and tasks contribute for the RADIC operation.

3.3.1 Message-passing mechanism

In the RADIC message-passing mechanism, an application process sends a message through its observer. The observer takes care of delivering the message through the communication channel. Similarly, all messages that come to an application process must first pass through its observer. The observer then delivers the message to the application process. Figure 3-4 clarifies this process.

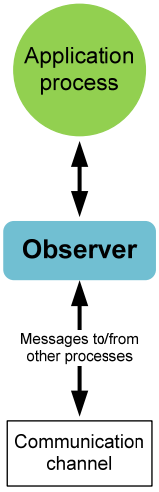

Figure 3-4: The message-passing mechanism in RADIC.

To discover the address of a destination process, each observer uses its routing table, the *radictable*, which relates the identification of the destination process inside the application level with the identification of the destination process inside the communication level. Table 3-2 represents a typical *radictable*.

Table 3-2: An example of *radictable* for the cluster in Figure 3-3

Process identification	Address
0	Node 0
1	Node 1
2	Node 2
3	Node 3
. .	. .

3.3.2 State saving task

In this task, protectors and observers collaborate in order to save snapshots of the parallel application's state. This task is the major responsible for resources consumed by the fault tolerance mechanism as well as for the enlargement in the execution time in the absence of failures.

The system must supply storage space for the checkpoints and the message-logs required by the rollback-recovery protocol. Furthermore, the checkpoint procedure introduces a time delay in the computation because a process may suspend its operation while the checkpoint occurs.

Additionally, the message-log interferes in the message latency, because a process only considers a message delivered after the message is stored in the message log.

Checkpoints

Each observer takes checkpoints of its application process, as well as of itself, and sends them to the protector located in its antecessor node. Figure 3-5 depicts a simplified scheme to clarify the relationship between an observer and its protector.

A checkpoint is an atomic procedure and a process become unavailable to communicate while a checkpoint procedure is in progress. This behavior demands that the fault detection mechanism differentiates a communication failure caused by a

real failure from a communication failure caused by a checkpoint procedure. We explain this differentiation in item 3.3.3.

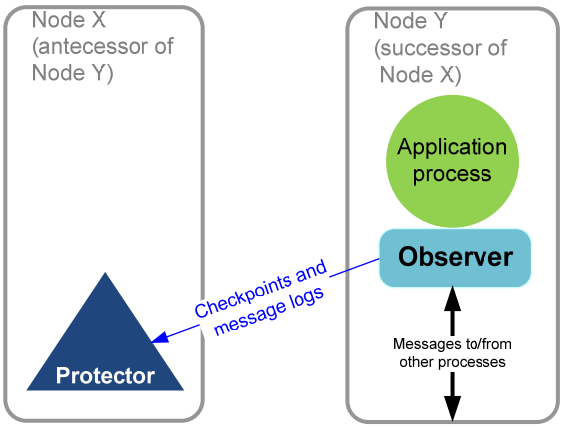

Figure 3-5: Relation between an observer and its protector.

The protectors operate like a distributed reliable storage. The reliability is achieved because the checkpoints and message logs of a process are stored in a different node. Therefore, if a process fails, all information required to recover it is in a survivor node.

Thanks to the uncoordinated checkpoint mechanism of the pessimistic message-log rollback-recovery protocol used by RADIC, each observer may establish an individual checkpoint policy for its application process. Such policy may be time-driven or event-driven. The RADIC architecture allows the implementation of any combination of these two policies.

The time-driven policy is very typical in the fault-tolerant implementations based on rollback-recovery. In this policy, each observer has a checkpoint interval that determines the times when the observer takes a checkpoint.

The event-driven policy defines a trigger that each observer uses in order to start the checkpoint procedure. A typical event-driven policy occurs when two or more observers coordinate their checkpoints. Such policy is useful when two processes have to exchange many messages. In this case, because the strong interaction between

the processes, coordinate the checkpoint is a good way to reduce the checkpoint intrusion over the message exchanging.

When an observer takes a checkpoint of its process, this checkpoint represents all computational work done by such process until that moment. Is such computational work that the observer sends to the protector. As the process continues its work, the state saved in the protector becomes obsolete. To make possible the reconstruction of the process' state in case of failure, the observer also logs in to its protector all messages its process has received since its last checkpoint. Therefore, the protector always has all information required to recover a process in case of a failure, but such state's information is always older than the current process' state.

Message logs

Because the pessimistic log-based rollback-recovery protocol, each observer must log all messages received by its application process. As we have explained in Chapter 2, the use of message logs together with checkpoint optimizes the fault tolerance mechanism by avoiding the domino effect and by reducing the amount of checkpoints that the system must maintain.

The message log mechanism in RADIC is very simple: the observer resends all received messages to its protector, which saves it in a stable storage. The log procedure must complete before the sender process consider the message as delivered. Figure 3-6 depicts the message's delivery mechanism and message's log mechanism.

The log mechanism enlarge the message latency perceived by the sender process, because it has to wait until the protector concludes the message log procedure in order to consider the message as delivered.

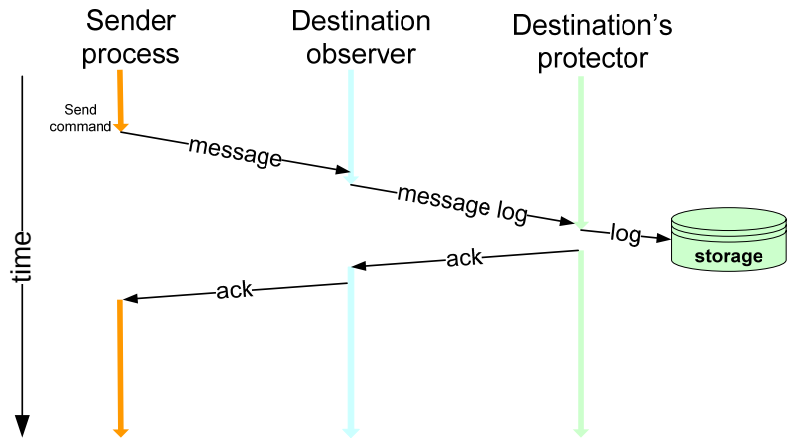

Figure 3-6: Message delivering and message log mechanism.

Garbage collection

The pessimistic message log protocol does not require any synchronization between processes. Each observer is free to take checkpoints of its process without caring about what is happening with other parallel application's process.

This feature greatly simplifies the construction of the garbage collector by the protectors. Because each checkpoint represents the current state of a process, whenever a new checkpoint comes from an observer, the protector may discard all prior checkpoints and message-logs related to that process. Therefore, after a protector receives a new checkpoint from a process, it automatically eliminates the older checkpoint of this process.

3.3.3 Failure detection task

The failure detection is an activity performed simultaneously by protectors and observers. Each one performs specific activities in this task, according to its role in the fault tolerance scheme.

How protectors detect failures

The failure detection procedure contains two tasks: a passive monitoring task and an active monitoring task. Because of this, each protector has two parts: it is, simultaneously, antecessor of one protector and successor of other.

There is a heartbeat/watchdog mechanism between two neighbors. The antecessor is the watchdog element and the successor is the heartbeat element. Figure 3-8 represents the operational flow of each protector element.

A successor regularly sends heartbeats to an antecessor. The heartbeat/watchdog cycle determines how fast a protector will detect a failure in its neighbor, i.e., the response time of the failure detection scheme. Short cycles reduce the response time, but also increase the interference over the communication channel. Figure 3-7 depicts three protectors and the heartbeat/watchdog mechanism between them. In this picture we see the antecessors running the watchdog routine waiting for a heartbeat sent by its neighbor.

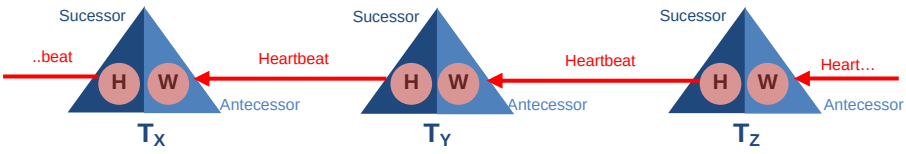

Figure 3-7: Three protectors (T_X, T_Y and T_Z) and their relationship to detect failures. Successors send heartbeats to antecessors.

A node failure generates events in the node's antecessor and in the node's successor. If a successor detects that its antecessor has failed, it immediately starts a search for a new antecessor. The search algorithm is very simple. Each protector knows the address of its antecessor and the address of the current antecessor of its antecessor. Therefore, when a antecessor fails, the protector know exactly who its new antecessor will be.

An antecessor, in turns, begins to wait for a new successor detects a failure in its current successor. Furthermore, the antecessor also starts the recovering procedure, in order to recover the faulty processes that were running in the successor node.

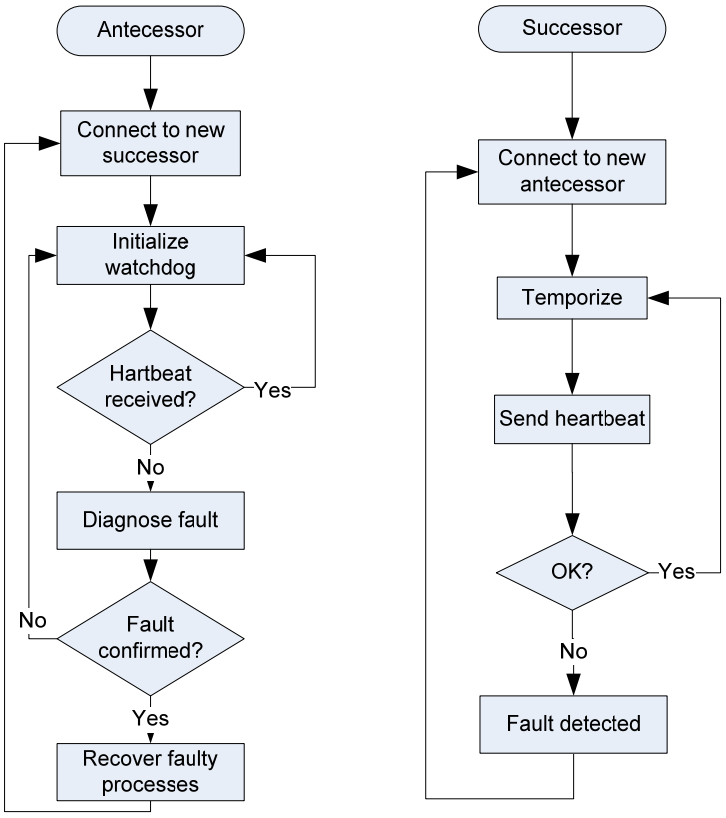

Figure 3-8: Protector algorithms for antecessor and successor tasks

How the observers detect failures

Each observer relates with two classes of remote elements: its protector and the other application processes. An observer detects failures either when the communication with other application processes fails or when the communication with its protector fails. However, because an observer just communicates with its protector when it has to do a checkpoint or a message log, an additional mechanism shall exist to certify that an observer will quickly perceive that its protector has failed.

RADIC provides such mechanism using a warning message between the observer and the local protector (the protector that is running in the same node of the observer). Whenever a protector detects a fail in its antecessor, such protector sends a warning message to all observers in its nodes because it knows that the failed antecessor is the protector that the local observers are using to save checkpoints and message logs.

When an observer receives such message, it immediately establishes a new protector and takes a checkpoint.

How the observers confirm a failure

There are two situations which create a communication failure between application processes, but that must not indicate a node failure. The first failure situation occurs when an observer is taking a checkpoint of its application process. The second occurs when a process fails and restarts in a different node.

In this paragraph, we explain how the observers get rids of the first problem. We will explain how the observer gets rid of the second situation in the description of the Fault Masking Phase.

A process becomes unavailable to communicate inside the checkpoint procedure. Such behavior could cause that a sender process interprets the communication failure caused by the checkpoint procedure as a failure in the destination.

Table 3-3: The *radictable* of each observer in the cluster in Figure 3-3.

Process identification	Address	Protector (antecessor address)
0	Node 0	Node 8
1	Node 1	Node 0
2	Node 2	Node 1
3	Node 3	Node 2
. .	. .	. .

In order to avoid this fake failure detection, before a sender observer assumes a communication failure with a destination process, the sender observer contacts the destination's protector and asks about the destination's status. To allow that each observer knows the location of the protector of the other process, the *radictable* now includes the address of the destination's protector, as shown in Table 3-3.

Analyzing Table 3-3, one may see that the protector in node eight protects the processes in node zero, the protector in node zero protects processes in node one and so forth.

Using its *radictable*, any sender observer may locate the destination's protector. Since the destination's protector is aware about the checkpoint procedure of the destination process, it will inform the destination's status to the sender observer. Therefore, the sender observers can discover if the communication failure is consequence of a current checkpoint procedure.

The radictable and the search algorithm

Whenever an observer needs to contact another observer (in order to send a message) or an observer's protector (in order to confirm the status of a destination), this observer will look for the address of the element in its radictable. However, after a failure occurs, the radictable of an observer becomes outdated, because the address of the recovered process and their respective protectors changed.

To face this problem, each observer uses a search algorithm for calculates the address of failed elements. This algorithm relies on the determinism of the protection chain. Each observer knows that the protector of a failed element (observer or protector) is the antecessor of this element. Since a antecessor is always the previous element in the radictable, whenever the observer needs to find an element it simply looks the previous line in its radictable, and finds the address of the element. The observer repeats this procedure until it finds the element it is looking for.

3.3.4 Recovery task

In normal operation, the protectors are monitoring computer's nodes, and the observers care about checkpoints and message logs of the distributed application

processes. Together, protectors and observers function like a distributed controller for fault tolerance.

When protectors and observers detect a failure, both actuate to reestablish the consistent state of the distributed parallel application and to reestablish the structure of the RADIC controller.

Reestablishing the RADIC structure after failures

The protectors and observers implicated in the failure will take simultaneous atomic actions in order to reestablish the integrity of the RADIC controller's structure. Table 3-4 explicates the atomic activities of each element.

When the recovery task is finished, the RADIC controller's structure is reestablished and henceforth is ready to manage new failures. Figure 3-9 presents the configuration of a cluster from a normal situation until the recovery task has finished.

Recovering failed application processes

Table 3-4: Recovery activities performed by the each element implicated in a failure.

Protectors	Observers
Successor: 1) Fetches a new antecessor 2) Reestablishes the heartbeat mechanism 3) Commands the local observers to checkpoint	Survivors: 1) Establish a new protector 2) Take a checkpoint
Antecessor : 1) Waits for a new successor 2) Reestablishes the watchdog mechanism 3) Recovers the failed processes	Recovered: 1) Establish a new protector 2) Copy current checkpoint and message log to the new protector 3) Replays message from the message-log

The protector that is the antecessor of the failed node recovers the failed application processes in the same node in which the protector is running. Immediately after the recovery, each observer connects to a new protector. This new protector is the antecessor of the node in which the observer recovers. The recovered observer gets the information about its new protector from the protector in its local node.

Indeed, the protector of any observer is always the antecessor of the node in which the observer is running.

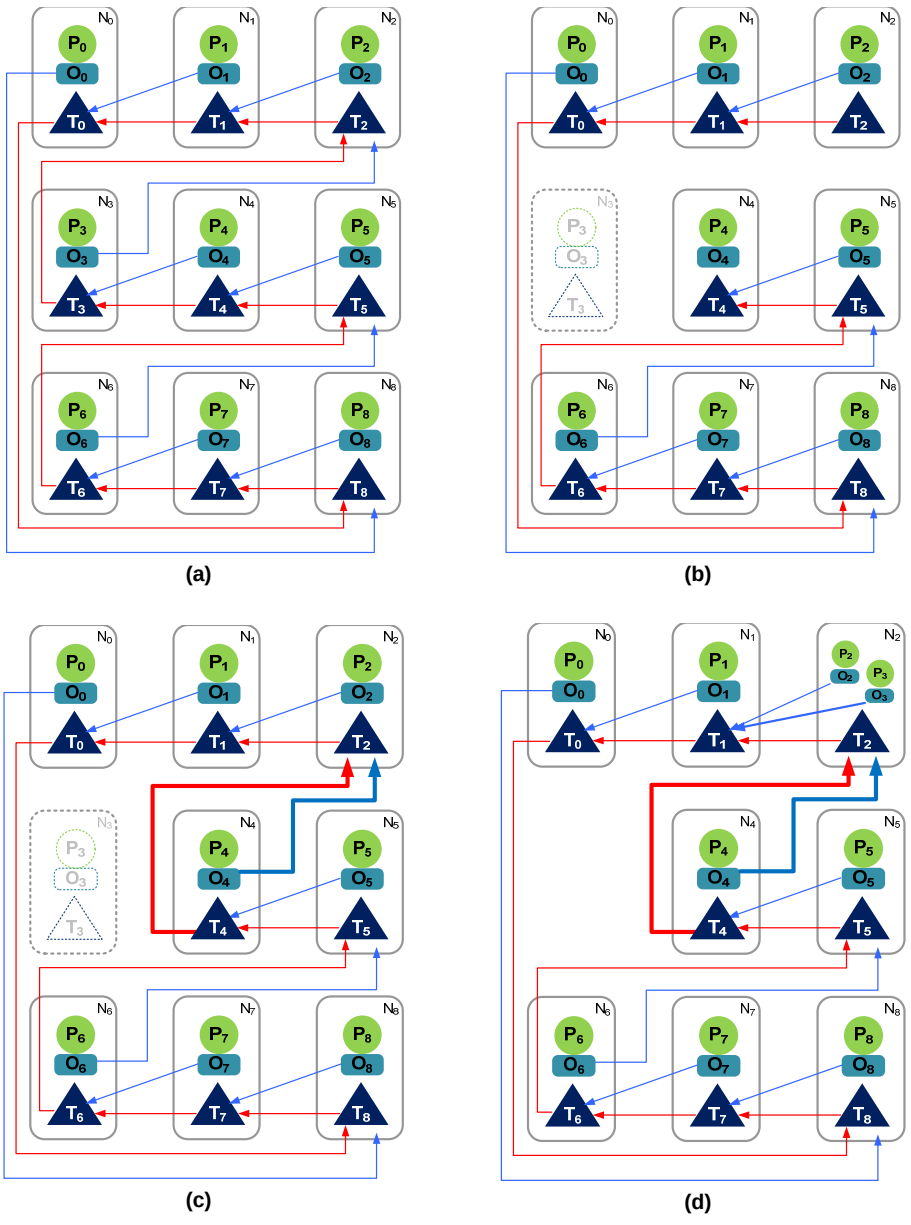

Figure 3-9: Recovering tasks in a cluster. (a) Failure free cluster. (b) Fault in node N3. (c) Protectors T2 and T4 detect the failure and reestablish the chain, O4 connects to T2. (d) T2 recovers P3/O3 and O3 connects to T1.

Load balance after recovering from faults

After recovering, the recovered process is running in the same node of its former protector. It means that the computational load increases in such node, because it now contains its original application processes plus the recovered processes. Therefore, the original load balancing of the system changes. This configuration change may imply in system degradation, resulting in performance loss in some cases. Moreover, after recovering, the memory usage in the node hosting the recovered process will rise leading to disk swap in some cases.

RADIC make possible the implementation of several strategies to face the load balance problem after process recovery. A possible strategy is to implement a heuristic for load balance that could search a node with lesser computational load. Therefore, instead of recovering the faulty process in its own node, a protector could send the checkpoint and the message logs of the faulty processes to be recovered by a protector in a node with less computational load.

3.3.5 Fault masking task

The fault masking is an observers' attribution. The observers assure that the processes continue to correctly communicate through the message-passing mechanism, i.e., the observers create a virtual machine in which failures does not affect the message-passing mechanism.

In order to perform this task, each observer manages all messages sent and received by its process. An observer maintains, in its private *radictable*, the address of all logical processes or the parallel application associated with their respective protectors. Using the information in its *radictable*, each observer uses the search algorithm, explained in sub-item *The radictable and the search algorithm* at the item 3.3.3, to locate the recovered processes.

Similarly, each observer records a logical clock in order to classify all messages delivered between the processes. Using the logical clock, an observer easily manages messages sent by recovered processes.

Table 3-5 represents a typical *radictable* including the logical clocks. One can see that the observer that owns this table has received three messages from the process zero and has sent two messages to this process. Similarly, the process has received one message and sent one message to process three.

Table 3-5: The *radictable* of an observer in the cluster in Figure 3-3.

Process id.	Address	Protector (antecessor addr.)	Logical clock for sent messages	Logical clock for recev. messages
0	Node 0	Node 8	2	3
1	Node 1	Node 0	0	0
2	Node 2	Node 1	0	0
3	Node 3	Node 2	1	1
...	...	...	...	...

Locating recovered process

When a node fails, the antecessor neighbor of the faulty node - which executes the watchdog procedure and stores checkpoints and message-logs of the processes in the faulty node – detects the fail and starts the recovering procedure. Therefore, the faulty processes now restart their execution in the node of the antecessor, resuming since their last checkpoint.

In order to clarify the behavior of a recovered process, in Figure 3-10 we represent four nodes of Figure 3-3 and the final configuration after a failure in one of these nodes. The process P_3 that was originally in the faulty node N_3 is now running in the node N_2. Therefore, all other processes have to discover the new location of P_3.

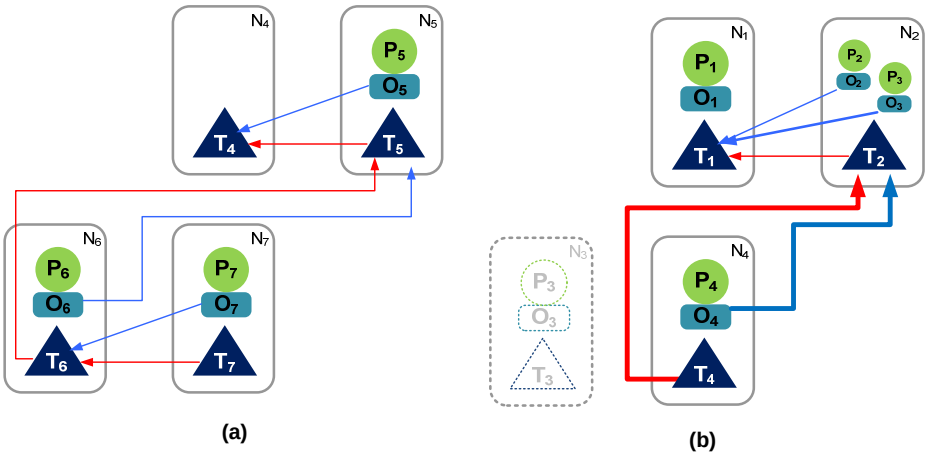

Figure 3-10: (a) A failure free cluster; (b) The same cluster after the management of a failure in node N_3.

In the explanation of the Fault Detection Phase, we defined two situations that create fake fault detection. The first situation occurs when an observer is taking a checkpoint of its application process, making this process unavailable to communicate. We described the solution for this problem in the Fault Detection Phase. Now, we describe the second situation and the solution for it.

After a node failure, all future communications to the processes in this node will fail. Therefore, whenever an observer tries to send a message to a process in a faulty node, this observer will detect a communication failure and start the algorithm to discover the new destination location.

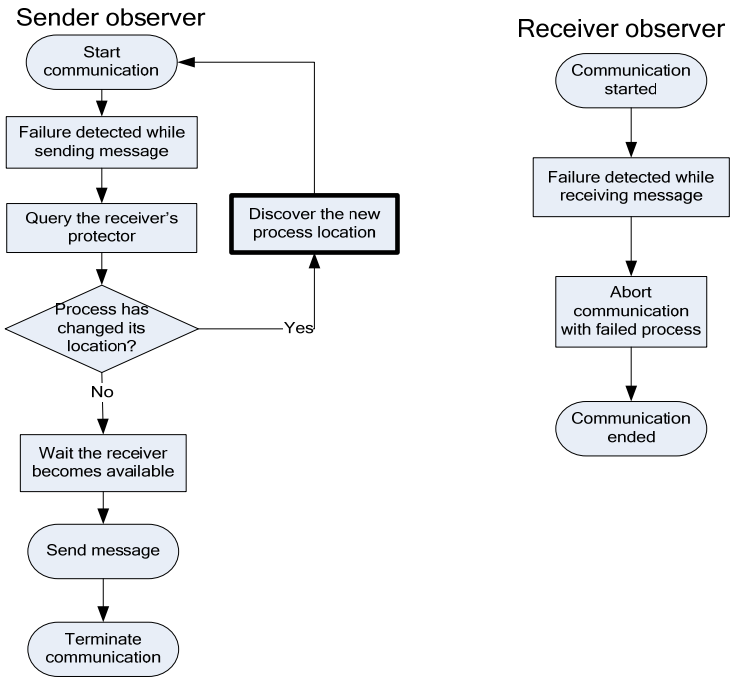

Figure 3-11: Fault detection algorithms for sender and receiver observers

Figure 3-11 describes the algorithms used by an observer if it acts as sender or as a receiver. An observer uses the search algorithm only the communication fails when it is sending a message to another process. If the failure occurs while the process is receiving a message, the observer simply aborts the communication because it knows that the faulty sender we restart the communication after it has recovered.

The search algorithm used by the sender observer uses the protector of the receiver process to inform the status of the receiver. However, if the receiver has recovered from a fault, its protector now is the antecessor of its original protector, because the recovered observer is now running in the same node of its original protector.

The example in Figure 3-10 clarifies the location of the recovered process P_3 after a failure in node N_3. The new protector of P_3 is now T_1, because P_3 currently is running in the same node of its original protector T_2.

If some observer tries to communicate with the faulty process P_3, such observer will obtain a communication error and will ask to the protector T_2 about the status of P_3. In this case, T_2 informs that it is not responsible for P_3 (because T_1 is now the current protector of P_3.)

In order to find who the current protector of P_3 is, the sender observer uses its *radictable* to follow the protector chain. The sender observer knows that if T_2 is no more protecting P_3, then the probable protector of P_3 shall be the antecessor of T_2 in the protector chain (because a faulty process always recover in the antecessor neighbor node).

Therefore, the sender observer reads its *radictable* and calculates the protector who is the antecessor of the protector T_2. In our example, the antecessor of the protector T_2 is the protector T_1. In the *radictable* the order of the protectors in the chain naturally follows the same order of the table index. Therefore, the antecessor of a node is always the node in the previous line of the table, as shown in Table 3-6.

Table 3-6: Part of the original *radictable* for the processes represented in Figure 3-10a.

Process identification	Address	Protector (antecessor address)
1	Node 1	Node 0
2	Node 2	Node 1
3	Node 3	Node 2
4	Node 4	Node 3

Now that the sender observer knows who the probable protector of the receiver process P_3 is, it contacts such protector and asks about the status of P_3. If the protector confirms the location of P3, the sender observer updates its *radictable* and restarts the communication process. Otherwise, the sender observer continues to follows the protection chain and asks for the next antecessor about P_3, until it finds where the process P_3 is.

In our example, the updated *radictable* of a process who tries to communicate with the recovered process P3 has the information presented in Table 3-7. In this table, the line three of the *radictable* (represent with bold font) represents the update location of process P_3 together with its new protector.

Table 3-7: Part of he updated *radictable* of a process that has tried to communicate with P3 after it has recovered as shown in Figure 3-10b.

Process identification	Address	Protector (antecessor address)
1	Node 1	Node 0
2	Node 2	Node 1
3	*Node 2*	*Node 1*
4	Node 4	Node 3

This process bases on the determinism of RADIC when recovering, which guarantees that the recovered process will be in the same node of its protector, allowing the explained heuristic. This heuristic will be change when we incorporate the dynamic redundancy, cause the spare node use may to generate a indeterminism when locating a failed process, once such process may recovers in any spare available.

Managing messages of recovered process

An application process recovers from its earlier checkpoint and resumes its execution from that point. If the process has received messages since its earlier checkpoint, those messages are in its current message log. The process' observer uses such message log to deliver the messages required by the recovered process.

If the recovered process resend messages during the recovery process, the destination observers discard such repeated messages. Such mechanism is simple to implement by using a logical clock. Each sender includes a logical time mark that identifies the message's sequence for the receiver. The receiver compares the time mark of the received message against the current time mark of the sender. If the received message is older than the current time mark from the specific sender, the receiver simply discards the message.

The observers discard the repeated messages received from recovered processes. However, a recovered process starts in a different node from the ones in which it was

before the failure. Therefore, it is necessary to make the observers capable to discover the recovered processes' location.

An observer starts the mechanism used to discover a process's location whenever a communication between two processes fails. Each observer involved in the communication uses the mechanism according to its role in the communication. If the observer is a receiver, it simply waits for the sender recovering.

On the other hand, if the observer is a sender it will have to search for the failed receiver in another node. The searching procedure starts by asking the receiver's status to the protector of the failed receiver. When the protector answers that the failed receiver is ready, the sender updates the location of the failed process and restart the communication.

3.4 RADIC functional parameters

The RADIC controller allows the setup of two time parameters: the checkpoint interval and the watchdog/heartbeat cycle.

To choose the optimal checkpoint interval is a difficult task. The interaction between the application and the checkpoints determines the enlargement of the application execution time. Using the interaction between the observers and the parallel application processes, the RADIC controller allows the implementation of any checkpoint interval policy. Each observer can calculate the optimal checkpoint interval by using a heuristic based in some local or distributed information. Furthermore, the observer may adjust the checkpoint interval during the process' execution.

The watchdog/heartbeat cycle, associated with the message latency, defines the sensitivity of the failure detection mechanism. When this cycle is short, the neighbors of the failed node will rapidly detect the failure and the recovery procedure will quickly start. However, a very short cycle is inconvenient because it increases the number of control messages and, consequently, the network overhead. Furthermore, short cycles also increase the system's sensibility regards the network latency.

The setting of the RADIC parameters, in order to achieve the best performance of the fault tolerance scheme, is strongly dependent of the application behavior. The application's computation-to-communication pattern plays a significant role in the interference of the fault-tolerant architecture on the parallel application's run time. For example, the amount and size of the messages directly define the interference of message log protocols.

3.5 RADIC flexibility

The impact of each parameter over the overall performance of the distributed parallel application strongly depends of the details of the specific RADIC implementation and the architecture of the parallel computer. Factors like network latency, network topology or storage bandwidth are extremely relevant when evaluating the way the fault-tolerant architecture affects the application.

The freedom to adjust of the fault tolerance parameters individually for each application process is one of the functional features that contribute to the flexibility of the RADIC architecture. Additionally, two features play an important role for the flexibility of RADIC: the ability to support concurrent failures and the structural flexibility.

3.5.1 Concurrent failures

In RADIC, a recover procedure is complete after the recovered process establishes a new protector, i.e., only after the recovered process has a new protector capable to recover it. In other words, the recover procedure is complete when the recovered process has done its first checkpoint in the new protector.

RADIC assumes that the protector that is recovering a failed process never fails before the recovery completion. We have argued in item that the probability of failure of an element involved in the recovery of a previous failure in other element is negligible. Nevertheless, the RADIC architecture allows the construction of an *N-protector* scheme in order to manage such situation.

In such scheme, each observer would transmit the process' checkpoints and the message logs to *N* different protectors. If a protector fails while it is recovering a failed application process, another protector would assume the recovering procedure.

For example, in the cluster of Figure 3-9, if the node N_2 fails before the recovery of P_3, the system will collapse. To solve this situation using a *2-protector* scheme, each observer should store the checkpoints and message-logs of its process in two protectors. In Figure 3-9, this would mean that O_3 should store the checkpoints and message-logs of P_3 in T_2 and in T_1. Therefore, T_1 will recover P_3 in case of a failure in T_2 while it is recovering the process P_3. During the recovery process, some election policy must be applied in order to decide the protector who will recover the failed process.

3.5.2 Structural flexibility

Another important feature of the RADIC architecture is the possibility of assuming different protection schemes. Such ability allows implementing different fault tolerance structures throughout the nodes, in addition to the classical single protectors' chain.

One example of the structural flexibility of RADIC is the possibility of clustering of protector's chain. In this case, the system would have several independent chains of protectors. Therefore, each individual chain would function like an individual RADIC controller and the traffic of fault tolerance information would be restricted to the elements of each chain. Figure 3-12 depicts an example of using two protectors' chains in our sample cluster.

In order to implement this feature is necessary to add one column to the radictable, the column that indicates the protector's chain. An observer uses the information in such column to search the protector of a faulty node inside each protectors' chain. The bold column in Table 3-8 exemplifies the chain information in a typical *radictable*.

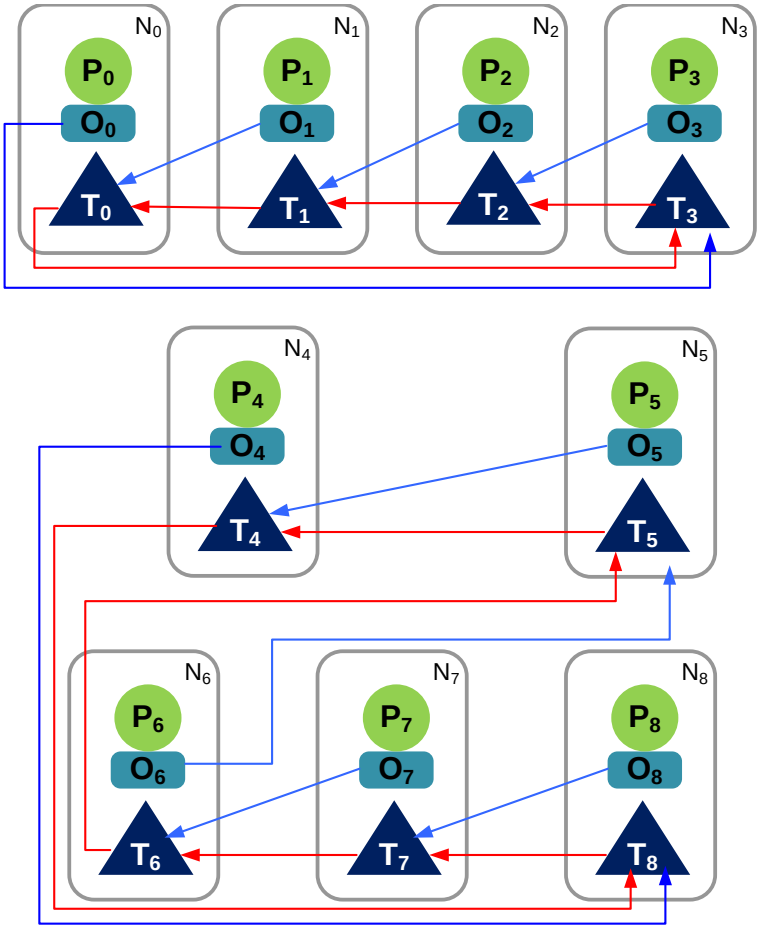

Figure 3-12: A cluster using two protectors' chain.

The RADIC architecture requires that, in order to manage at least one fault in the system, the minimum amount of protectors in a chain is four. This constraint occurs because each protector of the RADIC controller for fault tolerance requires two neighbors, an antecessor and a successor (see paragraph 3.2.1) Therefore, at least three nodes must compose a protector's chain. We depicted such minimal structure in Figure 3-13, in which each protector has an antecessor (to which it sends the heartbeats) and a successor (from which it receives heartbeats.)

Table 3-8: The *radictable* of an observer for a cluster protected by two protectors' chains like in Figure 3-12.

Process id.	Address	Protector (antecessor addr.)	Chain	Logical clock for sent messages	Logical clock for received messages
0	Node 0	Node 3	0	2	3
1	Node 1	Node 0	0	0	0
2	Node 2	Node 1	0	0	0
3	Node 3	Node 2	0	1	1
4	Node 4	Node 8	1	2	3
5	Node 5	Node 4	1	0	0
6	Node 6	Node 5	1	0	0
7	Node 7	Node 6	1	1	1
8	Node 8	Node 7	1	0	0

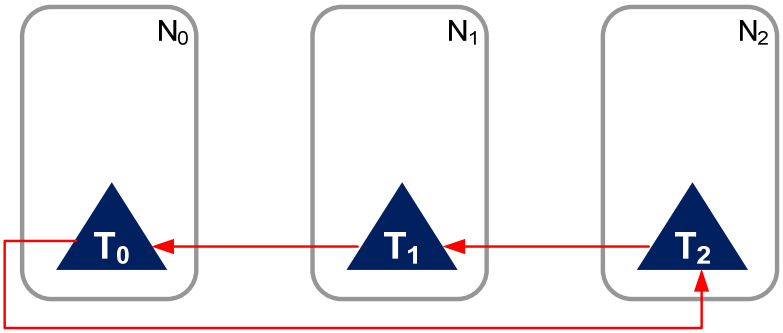

Figure 3-13: The minimum structure for a protectors' chain.

If we consider that a fault takes out a node of the chain, and that a chain with three nodes is not capable to handle any fault, it is easy to conclude that the minimum number of protectors in a chain defines the maximum number of faults that such chain can handle. Equation 2 expresses this relation; the maximum number of faults that a protector chain can handle is equal to the number of protectors in the chain minus three (the minimum number of protectors required to form a chain.)

$$MaxFaults = Number\_of\_Protectors - 3 \quad (2)$$

Chapter 4

Protecting the System

The last chapter explained about how RADIC is able to protect an application from the faults and to assure its correct finish. We saw the operational details when saving state, detecting faults and recovering a process.

During the recovery process, it was explained that RADIC, intending not using any extra resources, provoke a system configuration change that, in some cases, lefts the system in a undesirable situation.

In this chapter, we discuss about the side effects caused by the recovery process, and how these side effects may degrade the system, generating performance loss in some cases. This chapter also discusses about our solution in order to protect the system from these side effects, in other words, the system configuration changes that a recovery may cause.

4.1 Recovery Side-Effects

The fault tolerance activity incurs in some side effects in the cluster behavior, which vary according to kind of rollback-recovery protocol chosen, implementation details or architecture specifications. These effects generally incurs in some overhead in the application execution. Following, we will discuss about a specific side effect caused by some fault tolerant solutions.

4.1.1 System Configuration Changes

In order to provide fault tolerance to parallel machines, some rollback-recovery solutions does not demand any extra resources to perform its tasks. These solutions use the own cluster's nodes to execute the recovery procedures. When a failure occurs, some other node executing an application process is responsible to receive the

state data of the failed process, its checkpoint and log, and re-launch the failed process in this own node. Hence, more than the overhead imposed by the fault tolerance activity, basically the checkpoints and logs transmitting and storing [Rao *et al*, 2000], the post-recovery execution in these fault tolerant systems may be affected by this behaviour.

As said in chapter 1, some kind of applications demands a high computing power to perform satisfactorily its activities. This demand usually makes these applications use a parallel computer in order to achieve better results. Due to the application features, the user generally plans some process distribution over the cluster nodes aiming achieve the best performance possible. By example, the user may assign the process with more computing time in the node with more computing power, or allocate the process with high communication level in the closest nodes. This system configuration generally represents an optimal process distribution for the user idea, so, any changes in this configuration may represent a not desirable situation.

The RADIC architecture explained in Chapter 3 is an example of this kind of fault tolerant systems. As seen in item 3.3.4, the recovery process of RADIC leaves the system in an unplanned process distribution (Figure 4-11).

Following, we analyse an effect of this system configuration change over the performance in post-recovery executions.

4.1.2 Performance Degradation

The system configuration change explained in the last item leads to the presence of processes sharing a computing node. It is easy to perceive that in this node, both processes will suffer a slowdown in their executions and the memory usage in this node will be increased may leading to disk swap. Moreover, these processes will be accessing the same protector, competing by to send the redundancy data. Supposing that a previous process distribution was made, aiming to achieve a certain performance level according with the cluster characteristics, this condition becomes very undesirable, mainly if the application is not able to adapt itself to workload changes along the nodes.

Long time running programs are very susceptible to MTBF factor, the fault probability is constantly increasing during the time pass. In consequence, the number of overloaded nodes gradually increases, may leading to an impracticable situation. The Figure 2-1 depicts a nine nodes cluster in a situation after three recovered faults always occurred in the overloaded node. In this figure, we can see that in the node N_2 each process has at maximum 25% of the available node computing power. This may be an usual situation in clusters with thousands of nodes

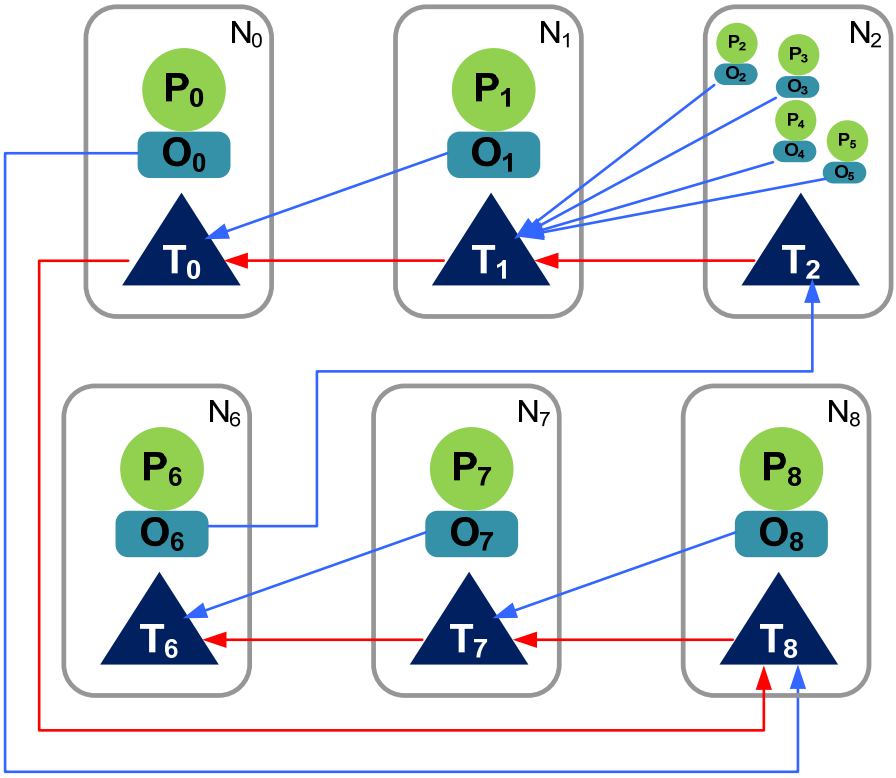

Figure 4-1: The cluster configuration after three sequential recovered failures.

The Figure 4-1 also depicts other problem caused by successive recovered faults: all the processes running in the node N_2 are storing their checkpoints and logs in the same neighbor, the node N_1. Checkpoints usually have large sizes in common scientific programs, and logs are very frequently in some kind of applications, consequently this happening may cause some situations:

a) The communication channel becomes a bottleneck due to the intensive traffic between the nodes

b) As each process is sending its checkpoint and log to the same protector, may occur a queuing of requests for checkpoint and log transmission in this protector.

c) The physical memory in the node N_2 is being consumed N+1 times more, where N is the number of unplanned processes running in the node divided by the number of original processes of the node. This fact may lead to use virtual memory on disk, which have a slower speed.

All of situations mentioned before, may engender a slowdown in all processes in the node, and may be combined between them, getting worse the system performance.

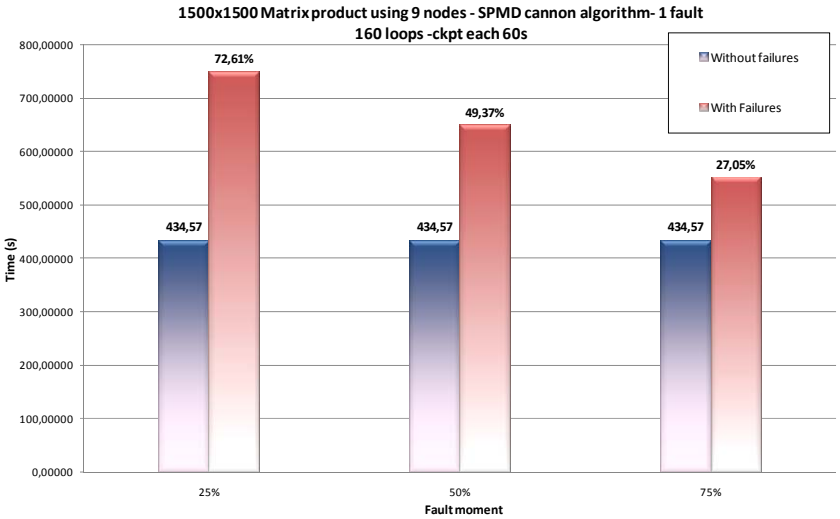

Figure 4-2: Execution times of a matrix product program implemented under the SPMD paradigm using a cannon algorithm. Occurrence of one failure per execution

Depending on the moment when the failure occurs, this disarrangement caused by the recovery process may affect the usability of the application results. For example in a weather prediction program, that deals with a lot of variables and has a well defined time limit to produce its results, that is before they really happens, a large delay cause the application produces obsolete results. Online (24x7) systems

will suffer gradual throughput degradation as failures occurs, generating response times that may become unacceptable.

An aggravation of this condition may occur in tightly coupled parallel systems where the communications between the processes is very intensive. Therefore, if some processes experience a slowdown in their execution, they will start to postpone their responses to other processes, then these ones will be held, waiting a message receive from the slow nodes, propagating the slowdown by the entire cluster. The chart in the Figure 4-2 shows the execution times of a SPMD implementation of a matrix multiplication (Cannon's algorithm). This SPMD algorithm has a communication mesh as depicted in Figure 4-3b.

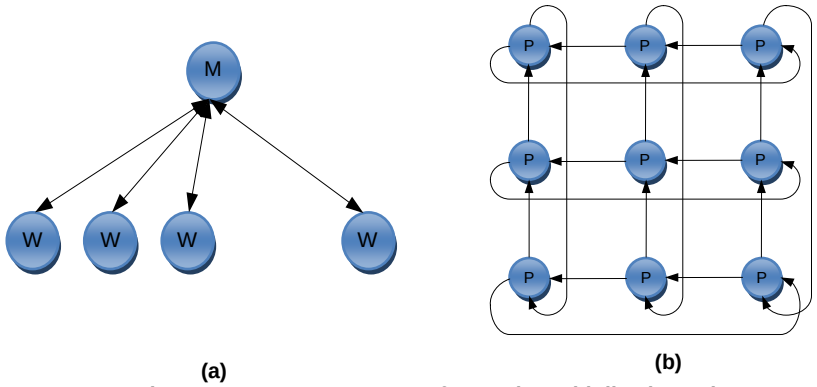

Figure 4-3: Message pattern of a matrix-multiplication using a) M/W paradigm and b) SPMD paradigm.

Each execution was performed in a nine nodes cluster and one fault was injected at different moments. In this chart, the greater bars indicate more execution time. We can see that having only one node sharing process causes considerable delays even when fault occurring near to the end of the processing. Comparatively, the Figure 4-4 depicts an analogue result with a master/work implementation of matrix multiplication. In this approach, the processing was distributed statically through the cluster nodes. As shown in Figure 4-3, the MW algorithm has a 1-to-N message pattern (Figure 4-3a). The master process communicates with all the worker processes. Each worker process only communicates with the master process. We can see that the effects of having nodes sharing the computing power of a node are very similar in different parallel paradigms.

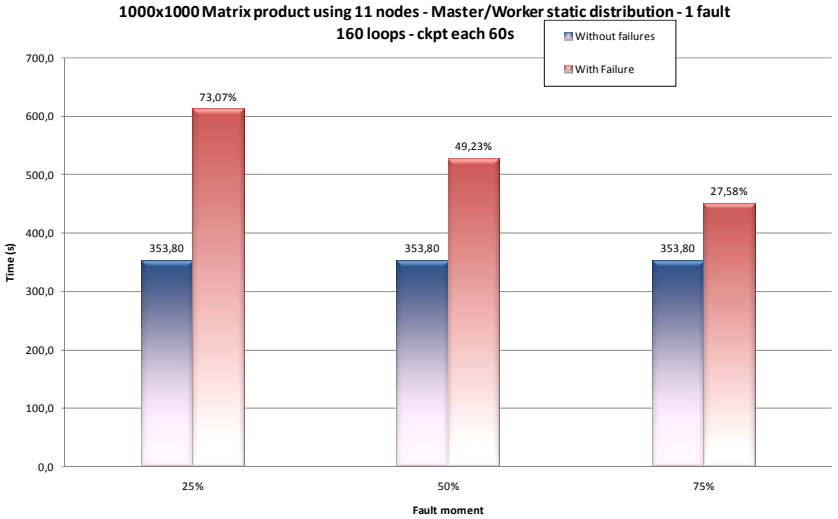

Figure 4-4: Execution times of a matrix product program implemented under the Master/Worker paradigm using a static distribution. Presence of one fault per execution

The factors exposed and the results showed in this chapter, demonstrate that the system configuration change may produce unwanted system slowdown. Those performance degradations may leave impracticable the use of some applications that have time constrictions or demands all computing power possible. Therefore, it is very desirable that the fault tolerance solutions avoid this phenomenon and more than protect just the application execution, also protect the system configuration from the possible changes.

4.2 Protecting the system

The item 4.1 explained about the side effects caused by the recovery process in some fault tolerant solutions. This item discusses about our solution in order to protect the system from these side effects, in other words, the system configuration changes that a recovery may cause.

Besides the high availability, the applications running in clusters of computers usually demands high performance. Indeed, recently studies [Nagajara *et al*. 2005] has demonstrated the relationship between these two requirements, retrieving the "*performability"* concept formally introduced by Meyer [Meyer, 1980], which takes

into consideration that in parallel computers, as a degradable system, performance and availability cannot be dissociated and the overall performance of some systems is very dependable of its availability.

With the *performability* concept in mind, we developed RADIC II, a new version of the RADIC architecture that, beyond guaranteeing the correct finish of the applications, protects the system from the performance degradation caused by fault recovery, allowing preserving the planned process distribution (system configuration) in order to conciliate performance and availability.

We increased the RADIC flexibility providing a dynamic redundancy feature that protects the system configuration from the possible changes imposed by the recovery process. The dynamic redundancy in opposite to static approach, bases on presence of backup components ready to assume the work of a failed one. These backup components also are called spares, if they are active, but not working, may be called hot spares.

This new feature introduces a fully transparent management of hot spare nodes in the RADIC architecture. The major challenge in this new RADIC architecture version is keep all the RADIC features and provide a mechanism to use and manage spare nodes in a fully distributed system. This mechanism was implemented in three different approaches, each one incorporating a new functionality:

a) Starting the application execution with a pre-allocated number of spare nodes. The spare nodes are being used as faults are occurring until reach zero.

b) New spares insertion, in order to replace the consumed ones. During the application execution, spare nodes are consumed as needed, this approach allows reestablish the planned number of spares in the system. This approach also is useful to replace failed nodes when there are not spares in the configuration.

c) Replacing the most fault probable nodes (due to factors like MTBF) before the fault occurs. This approach is useful for maintaining purposes. It is

possible to replace each node of a cluster without need to stop the entire application. It is applicable in long running applications.

Such mechanism improves the RADIC *perfomability* without to affect its four major characteristics: transparency, decentralization, flexibility and scalability. Following we explain in detail each one of these approaches, showing how it works.

4.2.1 Avoiding System Changes

In this approach, we provide a system that avoids the system configuration change mentioned in item 4.1.1 by providing a set of spare nodes used to assume failed processes, instead of to recover in some working node. A RADIC II configuration may have any spare nodes as desired. Each spare node runs a protector process in a *spare mode*.

Such approach aims to allow controlling the system degradation generated by the RADIC recovery process, once we avoid the node loss by replacing it by a spare node. We preserve the original flexibility allowing many spares as desired, without to affect the scalability feature once the spares does not participate of the fault tolerant activities excepting, of course, the recovery task. The RADIC transparency is kept by a management scheme that does not need any administrator intervention and keeps decentralized all information about the spares.

In this mode, the protector does not perform the regular tasks described in item 3.2.1, just staying in listening state waiting for some request. The Figure 4-5 depicts a RADIC II configuration using two spare nodes, in this figure the nodes N_9 and N_{10} are spare nodes protectors denoted by gray color, we can see that these protectors does not participate of the protectors' chain, avoiding generating the failure detection overhead with a workless node.

Each active Protector maintains the information about the spares presence. This information is stored in a structure called *spare table*. The excerpt in Table 4-1 shows the *spare table* structure: in the first column is the spare identification according with the same protector's identification. The second field is the physical address of the

spare node. Finally, the third column indicates the number of observers (processes) running on this spare; this field is useful to indicate if the spare is still in an idle state.

Table 4-1: The *spare table* of each observer in the cluster in Figure 3-3.

Spare identification	Address	Observers
9	Node 9	0
10	Node 10	1
. .	. .	. .

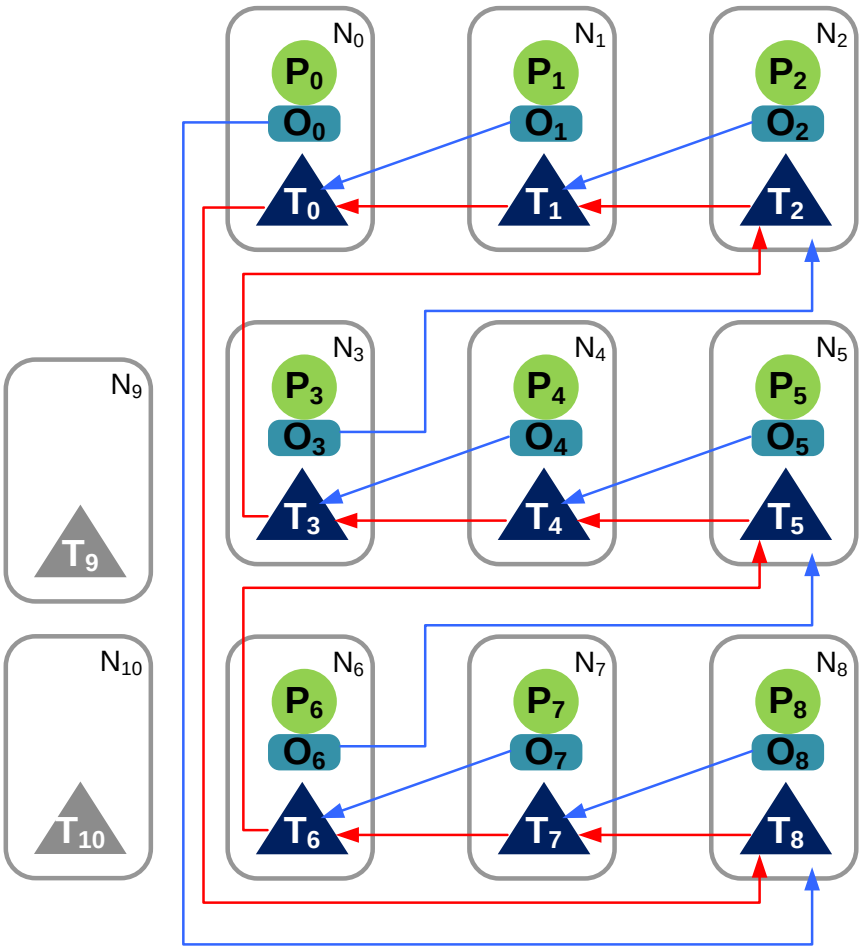

Figure 4-5: A cluster using the RADIC II architecture with two spare nodes (N_9 and N_{10}).

How the active protectors detects the spare nodes

In order to keep RADIC II as a decentralized system, the spare nodes must spread their existence for all active nodes of the cluster. To achieve this requirement, the protector, when starts in spare mode, announces itself to the other protectors through a reliable broadcast basing in the message forwarding technique [Jalote, 1994, p. 142]. This technique was chosen because does not affects the original RADIC scalability.

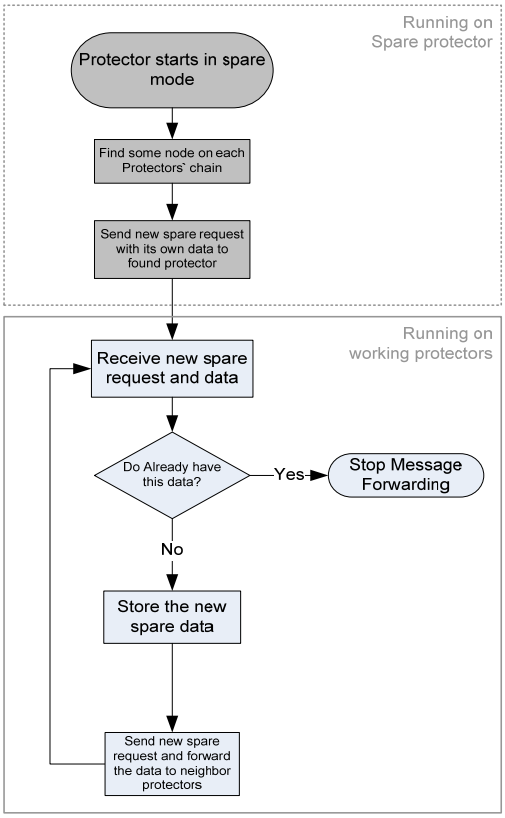

Figure 4-6: How a protector in spare mode announces itself to the other protectors

The protector when running in spare mode searches some active node in each protector's chain running with the application and starts a communication protocol with him requesting for its addition in the protectors' *spare table*. The working protector that receives this request, searches if the new spare data already is on its *spare table*. If not, this protector adds the new spare data and forwards this request to its neighbors, passing the new spare information in sequence. Each protector performs the same tasks until to receive an already existent spare node data, finalizing the message forward process. The flow in the Figure 4-6 clarifies this announcement procedure.

This procedure occurs before the application starts, while RADIC is mounting its *radictable* and just after has been started the Protectors (see item 3.3.1), hence it is a

latency caused by the initialization process, does not considered as overhead in the execution time. At the end of the spare nodes announcement, all the protectors have a spare list containing the data of all spares available.

4.2.2 Recovering Using Spare Nodes

In the RADIC II, we modified the original RADIC recovery task described in the item 3.3.4 in order to contemplate the spare node use. Currently, when a protector detects a fault, it first searches a spare data in its spare table, if found some unoccupied spare, i.e. the number of observers reported in the spare table still is equals to zero, it starts a spare use protocol. In this protocol, the active protector communicates with the spare asking for its situation, i.e. how many observers are running on its node. At this point, two situations may happen:

a) If the spare answer that already has processes running on its node, the protector then search other spare in its table and restart the procedure. If the protector does not find any idle spare, it executes the regular RADIC recovery task.

b) If the protector confirms the idle situation, the protector then sends a request for its use.

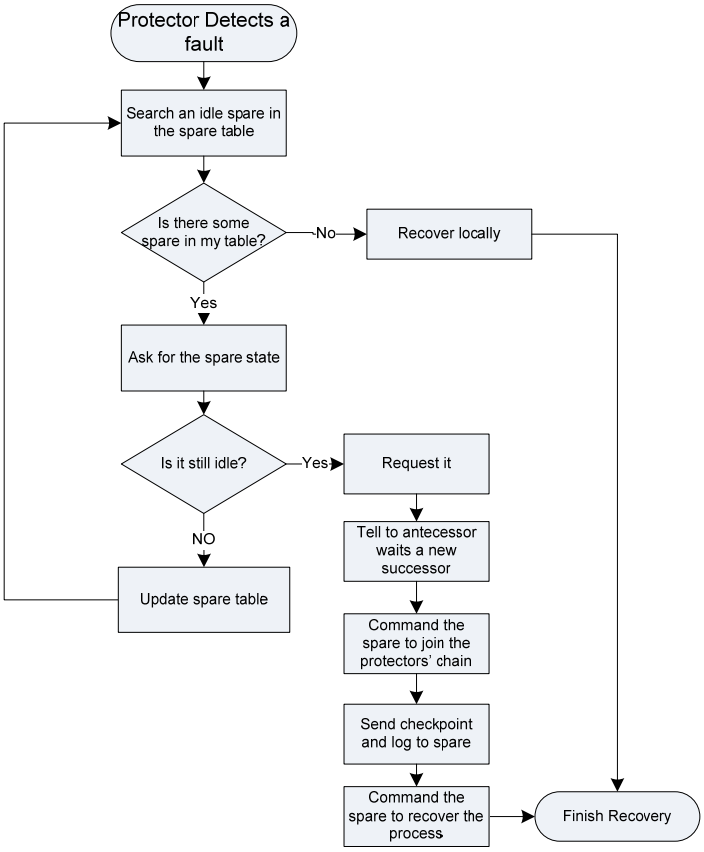

Figure 4-7: The recovery task using spare nodes

From the request moment, the spare will not accept any requests from other protectors. After receive the request confirmation, the protector commands the spare to join the protectors' fault detection scheme. This step consists in these phases:

a) The protector tells to its antecessor in the chain to wait the connection of the spare to be its new successor.

b) Simultaneously, the protector commands to spare to connect to this antecessor and make it its own antecessor.

c) The protector instructs the spare to wait its connection as its new successor.

d) Finally, the protector connects to spare as its new antecessor.

Figure 4-8: Recovering tasks in a cluster using spare nodes.

After finishes this step, the protector sends the failed process checkpoint and log to the spare, and command it to recover the failed process using the regular RADIC recovery process. The flow in the Figure 4-7 clarifies this entire process, complementing the understanding of this process.

The Figure 4-8 depicts the system configuration in four stages of the recovery task: a) Failure free execution with presence of spare nodes; b) a fault occurs the node N_3; c) the protector T_4 starts the recovery activating the spare N_9; d) process recovered in the spare node.

Changes in the Fault-Masking Task

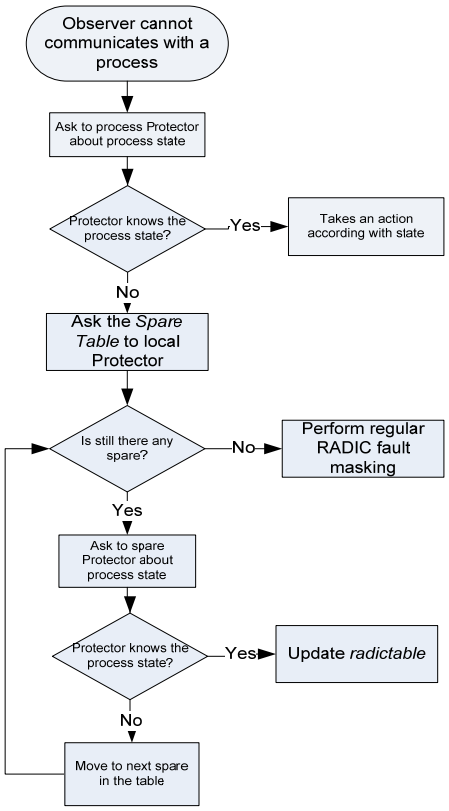

Figure 4-9: The new fault mask procedure

The original RADIC fault-masking task presented in the item 3.3.5 bases on a heuristic to determine where a faulty process will be after the recovery. This heuristic was very efficient in RADIC, because its recovery process was very deterministic, i.e. the failed process always recovers in its protector. RADIC II inserts a small indeterminism in locating a failed process because this process may has been recovered in any spare of the configuration, and the process may not able to locate the recovered process in this spare.

In order to solve this indeterminism, we implemented a small change in the original RADIC fault-masking task. This change consists in if after the observer did not find the recovered process asking by its original protector, it searches in the spares using spare table, looking for the faulty process. However, as we said in the topic *How the active protectors detects the spare nodes,* only the protectors have the spare table information, not the observers. Hence, we increased the communication between the Observers and the Protector running in its node, the *local protector***.** In order to execute the new fault masking protocol, the Observer communicates with the local Protector, asking for the spare table, and seeks in this table for the recovered process. After the Observer has found the recovered process, it updates its *radictable* with the new location, does not needing to perform this procedure again. The Figure 4-9 contains the flow of this new procedure.

4.2.3 Restoring the System Configuration

Until now, we saw how the proposed RADIC II mechanism avoids the system configuration changes. Most of the concepts presented will be also applied in this approach.

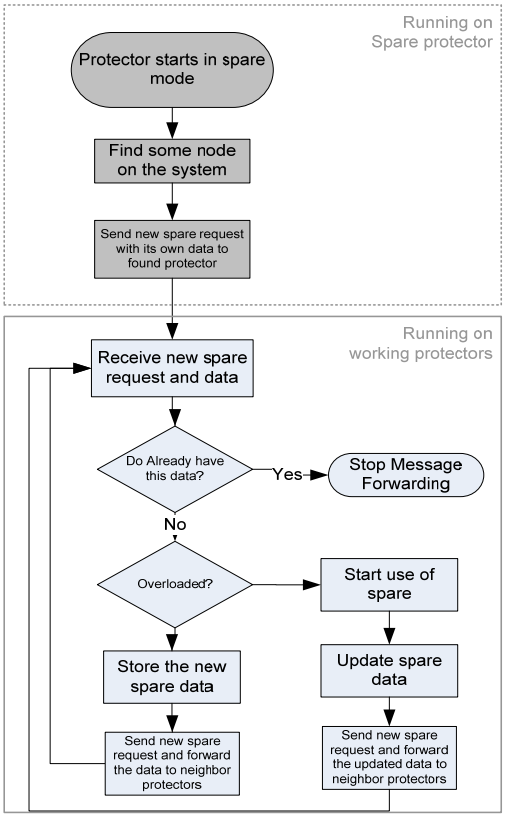

Figure 4-10: How a spare is used to replace a faulty node

Let us suppose a long-term execution application, like the 24x7 systems. This system usually needs a fault tolerant solution based in redundancy in order to avoid the degradation of the system, keeping the *performability*. If it using a dynamic redundancy solution provided by spare components, the system will be able to keep the performance by a certain period, but as the number of spares is a fixed value, they are used in each fault until reach zero. From this moment, this system starts to suffer some degradation after fault recoveries.

The flexibility of RADIC II allows avoiding this happening by restoring the initial system configuration. The procedure explained in topic *How the active protectors detects the spare nodes* may be applied to insert new nodes in order to replace the used spare nodes. Despite the process described in this topic be performed at the start of an application, it may be executed at any moment during the program execution, without need stop the application. Thus, the failed nodes can return to the configuration after be repaired.

More than to replace used spare nodes, that procedure may replace faulty nodes too. We extended the announcement task to permit that if some node of the configuration is already overloaded, i.e. it has more processes executing than the original planning, it can request the spare at the announcement moment, transferring the extra processes to the inserted node. If there is not any overloaded node, this new node remains just a new spare in the configuration.

In the Figure 4-10 we show the flow contemplating this approach. Some election policy may be applied in order to choice what node will migrate its extra processes to the new node, i.e. the first one found or the most overloaded. After initiate the use of the new node, the protector update the spare data informing that this spare already is in use and continues the spare spreading procedure, thus all protectors will know of the existence of this node.

In this approach, we have a limitation, which occurs in the case of all of the nodes already have been replaced. In this situation, the new spare has not how to discover any node of the application without some extra information provided at the start of the procedure.

4.2.4 Avoiding Faults

Despite the availability, the major goal of continuous applications using high performance computing cluster is to obtain faster response times. It is a common sense that the fault avoidance is a better approach than the fault tolerance. The usual manner to avoid faults occurrence bases on a preventive maintenance of the system components.

Preventive maintenance generally involves the periodic replacement of the fault-imminent or fault-probable components. This procedure generally means to stop the component or its host. However, the referred kind of applications doesn't expect maintenance downtimes, implying in the existence of a mechanism that allows these replacements without need stop the execution of the application.

The dynamic redundancy scheme present in RADIC II, explained above, provides a scheme that enable the architecture to perform a scheduled hot replacement (not stopping the application execution) of a cluster node. As RADIC II allows the spare node insertion after the application starts without requiring any stop in the program execution, we just need turn off the node to be replaced and their processes automatically will be recovered in the recent spare added.

Such mechanism is very simple, and may be improved by implementing some automation feature that allows commanding the machine to be replaced to automatically turn off, or to take it checkpoint directly in to new spare added, just before to suicide. Other improvement may be the inclusion of some fault prediction algorithm that permits a best choice of the machines to be replaced.

Chapter 5

Implementing the RADIC II Architecture

In order to prove and to perform experimental evaluation, Duarte implemented a prototype of RADIC architecture called RADICMPI [Duarte *et al,* 2006]. RADICMPI implements a set of the Message Passing Interface (MPI), a widely used specification for message passing implementations.

Since our solution bases on the RADIC architecture, we also used the RADICMPI prototype to implement the dynamic redundancy feature. We performed some modifications in the original program in order to allow the transparent management of the spare nodes. By the other hand, we also incremented the set of MPI functions implemented in RADICMPI with the nonblocking functions, as part of a continuous work to achieve a complete MPI implementation, which allow us to perform a variety of experiments.

This chapter shows how we made a practical implementation of the RADIC II architecture over the existent RADIC prototype.

5.1 RADICMPI

The functional validation and all of RADIC experiments was conducted using a prototype implementation called RADICMPI. The MPI specification was chosen because is a widely adopted standard for message passing and have a fail-stop semantic that demands some fault tolerance support.

Following we list the RADICMPI major features:

a) It relies in open source software that allow an improved control of its components.

b) It is a multithread program written in the C++ language and running on Linux operating system for i386 architecture.

c) All network communication bases in the TCP/IP protocol

d) The checkpoint/recovery functions are performed by the well known library BLCR (Berkeley Labs Checkpoint/Restart) [Hargrove and Duell, 2006].

The RADICMPI prototype system is compounded by three parts:

a) A set of shell scripts: *radiccc* - It used to compile RADIC MPI compliant programs, indeed it is a convenience wrapper for the local native C and C++ compilers including the RADICMPI library; *radicrun* – It launches a RADIC application, it receives the RADIC parameters and after parse them, bypass them to other RADICMPI components.

b) The Protector program – It is a standalone program that implements the RADIC Protector concept.

c) The RADICMPI Library - is a multithread library. Three threads exist when we execute a program compiled with the RADICMPI library: the program main thread, the observer thread and the checkpoint thread

RADICMPI has a built-in failure injector system that allows simulating faults driven by pre-programmed events like passed time or number of messages. This failure injector also permits determine which machine will fail. It also provides a event logger that performs a detailed monitoring of the execution either of the observers or of the protectors. Finally, RADICMPI allows a time instrumentation of specific events embedded in its code allowing measuring the performance of determined tasks.

Actually, RADICMPI Library implements a set of the MPI-1 compliant functions: the blocking communication functions: MPI_Send, MPI_Recv, MPI_Sendrecv; the initialization and finalization functions MPI_Init, MPI_Finalize;

and a group of auxiliary functions: MPI_Comm_rank, MPI_Comm_size, MPI_Wtime, MPI_Type_size and MPI_Get_processor_name.

The observer thread in the RADICMPI Library, as responsible by processes communications, uses an internal buffer to deal with the arrived messages. The incoming message process of RADICMPI works through this continuing running thread, thus, the reception of a message is not associated with an MPI_Recv command. An observer is always ready to receive messages, unless it is performing a checkpoint of its process.

The Protector program performs the node protection activities and the distributed storage tasks. It owns two entities related with its functionalities: The Antecessor executing the watchdog thread, responsible to detect the neighbor failure and to manage the neighbor's observers, receiving their checkpoint and logs; the Successor performing the heartbeat thread indicating when the node is alive.

5.2 The New Spare Nodes Feature

In order to enable the RADICMPI implementation for the dynamic redundancy described in the RADIC II architecture, we performed some modifications in its code. These modifications are mainly implemented in the Protector program's code, which is commonly associated with node tasks.

The Spare Mode

The first modification was the creation of the spare operation mode. The Protector now receives a parameter indicating when it is a spare. In this situation, the Protector starts neither the heartbeat/watchdog functions nor the observers' management.

The Protector now receives as parameter the name of the node that it must to connect to inform its presence. This node name is given either by the *radicrun* script when the spare starts with the application or by the new *radicadd* script, responsible to add new spares during the application execution. This script takes a machine name in the list of application nodes and runs then Protector in spare mode passing such

name as parameter. The Protector now tries to connect with this node, if fails returns an error code to script, which takes other machine name from the list and repeats the process. If the Protector gets connected, it requests its addition in the spare table, starting the spreading of its data. The flow in the Figure 5-1 makes this procedure clearer.

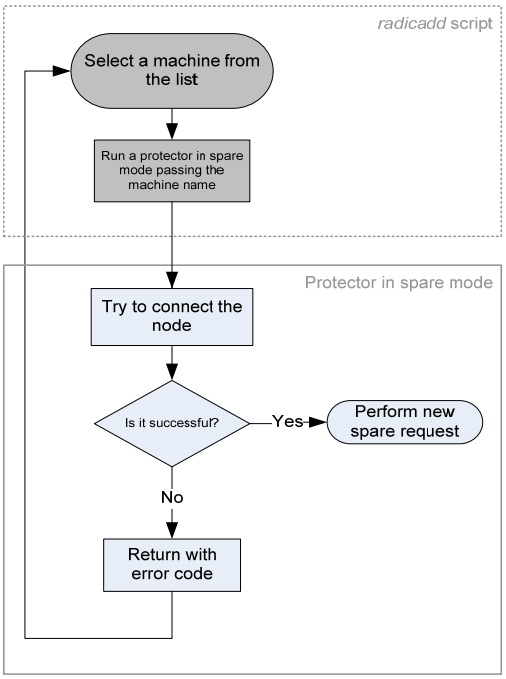

Figure 5-1: How a Protector in spare mode discover and connects with an application node

Furthermore, we implemented a new function in the Protector that is in charge to deal with management requests.

Management Functions

In order to enable the Protector to deal with the new features incorporated by RADIC II, we create a set of management functions and a management thread to attend the requests. When the Protector starts in spare mode, it only starts the management thread. This thread opens a TCP socket listening for any request from the external world made by other Protector, an Observer or a Monitoring program.

When a connection arrives, the management thread analyzes the request and starts a specialized function to attend it performing needed tasks. The request is a message containing a header indicating its type, and generally, some other information needed to perform it. Following we list the request types and its functionality.

USESPARE – Request the use of the spare to recover a failed process. The protector that detects the fault, send this request to spare. When the spare Protector receives this request, performs the tasks to join the protector's chain, to receive and to recover the failed process.

CNCTPROT – It commands the Protector to connect with other one. The request contains the Protector address to be connected. This request makes part of the tasks performed when recovering a failed process.

NEWSPARE – It is the request for insert the information of a new spare in the spare table. The new spare sends this request to a Protector and this Protector, after adding this information, sends the same request to its neighbor, until receive back the same request, finalizing the new spare addition process.

OBSQTY – When a Protector (spare or not) receives this request, it answer with the number of observers running in this node. This information is useful to determine when a spare still remain idle and can be requested for recovering. A Protector sends this request to discover the idle spares.

MLOGTX – This request is inherited from the Observer-Protector protocol used to perform the regular RADIC recovery process. It is a request for the log file transmission. When the spare receives this request, waits for the log file reception, and store it for the recovery process.

CKPTTX – This request is also inherited from the Observer-Protector protocol used to perform the regular RADIC recovery process. It is a request for the checkpoint file transmission. When the spare receives this request, waits for the checkpoint file reception, and store it for the recovery process.

SNDSPARETBL – It asks to Protector to send its spare table.

SNDSPARETBLOBS – It is similar to the previous, but in this case, the Observers who sent this kind of request. This request is useful in the fault-masking task performed by the Observers and they need to know where the spares in the actual configuration are in order to find some recovered process.

RCVSPARETBL – It is a request that tells to the Protector to stay waiting for a spare table transmission, i.e. to get ready for to receive a spare table. A Protector sends this request when the spare assumes a failed process, because at this moment the spare still not know who the other spares are.

SNDINFONGHB – It asks to Protector to inform who its neighbors are. It is useful to a monitoring tool in order to discover the actual protectors' chain.

RCVROBS – This is the command to the spare to recover the Observer using the checkpoint and log transmitted before. The data of the Observer goes just after the request.

STOPTHREAD – It is an operational request. Applied when the application has finished and the RADIC II environment is shutting down. It performs the finish of the management thread.

STOPRCV – It commands to Protector to stop the management session, backing to wait a new session.

How RADICMPI Recovers a Process Using a Spare Node

When a Protector detects a fault, it starts a recovery process using the management commands mentioned before. Following, we describe the recovery process using a spare node.

The Protector searches in its spare table if there is some spare. If not, it performs the regular RADIC recovery process. If there is some spare in the table, the protector sends to it a OBSQTY request, if the response is greater than zero, it will be taken the next spare in the table and repeat this step. If does not remains any spare in the table, it performs the regular RADIC recovery process. If the spare answers that have zero observers, the Protector assumes a commander function in relation to the spare and sends a USESPARE request. The spare then starts a *usespare* routine not attending

requests from any other Protector. The commander Protector then sends a CNCTPROT request in order to the spare join itself to the Protector's chain. The commander sends a CKPTTX request, and transmits the checkpoint file, following sends a MLOGTX if there is any log file and send it to the spare. After the files transmition, the Protector sends a RCVROBS request and the spare performs the regular RADIC recovery. Finally, the commander sends a STOPRCV to stop the recovery session. The Figure 5-2 depicts the message exchange between Protector and spare.

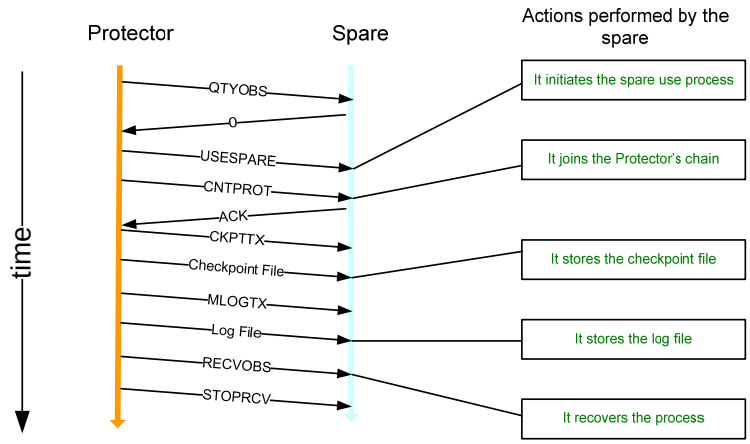

Figure 5-2: The recovery RADICMPI protocol using spare.

Changes in the Fault Masking Routine

After a successful recovery procedure in the spare, the other processes in the application might not know that where the recovered process is running. As the old RADICMPI recovery routine was deterministic, i.e. the process always recover in its Protector (a neighbour), the fault masking routine was based in to search back in each Protector of the chain. At present, the recovered process location is a quite undetermined, because may be located in any of the spares of the configuration, or in its original Protector either. In order to solve this question, we changed the original fault masking routine, making it to search the recovered process in the spare table if didn't find in the original protector. For in such a way, the Observer attached to the process locating the recovered one must to know the spares in the actual

configuration, and then it starts a communication with the local Protector, i.e. the Protector running in the same node, and send a SNDSPARETBLOBS request. The local Protector sends its spare table to Observer. The Observer will ask to each Protector' spare in the table for its status. If after this routine the recovered process is not found, it means that the process was recovered in the original way, so the Observer starts the original fault masking routine. After the Observer has found the recovered process, it updates its radictable with the new location, does not needing to perform this procedure again.

5.3 New MPI Functions

In order to improve our experimental work, increasing the number of possible experiments and contribute with the RADICMPI completeness, we performed an additional work, adding some new MPI functions. All of these functions was designed and implemented taking care with the fault tolerance issues. Following we describe these functions.

5.3.1 The Non-Blocking Functions

We believe that an important step after to have the major blocking functions implemented is to have available the non-blocking MPI functions. These functions perform communication without need to block the processing until the message buffer is free. This behavior allows overlapping computation with communications, i.e. while the communication request is not finished, some computation may be performed. We implemented the functions: MPI_Isend, MPI_Irecv, MPI_Test, MPI_Testany, MPI_Testall, MPI_Wait, MPI_Waitany, MPI_Waitall and the object MPI_Request.

MPI_Request

MPI_Request is an object used to control the requests of non-blocking functions. Every time that a non blocking function starts, it creates a MPI_Request object, when the function completes its work, it updates this object.

MPI_Isend

MPI_Isend commands the Observer to use a thread to send the message stored in a buffer informed by the application process. At this moment, the function creates a MPI_Request object that is returned to application. The processing continues despite the observer not has delivered the message to the communication channel yet. When the message is delivered, the MPI_Request object has the information of its completion set to true.

MPI_Irecv

The MPI_Irecv drives the Observer to check in the RADICMPI buffer if a requested message was delivered. If not, it puts such message in a pending message buffer, and let the processing continues, setting the MPI_Request completion field to false. Every time that a message arrives, the Observer checks in the pending message buffer if there is a correspondent message request, if it finds the message, it sets the MPI_Request completion field to true.

MPI_Test

The MPI_Test receives an MPI_Request object and probes its completion returning the result of the probe. Moreover if the request command was a MPI_Irecv, it sets the MPI_Status referred with this command.

MPI_Testany

MPI_Testany performs a MPI_Test function in a set of MPI_Request objects returning true if any of them is complete. Moreover it also sets the MPI_Status of the completed command in the MPI_Irecv cases.

MPI_Testall

MPI_Testall performs a MPI_Test function in a set of MPI_Request objects returning true if all of them are complete. Moreover it also sets the MPI_Status of the completed commands in the MPI_Irecv cases. Indeed, the MPI_Test is a special case of this function, where the set of requests only has one entry.

MPI_Wait

The MPI_Wait receives an MPI_Request object, probes its completion, and blocks the processing until the request finishes. In the MPI_Isend case, waits that the started thread signalizes its finish through a lock variable. The MPI_Irecv cases relies in the arriving of messages that signalize the checking of pending messages routine.

MPI_Waitany

MPI_Waitany performs probe in a set of MPI_Request objects waiting until any of them is complete. Moreover it also sets the MPI_Status of the completed command in the MPI_Irecv cases.

MPI_Waitall

MPI_Waitall performs a MPI_Wait function in a set of MPI_Request objects waiting until all of them are complete. Moreover it also sets the MPI_Status of the completed commands in the MPI_Irecv cases. Indeed, the MPI_Wait is a special case of this function, where the set of requests only has one entry.

5.3.2 The Collective Functions

In order to carry out a specific experiment, we just implemented the MPI_Allgather function, performed with MPI_Sends and MPI_Recvs commands. This function sends a buffer in each process for all of the other process. After performing, each process has a vector containing the buffer value of each process executing.

Chapter 6

RADIC II Experiments

After to design and to implement the RADIC II features, the next step in our work certainly is to test our idea through a set of experiments that allows to validate and to evaluate our solution.

In order to perform this activity we made a plan of experiments, taking into consideration the aspects that we would to analyze, and the expected results. This plan of experiments contains the kind of experiment, how to perform it, objective and results expected versus results obtained.

Our experiments not intended to prove or to evaluate the RADIC functionalities. We believe that these functionalities and features are already proved [Duarte*, et al.*, 2007] and RADIC perform its tasks with correctness, producing valid results. We stated our experiments in the flexible dynamic redundancy feature, validating its functioning and evaluating comparatively its results. All of our generated results were compared with the same program compiled and executed with MPICH-1.2.7, guaranteeing the correctness of the execution.

6.1 Experiments Environment

In order to proceed with our experiments, we used a cluster with the following characteristics: twelve Athlon-XP2600+/ 1.9GHz/ 512KB L2 cache, 768 MB RAM, 40GB ATA running Linux Kernel 2.6.17 with gcc v4.0.2 compiler. An Ethernet 100-baseTX switch interconnects all of nodes. The network protocol used was TCP/IP v4.

All executions were performed using the RADICMPI prototype. RADICMPI provides two important tools to perform experiments with fault tolerance: a fault injection mechanism and a debug log.

The generation of the faults may be deterministic or probabilistic. In deterministic testing, the tester selects the fault patterns from the domain of possible faults. In probabilistic testing, the tester selects the fault patterns according to the probabilistic distribution of the fault patterns in the domain of possible faults.

The fault injection mechanism implemented in RADICMPI served for testing and debugging. The operation of the mechanism was deterministic, i.e., we programmed the mechanism to force all fault situations required to test the system functionality.

The mechanism is implemented at software level. This allowed a rigorous control of the fault injection and greatly facilitated the construction and operation of the fault injection mechanism. In practice, the fault injection code is part of the code of the RADICMPI elements.

The RADICMPI debug log mechanism served to help us in the development of the RADICMPI software and to validate some procedures. The mechanism records the internal activities in a log database stored in the local disk of each node.

Table 6-1 describes each field of the debug log database. The database has the same structure for protectors and observers

Table 6-1: Fields of the debug log

Column	Field name	Description
1	Element ID	Indicate the rank of the element. T# elements are protectors and O# elements are observers
2	Event id	Identifies the event type
3	Event time	Elapsed time in seconds since the program startup
4	Function name	Name of the internal function that generate the event
5	Event	Description of the event

6.2 Validation Experiments

Firstly, we perform a set of experiments in order to validate the spare nodes functioning. These experiments were performed based on a common ping-pong

algorithm execution and the results contained in debug log generated by RADICMPI. The Figure 6-1 depicts the flow of the validation process applied. After to define a test scenario and the expected results, we created specific events triggered by actions performed by the spare nodes usage. Finally, we analyze the debug log, allowing us to validate the correct system functioning.

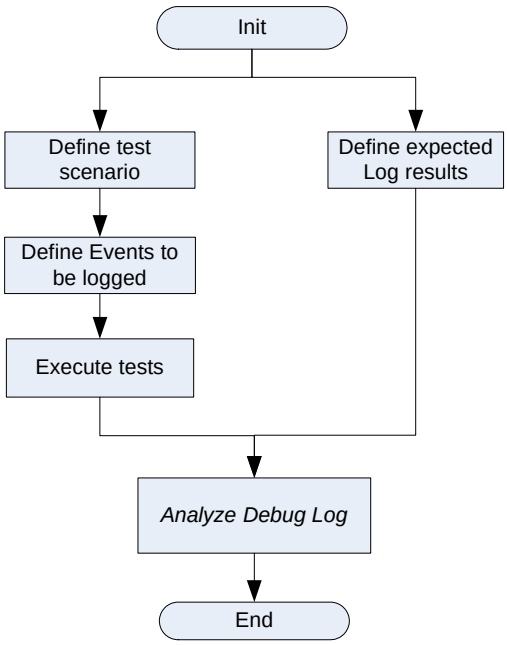

Figure 6-1: Flow representing the validation process

Validating the spare adding task

In this experiment, we aim to validate the procedure described in the item 4.2.1. Hence, we expect that a Protector was able to identify a new spare request and forward this request until receive it again. We put some event triggers when the Protector receives a request and when the Protector detects a repeated request and stops the forwarding. The Figure 6-2 shows the debug log excerpt containing these events.

```
.
.
.
2  1   management_thread  prt 3 : waiting message reception
8  1   management_thread  prt 3 : Request received. Type: Add new spare request
8  1   addNewSpare        prt 3 : waiting for spare data...
2  1   addNewSpare        prt 3 : Receiving a New spare. Initiating the message
forwarding:
8  1   addNewSpare        prt 3 : New spare added:  158.109.65.216
8  1   sendNewSpare       prt 3 :  new spare request sent to:  158.109.65.213
8  1   sendNewSpare       prt 3 :  new spare sent to:  158.109.65.213
2  1   management_thread  prt 3 : waiting connection
2  1   management_thread  prt 3 : waiting message reception
8  1   management_thread  prt 3 : Request received. Type: Stop session
2  1   management_thread  prt 3 : waiting connection
8  1   connect    prt 3 : ANTECESSOR: receiving information from : 158.109.65.215
8  1   connect    prt 3 : ANTECESSOR:  sending my information to 158.109.65.215
8  1   connect    prt 3 : successor OK
33  1  main       prt 3 : PRE_ANTE= 158.109.65.212 - ANTE= 2 - SUC= 4
45  2  obs_managing_thread        prt 3 : Command 24 from observer 3
10  2  manage_local_observers_list        prt 3 : + obs 3 from node -1. 1
observers attached
2  2   management_thread  prt 3 : waiting message reception
8  2   management_thread  prt 3 : Request received. Type: Add new spare request
8  2   addNewSpare        prt 3 : waiting for spare data...
2  2   addNewSpare        prt 3 : I already have the New spare. stopping the
message forwarding:
2  2   management_thread  prt 3 : waiting connection
.
.
.
```

Figure 6-2: Debug log excerpt of a Protector containing the new spare events

In this execution, we reserved one node to acts as a spare node. Such node starts a communication with the Protector number 3 of the configuration and sends its information. As shown in the last Figure, the Protector 3 receives the first new spare request, add its information and forward it to its neighbor. When the Protector receives the request for add the same spare, it stops the message forwarding. These

```
.
.
.
2   0  main    prt 5 : I am a spare
2   0  management_thread       prt 5 : Thread created
2   0  management_thread       prt 5 : waiting connection
2  70  management_thread       prt 5 : waiting message reception
8  70  management_thread       prt 5 : Request received. Type: Observers quantity
request
8  70  getObserverQty  prt 5 : Sending observer qty:  0
8  70  getObserverQty  prt 5 : Qty of observer sent:  0
2  70  management_thread       prt 5 : waiting connection
2  70  management_thread       prt 5 : waiting message reception
8  70  management_thread       prt 5 : Request received. Type: Receive spare list
request
8  70  rcvSpareList    prt 5 : waiting for spare data...
8  70  rcvSpareList    prt 5 : first spare data received:  158.109.65.216
2  70  management_thread       prt 5 : waiting connection
2  70  management_thread       prt 5 : waiting message reception
8  70  management_thread       prt 5 : Request received. Type: Use spare request
```

```
2  70  management_thread       prt 5 : calling use_spare function
8  70  use_spare       prt 5 : Changed: starting use_spare function.
communicating with:  158.109.65.213
2  70  use_spare       prt 5 : waiting command
8  70  use_spare       prt 5 : Request received. Type: Join the protector's chain
8  70  use_spare       prt 5 : request for heartbeat/watchdog received from:
158.109.65.213
2  70  watchdog_threadprt 5 : Thread created
2  70  connect prt 5 : ANTECESSOR: waiting receive information from my neighbour
2  70  heartbeat_thread        prt 5 : Thread created
8  70  heartbeat_thread        prt 5 : Connecting with:  158.109.65.213
8  71  connect prt 5 : SUCCESSOR: response received from an active protector with
its information:  158.109.65.213
8  71  connect prt 5 : antecessor OK
8  78  connect prt 5 : ANTECESSOR: receiving information from : 158.109.65.215
8  78  connect prt 5 : ANTECESSOR:  sending my information to 158.109.65.215
8  78  connect prt 5 : successor OK
45 78  obs_managing_thread     prt 5 : Command 12 from observer 4
54 78  manage_observers_list   prt 5 : Managing obs 4
10 78  manage_observers_list   prt 5 : + obs 4 from node 4. 1 observers attached
33 79  use_spare       prt 5 : PRE_ANTE= 158.109.65.212 - ANTE= 2 - SUC= 4
8  79  use_spare       prt 5 : Request received. Type: Checkpoint file transfer
request
2  79  use_spare       prt 5 : Chekpoint file transmission request
8  79  recvCkptLogFile prt 5 : starting receive file from:  158.109.65.213
2  79  recvCkptLogFile prt 5 : Checkpoint or Log file successfully received.
8  79  use_spare       prt 5 : Request received. Type: Message log file transfer
request
8  79  recvCkptLogFile prt 5 : starting receive file from:  158.109.65.213
2  79  recvCkptLogFile prt 5 : Checkpoint or Log file successfully received.
8  79  use_spare       prt 5 : Request received. Type: Recover observer request
2  79  use_spare       prt 5 : adding the observer in the observer list before
recover
2  79  recover_observers       prt 5 : Initiating recovery
39 79  recover_observers       prt 5 : Recovering process 3
8  79  use_spare       prt 5 : Request received. Type: Stop session
2  79  management_thread       prt 5 : returned from use_spare function
2  79  management_thread       prt 5 : waiting connection
4.
.
.
```

Figure 6-3: Debug log excerpt of a spare node Protector performing a recovery

events are highlighted in the excerpt shown before.

Validating the recovery task using spare

Beyond the correct finish of the application after been recovered using a spare node, we validate this task by using the debug log to certify that the correct steps were performed. In this experiment, we expect that after inject a fault, the Protector that detects this fault searches for a spare and starts the protocol for spare use. In the Figure 6-3 we can see the debug log of a spare node Protector, denoting the events related with the recovery process.

```
.
.
.
36 69  obs_managing_thread       prt 3 : Reception finished : 6400 B / 0.000754
```

```
s
12 69  watchdog_thread    prt 3 : ERROR : Success
2  69  manageFault        prt 3 : Fault detected
8  69  manageFault        prt 3 : Spare node Found! address:  158.109.65.218
8  69  querySpareObsQty   prt 3 :  Observer qty query sent to:  158.109.65.218
57 69  querySpareObsQty   obs 3 : Fault  observer qty=:  message 0
60 69  manageFault        prt 3 : obsQty= 0
8  69  manageFault        prt 3 : using Spare address:  158.109.65.218
8  69  sendRequest        prt 3 :  initiating request send command :  Receive
spare list request
8  69  sendRequest        prt 3 :  Socket closed. Opening with new IP Address...
158.109.65.218
8  69  sendRequest        prt 3 :  request sent command :  Receive spare list
request
8  69  sendSpareList      prt 3 : Start sending spare list to 158.109.65.218
8  69  sendSpareList      prt 3 : Finalizing spare list sending to 158.109.65.218
8  69  sendRequest        prt 3 :  initiating request send command :  Stop session
2  69  sendRequest        prt 3 :  Socket exists and has the same IP Address. Do
nothing...
8  69  sendRequest        prt 3 :  request sent command :  Stop session
8  69  sendRequest        prt 3 :  initiating request send command :  Use spare
request
8  69  sendRequest        prt 3 :  Socket closed. Opening with new IP Address...
158.109.65.218
8  69  sendRequest        prt 3 :  request sent command :  Use spare request
2  69  manageFault        prt 3 : fetching new neighbourhood with the spare:
8  69  sendRequest        prt 3 :  initiating request send command :  Join the
protector's chain
2  69  sendRequest        prt 3 :  Socket exists and has the same IP Address. Do
nothing...
8  69  sendRequest        prt 3 :  request sent command :  Join the protector's
chain
45 69  obs_managing_thread        prt 3 : Command 19 from observer 4
14 69  storage_message_log        prt 3 : Logging 6400 Bytes of message 327 from
source 3
.
.
```

Figure 6-4: Debug log excerpt of the Protector commanding the recovery in a spare node

The highlighted events are the requests that correspond with the process recovery detailed in item 4.2.2.

6.3 Evaluation Experiments

In order to evaluate the RADIC II behavior, we performed some experiments running well-known applications in different contexts. We tried to represent some distinct approaches to common parallel applications and measuring comparatively the effects of use or not the spare nodes approach.

For such class of experiments, we applied two kind of parallel programs: a master-worker matrix product and an N-Body particle simulation using non-blocking functions in a pipeline approach.

We choose the matrix product algorithms because we could apply different parallel paradigms over it. We used a master-worker and a SPMD algorithm, facilitating the creation of different fault scenarios. As shown in Figure 4-3, the MW algorithm has a 1-to-N message pattern (Figure 4-3a). The master process communicates with all the worker processes. Each worker process only communicates with the master process. The SPMD algorithm has a communication mesh (Figure 4-3b). Each application process communicates with their neighbors, representing a tightly coupled application.

The MW algorithm also offered an additional control over the application behavior; it was possible to use two strategies to balance the computation load between the workers: dynamic and static.

In the static strategy, the master first calculates the amount of data that each worker must receive. Next, the master sends the data slice for each worker and waits until all workers return the results. In this strategy, the number of messages is small but each message is large, because the master only communicates at the beginning, to send the matrices blocks to the workers; and at the end, to receive the answers.

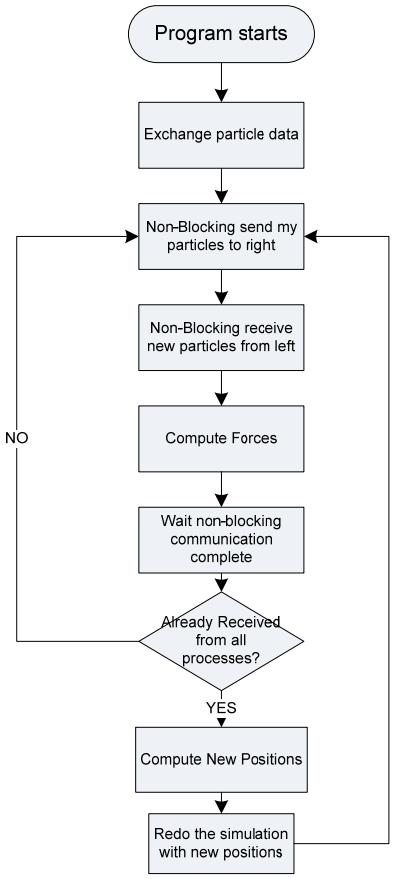

Figure 6-5: The N-Body particle simulation flow

In the dynamic strategy, the master slices the matrices in small blocks and sends pairs of blocks to the workers. When a worker answered the block multiplication's results, the master consolidates the result in the final matrix and sends a new pair of blocks to the worker. In this strategy is easy to control the computation-to-communication ratio by changing the block size. Small blocks produce more communication and less computation. Conversely, large blocks produce less communication and more computation.

The N-Body program bases in the example presented by Gropp [Gropp *et al, 1999, p. 117*]. This program performs a particle simulation, calculating the attraction forces between them. It is implemented under the pipeline parallel paradigm and uses

non-blocking MPI communication functions to increase the performance. The Figure 6-5 represents the flow of the actions performed by each process.

6.3.1 Evaluating According with the Fault Moment

In this experiment series, we evaluate the behavior of the applications according with the moment of the fault when using or not dynamic redundancy.

In order to perform these experiments, we executed two approaches for the matrix product algorithm, the master-worker static distributed and the SPMD based on the cannon algorithm. Thus, we can evaluate the coupling factor too, once the SPMD algorithms are commonly tightly coupled.

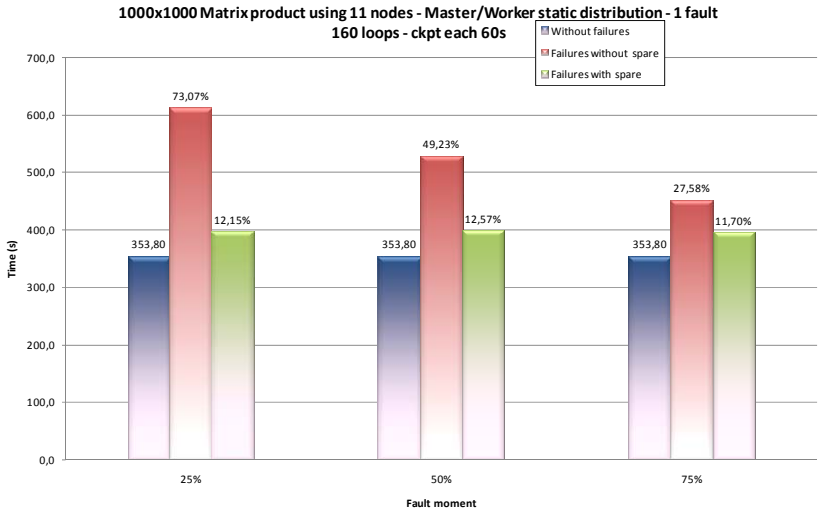

Figure 6-6: Results of matrix product using a master-work static distributed program

Intending to obtain more diversity, we performed this experiment executing a product of two 1000 X 1000 matrixes of float values in the master-work approach over a cluster with eleven nodes in the first case. In order to increase the computing time, we repeat the product 160 times in all executions. In the second case, we executed the cannon algorithm with 1500 X 1500 matrixes over a nine nodes cluster. In both cases, we inject a fault at approximately 25%, 50% and 75% of the total execution time and we compared with a failure-free execution and with the spare

nodes usage. In this case, we repeated the computing 160 times in order to enlarge the execution time.

The Figure 6-6 contains a chart showing the results with the master-worker approach. In this chart, we can see that the overhead caused by a recovery without spare (the red middle column in each fault moment) versus using spare (the green right column in each fault moment) with one fault occurring in different moments. The overhead not using spares shows itself inversely proportional to the moment when the fault occurs, generating greater overheads (reaching 73.07% in the worst case analyzed) in premature fault case, while using spare, the overhead keeps constantly and low despite the moment of the fault.

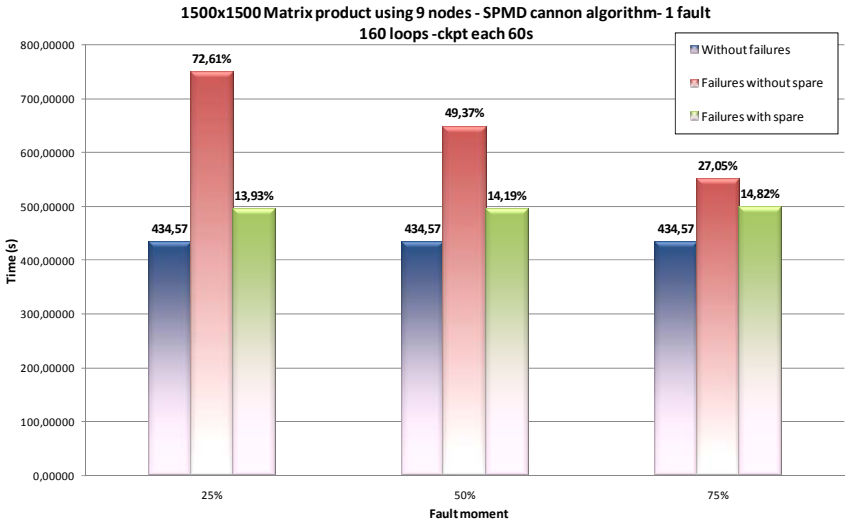

Figure 6-7: Results of matrix product using a SPMD program based in the cannon algorithm

The Figure 6-7 shows de result chart with the SPMD program. We see an analogous behavior with the overhead caused by not using spare nodes. The overhead caused by the spare nodes usage is slightly greater than the static distribution approach. This increment is due to the high coupling level in the SPMD approach, the time spent in the recovery affects directly the communications with the neighbors' processes and this delay continues propagating by the others process of the

application, while the recovery in the master-worker approach only affects the failed worker.

6.3.2 Evaluating According with the Number of Nodes

In these experiments, we evaluated the behavior of the fault recovery in different cluster sizes. Due to our physical limitations, we could not to prove in large size clusters, which it allows to certify the RADIC II scalability. The current experiments only give us an idea that RADIC II does not affect the scalability of a program.

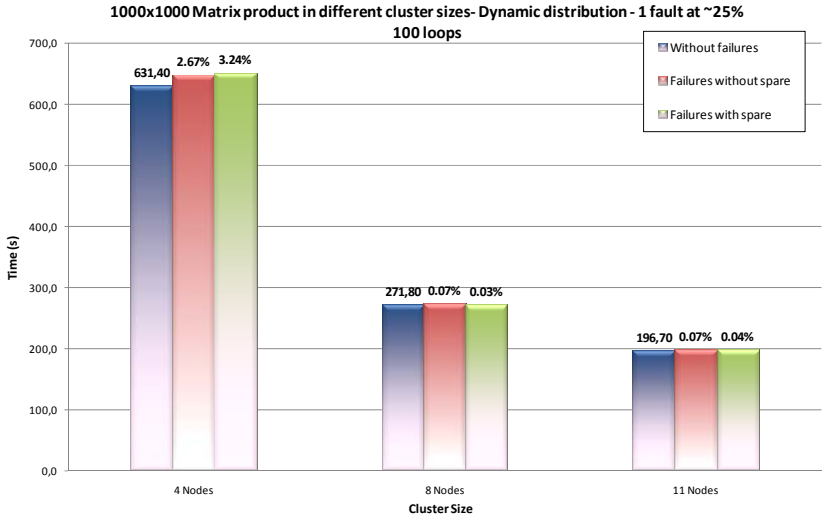

Figure 6-8: Results of matrix product using a master-worker program with load dynamically balanced running in different cluster sizes

We ran two approaches for a master-work matrix product: using a static distribution and using a dynamic distribution of matrix blocks. In both cases we performed a product between two 1000 X 1000 matrixes. We executed the program with four, eight and eleven nodes. We injected faults always at 25% of the execution time, approximately. We measured the execution time when using or not the spare nodes and comparing with a fault free execution time.

The Figure 6-8 shows a chart with the results of the execution with a dynamic load balancing approach. We can see clearly that the load balancing can to mitigate the side-effects of the RADIC regular recovery, and the spare nodes use is almost equal than not using it, being the worse approach in the smaller cluster. Indeed, the processes in the overloaded node start to perform fewer tasks than other nodes, and their workload is distributed among the cluster, almost not affecting the execution time. As the cost of recovery using spare nodes may slightly greater, may be better do not use this feature in some cases.

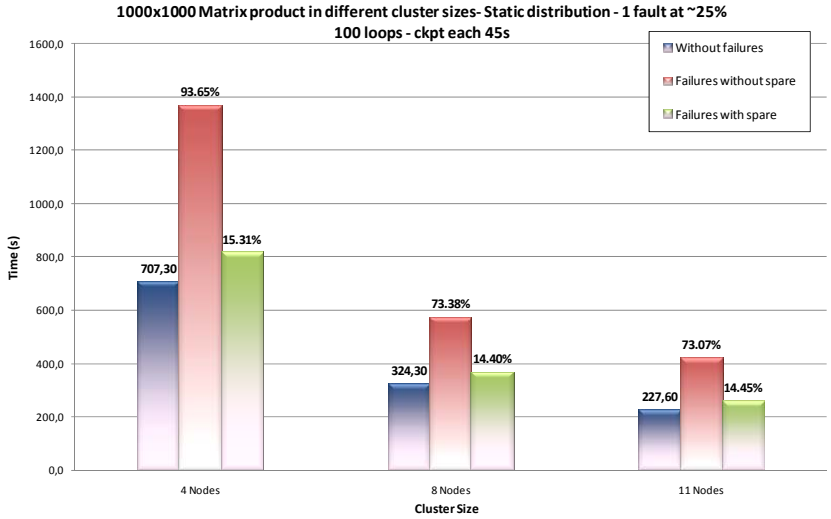

Figure 6-9: Results of matrix product using a master-worker program with static load distribution running in different cluster sizes

The chart in the Figure 6-9 shows the benefits of using spare nodes in some parallel approaches. In this case, using a static load distribution, the node that hosts the recovered process suffers a strong degradation, high affecting the overall execution time independently of the size of the cluster. By other side, using the spare nodes approaches, the overall impact in the execution time is low and stable, also independently of the number of nodes.

6.3.3 Evaluating the Throughput in Continuous Applications

As many of the actual parallel applications are intended to run continuously in a 24x7 scheme, we performed an experiment intending represent the behavior of these applications. In this experiment, we executed continuously the N-Body particle simulation in a ten nodes pipeline and injected three faults in different moments and different machines, measuring the throughput of the program in simulation steps per minute. We analyzed four situations: a) a failure-free execution, used as comparing; b) three faults recovered without spare in the same node; c) three faults recovered without spare in different nodes and d) three faults recovered with spare.

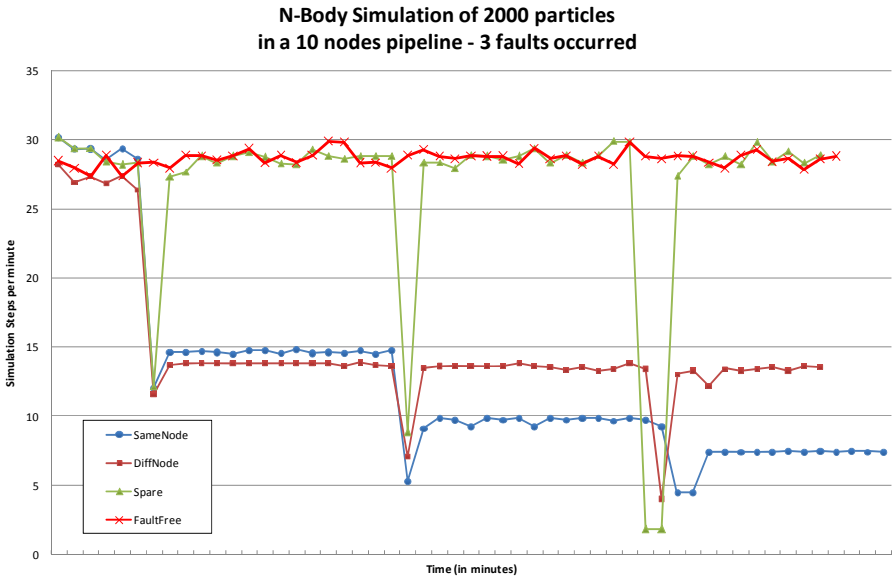

Figure 6-10: Results of an N-Body program running continuously after three faults in different situations.

In the Figure 6-10 we can see the result chart of this experiment. In this experiment, we can perceive the influence of the application kind over the post-recovery execution. When the three faults are recovered in different nodes, the application's throughput suffers an initial degradation, but in the subsequent faults, just changes a little. This behavior occurs because the pipeline arrangement: the degradation of the node containing the second recovered is masked by the delay

caused by the first recovered process node. This assumption is confirmed when all faults processes are recovered in the same node, we can perceive a degradation of the throughput after each failure. When executing with spare nodes presence we see that after quick throughput degradation, the system backs to the original simulation step rate. We see also that the penalization imposed by the recovery process using spare is greater than the regular RADIC process, but this loss is quickly compensated by the throughput restoring in the remaining execution.

Chapter 7

Conclusions

In this dissertation we saw that the demands for high performance computing, generally represented by parallel systems seems to continue growing. Several applications have been ported to parallel environments, expecting obtain a certain gain of performance, commonly represented by their execution times. Furthermore, besides of performance, the users of these applications usually aim a certain level of availability.

We explained that due to the reliability constraints, many large parallel systems may not offer the expected level of availability. Therefore, the fault tolerance has been a constantly growing issue in order to improve the availability of these systems, resulting in a considerable number of researches in this area.

The RADIC (Redundant Array of Independent Fault Tolerance Controllers) architecture was studied as a fault tolerance solution for messaging passing providing transparency, decentralization, scalability and flexibility. We have presented the concepts of this architecture, which it bases on two kind of processes working together to provide fault tolerance, the Observers and the Protectors. We saw the *Modus Operandi* of these processes and their issues besides a practical implementation of the architecture.

We argued that in order to tolerate a fault, some message passing based solutions may generate a system configuration change at the recovery, i.e., respawning failed process in other active node of the application, which changes the original process distribution. Moreover, we shown that this change of configuration may lead to some

system degradation, generally meaning overall performance loss as demonstrated by some experiments ran over the RADIC architecture.

This project was undertaken to design a fault tolerant architecture avoiding the side effects of the recovery process and evaluate it effectiveness. We used RADIC as basis to the development of RADIC II, an architecture that adds other protection level to RADIC, inserting a flexible dynamic redundancy feature.

The implementation of this new feature, represented by the use of spare nodes, did not affect the RADIC characteristics of transparency, decentralization, flexibility and scalability. Hence, we developed a transparent management of spare nodes, which is able to request and use them without need any centralized information.

In RADIC II, we implemented the ability of dynamically insert new spare nodes during the application execution. The architecture now also allows the replacement of faulty nodes. Moreover, the new features implemented may be applied to perform maintenance tasks, injecting faults in specific nodes forcing the processes running on them to migrate to a recently inserted spare node. These abilities represent the flexibility of RADIC II, beyond to keep the original RADIC structural flexibility.

We modified the original RADICMPI prototype implementing the flexible redundancy feature. With the new version of this prototype, we performed several experiments with RADIC II in order to validate its functionality and to evaluate its appliance in different scenarios. All of scenarios bases on a twelve nodes cluster running the LINUX operating system over a TCP/IP network.

We validated the RADIC II operation by using a debug log provided by the RADICMPI original implementation. We inserted new events related with the spare nodes usage. Initially we executed a ping-pong algorithm due to its simplicity, then after injected a fault in some node and the application has been correctly recovered, we checked the debug log in order to certify that all steps were correctly performed. We applied the same approach in order to validate the non-blocking function.

We evaluated the appliance of our solution under two concepts: the overall execution time, and the throughput of an application. We compared the effects of

faults having or not available spare nodes. In the execution time case, we applied different approaches for a matrix product algorithm, using a static distributed Master/Worker and a SPMD approach implementing a Cannon algorithm. In order to evaluate the throughput, we executed an N-Body particle simulation using a pipeline paradigm, measuring the simulation steps performed comparing, again, the appliance or not of spare nodes. Hence, the following conclusions can be drawn from the present study.

Our experiments has shown that the side-effects caused by some recovery approaches is dependant of the factors like application characteristics, i.e. message pattern, parallel paradigm applied, i.e. pipeline, Master-Work, or SPMD and where the process is recovered. We saw that the fault recovery, generally affects the overall performance of the system, and the generated degradation may vary according to where the process recovers and the parallel paradigm applied. Other relation perceived is about the application coupling. Applications with high coupling level between the computing nodes tend to suffer more intensively with the system configuration change caused by the recovered process.

Moreover, we conclude that the use of a flexible redundancy scheme is a good approach to avoid the effects of the system configuration changes. Our solution has shown to be effective even in faults near at the application finishing. RADIC II also shows a small overhead caused by the recovery process. The experimental results have shown execution times and throughput values very near to a failure-free execution. This work was presented at CACIC'2006 congress [Santos*, et al.*, 2006]. After, we obtained new results that are going to appear in the ParCo2007 congress [Santos*, et al.*, 2007].

These findings enhance our understanding about the fault tolerance issues, like the relationship with application characteristics and behavior, or the influence of a parallel paradigm in recovered applications.

The project was limited in some ways. First, the project used a small sized cluster, which does not reflects the reality and makes hard to test some aspects like the scalability, hence, caution must be applied, as the findings might not be

transferable to large scale clusters. Despite of our efforts in to increase the number of MPI functions implemented in RADICMPI, the actual set has restricted the possibility of probe our solution in different benchmark application. Finally, we found an intrinsic limitation of our solution when adding spare nodes during the execution. Such limitation occurs when all nodes of the cluster already was been replaced, so the new spare inserted does not have to know any machine in the cluster, needing some additional information.

7.1 Open Lines

After a lot of work, we look to the present and we see many open lines that were found during the path to reach here. These open lines represent the future work that may be performed, expanding the RADIC II horizon.

The ideal number of spare nodes and its ideal allocation through the cluster still are undiscovered subject. Further research might be investigate how it is possible to achieve better results allocating spare nodes according with some requirements like degradation level acceptable, or memory limits of a node.

New technologies are arriving each day. A permanent task will be to study how to adapt and use RADIC II with the new trends of the high computing area, i.e. how behaves RADIC II using *multicore* computers?, how can we exploit the characteristics of this architecture?

Fault tolerant systems generally are very complex systems. RADIC II is not an exception. Considerably more work will need to be done to generate a RADIC II analytical model, but will be very useful helping to understand better the architecture and providing tools to improve the RADIC II functioning allowing to determine better values for some parameters like checkpoint interval or protection mapping. Furthermore, this model may be applied in order to foresee the execution time under some parameters.

A study about the possible election policies to be used in the node replacement feature will be useful to determine which the ideal behavior to be taken in these situations, considering factors like load balance or time to recover.

The maintenance feature of RADIC II is still not a widely explored subject. Additional research might be address the integration of this feature with some fault prediction scheme, which will allow RADIC II to perform transparent process migration avoiding faults before their happening.

It would be interesting to assess the effects of RADIC II in large clusters and with different kind of applications. This study will give us a real knowledge about the RADIC II scalability. Due to physical difficulties to access these machines, a RADIC II simulator would be necessary beside to complete the MPI implementation in RADICMPI.

The autonomic computing systems [Kephart and Chess, 2003] have been a new trend in computing systems. Basing on the human autonomic system, this new trend establishes a new group of systems having the abilities of self-healing, self-configuring, self-protect and self-optimize. We see that RADIC already provides the self-healing ability, while RADIC II implements the self-configuring capacity. Hence, a new research might perform the steps toward an autonomic fault tolerant system implementing the self-protecting and self-optimizing features.

References

[Agbaria and Friedman, 1999] - Agbaria, A. M. and Friedman, R. *Starfish: fault-tolerant dynamic MPI programs on clusters of workstations*. In Proceedings of The 8th International Symposium on High Performance Distributed Computing, pp. 167-176. Redondo Beach, USA. 3-6 August, 1999.

[Alvisi and Marzullo, 1998] -Alvisi, L. and Marzullo, K.. *Message Logging: Pessimistic, Optimistic, Causal, and Optimal*. IEEE Transactions on Software Engineering. 24(2), pp 149-159 Feb. 1998.

[Alvisi, *et al.*, 1999] - Alvisi, L., Elnozahy, E., Rao, S., Husain, S. A. and de Mel, A. *An analysis of communication induced checkpointing*. In Proceedings of The 29th Annual International Symposium on Fault-Tolerant Computing, pp. 242-249. Winsconsin, USA. June 15-18, 1999.

[Bosilca *et al, 2002*] - Bosilca G., Bouteiller A., Cappello F., Djilali S., Fedak G., Germain C., Herault T., Lemarinier P., Lodygensky O., Magniette F., Neri V. and Selikhov A., *MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes*, in Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, 2002, pp. 1-18.

[Bouteiller, *et al.*, 2003a] - Bouteiller, A., Cappello, F., Hérault, T., Krawezik, G.,Lemarinier, P., and Magniette, F.. *MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging*. High Performance Networking and Computing (SC2003), Phoenix USA, IEEE/ACM..

[Bouteiller, *et al.*, 2003b] - Bouteiller, A., Lemarinier, P., Krawezik, K. and Capello, F. *Coordinated checkpoint versus message log for fault tolerant MPI*. In Proceedings of the 2003 IEEE International Conference on Cluster Computing, pp. 242-250. Hong Kong, China. December 1-4, 2003. IEEE Computer Society.

[Bouteiller, *et al.*, 2006] - Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P. and Cappello, F. *MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI*. International Journal of High Performance Computing Applications, 20(319-333 2006)

[Burns *et al, 1994*] - Burns, G., Daoud, R., Vaigl, J., *LAM: An Open Cluster Environment for MPI*. In Ross, J.W., ed.: Proceedings of Supercomputing Symposium ’94, University of Toronto (1994) 379–386.

[Duarte, 2007] -Duarte, A. *RADIC: A Powerful Fault Tolerant Architecture.* PHD Thesis, Computer Architecture and Operating Systems Department. Universitat Autònoma de Barcelona, Spain.

[Duarte*, et al.*, 2006] - Duarte, A., Rexachs, D. and Luque, E. *An Intelligent Management of Fault Tolerance in cluster using RADICMPI.* Lecture Notes on Computer Science - Proceedings of The 13th European PVM/MPI User's Group Meeting, 4192:150-157. Springer Berlin / Heidelberg, 2006b.

[Duarte*, et al.*, 2007] - Duarte, A., Rexachs, D. and Luque, E. *Functional Tests of the RADIC Fault Tolerance Architecture*. In Proceedings of The 15th Euromicro Conference on Parallel, Distributed and Network-based Processing, pp. 278-287. Napoli, Italy. February 7-9, 2007. IEEE Computer Society.

[Chandy and Lamport, 1985] - Chandy, K. M. and Lamport, L. *Distributed snapshots: determining global states of distributed systems*. ACM Transactions on Computer Systems, 3(1):63-75 (February 1985)

[Elnozahy*, et al.*, 2002] - Elnozahy, E. N., Alvisi, L., Wang, Y.-M. and Johnson, D. B. *A survey of rollback-recovery protocols in message-passing systems*. ACM Computing Surveys, 34(3):375-408 (September 2002)

[Gao *et al,* 2005] - Gao, W., Chen M. and Nanya, T., *A Faster Checkpointing and Recovery Algorithm with a Hierarchical Storage Approach*, Hpcasia, vol. 0, pp. 398-402, 2005.

[Gropp *et al, 1999*] - Gropp, W., Lusk, E., and Skjellum, A.. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Second edn. Anon. Cambridge, MA: MIT Press. ISSN 0-262-57104-8.

[Hariri*, et al.*, 1992] - Hariri, S., Choudhari, A and Sarikaya, B. *Architectural Support for Designing Fault-Tolerant Open Distributed Systems*. Computer, 25(6):50-62 (June 1992)

[Hargrove and Duell, 2006] - Hargrove, P. H. and Duell, J. C. *Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters*. In Proceedings of Scientific Discovery through Advanced Computing (SciDAC 2006), pp. 494-499. Denver, USA. June 25-29, 2006. U.S. Department of Energy.

[Kalaiselvi and Rajaraman, 2000] - Kalaiselvi, S. and Rajaraman, V. *A survey of checkpointing algorithms for parallel and distributted computers*. Sãdhanã, 25(5):498-510 (October 2000)

[Kephart and Chess, 2003] - Kephart, J.O. and Chess, D.M., *The Vision of Autonomic Computing*. Computer. 36 (1):41-52 , New York, 2003

[Koren and Krishna, 2007] - Koren, I. and Krishna, C. M., *Fault Tolerant Systems*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007

[Kondo, M., *et al*. 2003] - Kondo, M., Hayashida, T., Imai, M., Nakamura, H., Nanya, T.; Hori, A.. *Evaluation of Checkpointing Mechanism on SCore Cluster System*. IEICE Transactions on Information and Systems. 86(12), pp 2553-2562

[Jalote, 1994] - Jalote, P. *Fault Tolerance in Distributed Systems*. 1st. ed. Englewood Cliffs, USA: Prentice Hall, 1994.

[Li and Lan, 2006] - Li, Y., Lan, Z., *Exploit failure prediction for adaptive fault-tolerance in cluster computing*. In: CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), Washington, DC, USA, IEEE Computer Society (2006) 531–538

[Meyer, 1980] - Meyer, J.F. *On Evaluating the Performability of Degradable Computing Systems*. IEEE Transactions on Computers, 29(8) 720-731(Aug 1980).

[MPI Forum, 2006] - Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Available at http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html. Accessed in may, 2006. Knoxville, USA. 1995

[Nagajara *et al*. 2005] - Nagaraja, K., Gama, G., Martin, R. P., Jr., W. M., Nguyen, T. D., *Quantifying Performability in Cluster-Based Services*. IEEE Transactions on Parallel and Distributed Systems 16, 5 (May 2005).

[Piedad And Hawkins, 2001] - Piedad, Floyd and Hawkins, Michael. *High Availability. Design, Techniques, and Processes* . 1st. ed. Upper Saddle River, USA: Prentice Hall, 2001

[Rao *et al*, 2000] - Rao, A. , Alvisi, L. and Vin, H., *The Cost of Recovery in Message Logging Protocols*. IEEE Transactions on Knowledge and Data Engineering, Vol. 12, No. 2, pp. 160-173, April 2000.

[Santos, *et al.*, 2006] - Santos, G. A., Duarte, A., Rexachs, D. and Luque, E. *Recuperando prestaciones en clusters tras ocurrencia de fallos utilizando RADIC*. In Proceedings of XII Congreso Argentino de Ciencias de la Computación (CACIC 2006), pp. Potrero de los Funes, Argetina. October 17-21, 2006.

[Santos, *et al.*, 2007] - Santos, G. A., Duarte, A., Rexachs, D. and Luque, E. *Mitigating the post-recovery overhead in fault tolerant systems*. In Proceedings of Parallel Computing 2007 (ParCo2007), pp. (to appear). Aachen, Germany. September 3-7, 2007.

[Shooman, 2002] - Shooman, Martin L. *Reliability of Computer Systems and Networks.Fault Tolerance: Analysis, and Design*. 1st. ed. New York, USA: John Wiley & Sons, Inc., 2002

[Squyres and Lumsdaine, 2003] Squyres, J. M. and Lumsdaine, A., *A component architecture for LAM/MPI*, in Proceedings, 10th European PVM/MPI, Lecture Notes in Computer Science, Vol. 2840, pp. 379-387, 2003

[Stellner, 1996] - Stellner, G. *CoCheck: Checkpointing and Process Migration for MPI*. In Proceedings of The 10th International Parallel Processing Symposium (IPPS'96), pp. Honolulu, Hawaii. April 15-19, 1996.

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

[Zambonelli, 1998] - Zambonelli, F. *On the effectiveness of distributed checkpoint algorithms for domino-free recovery*. In Proceedings of The 17th International Symposium on High Performance Distributed Computing, pp. 124-131. chicago, USA. July 28-31, 1998.