Universitat Autònoma de Barcelona

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

RADIC: A POWERFUL FAULT-TOLERANT ARCHITECTURE

Thesis submitted by Angelo Amancio Duarte in fulfillment of the requirements for the degree of Doctor per the Universitat Autònoma de Barcelona.

Bellaterra, May 2007

RADIC: A POWERFUL FAULT-TOLERANT ARCHITECTURE

Thesis submitted by Angelo Amancio Duarte in fulfillment of the requirements for the degree of Doctor per la Universitat Autònoma de Barcelona. This work has been developed in the Computer Architecture and Operating Systems department of the Universitat Autònoma de Barcelona and was advised by Dr. Dolores Isabel Rexachs del Rosario.

Bellaterra, May 2007

Thesis Advisor

Dr. Dolores Isabel Rexachs del Rosario

To Deborah, Amanda and Sarah, the goddesses who offered me all energy to complete this work.

Acknowledgments

It is hard to believe that almost four years have gone since the beginning of my Ph.D. course. Now, at the end of this journey, I remember how many people I have known and how important were these people to me. Although I remember all them with kindness, some of them made the difference. Besides helping me, these persons have improved me. The following lines are a tribute for them.

To my wife Deborah and my daughters Amanda and Sarah, today the most important people in my life, I hope I can compensate the suffering and effort that they have done in the last four years. I thank them to be my emotional pillar and I thank the love they unconditionally give me.

I had luck in having two great tutors in my job: Emilio Luque and Dolores “Lola” Rexachs. The lessons they gave me really made the difference when things did not go as well as I wished. Both always cared about how I was living in Spain and they gave me full support in my decisions. The word “tutor” has gained more significance since I started to work with them.

I am very grateful to my two great Brazilian friends: Eduardo Argollo and Genaro Costa for teaching me most of the technical knowledge that helped me to develop my job.

To my Brazilian friends Ines Martinez, Guna Santos and Christiane Dalforno, I thank the funny times that made the life easier when I missed my family or when the work was harder.

To my Catalan friends Carlos Moreno, David Ruiz and Ana Esteves, I thank the lessons about the Catalunya. Because of them, I could better comprehend the Catalan people.

I thank to my Argentine friends Mauricio Hanzich, Diego Mostaccio and Paula Fritzsche, for teaching me about Argentina and for proving that real friendship between Brazilian and Argentine is possible.

Thanks to my Chinese friend Xiao Yang, from whom I have learned a lot about the Chinese culture (and a few Chinese words, too). Because of him, I matured many philosophical concepts at the “Friday's Philosophical Meetings”, mediated by our common friend Mauricio. It was also because of him that I ate the best Chinese food so far.

I express my gratitude to all professors of the Ph.D. program at CAOS Department, for teaching me new matters that have helped me in my job. I also thank to the technicians Daniel Ruiz and Jordi Valls, for maintaining the machines working.

Finally, I thank to Universidade Católica do Salvador (UCSal) by the financial support during these years.

Bellaterra, June 2007

Angelo Amancio Duarte

Resumen

La tolerancia a fallos se ha convertido en un requerimiento importante para los ingenieros informáticos y los desarrolladores de software, debido a que la ocurrencia de fallos aumenta el coste de explotación de un computador paralelo. Por otro lado, las actividades realizadas por el mecanismo de tolerancia de fallo reducen las prestaciones del sistema desde el punto de vista del usuario.

Esta tesis presenta una arquitectura tolerante a fallos para computadores paralelos, denominada RADIC (Redundant Array of Distributed Fault Tolerance Controllers,), que es simultáneamente transparente, descentralizada, flexible y escalable.

RADIC es una arquitectura tolerante a fallos que se basa un controlador distribuido para manejar los fallos. Dicho controlador se basa en procesos dedicados, que comparten los recursos del usuario en el computador paralelo.

Para validar el funcionamiento de la arquitectura RADIC, se realizó una implementación que sigue el estándar MPI-1 y que contiene los elementos de la arquitectura. Dicha implementación, denominada RADICMPI, permite verificar la funcionalidad de RADIC en situaciones sin fallo o bajo condiciones de fallo. Las pruebas se han realizado utilizando un inyector de fallos, involucrado en el código de RADICMPI, de manera que permite todas las condiciones necesarias para validar la operación del controlador distribuido de RADIC.

También se utilizó la misma implementación para estudiar las consecuencias de usar RADIC en un ambiente real. Esto permitió evaluar la operación de la arquitectura en situaciones prácticas, y estudiar la influencia de los parámetros de RADIC sobre el funcionamiento del sistema.

Los resultados probaron que la arquitectura de RADIC funciona correctamente y que es flexible, escalable, transparente y descentralizada. Además, RADIC estableció una arquitectura de tolerancia a fallos para sistemas basados en paso de mensajes.

Abstract

Fault tolerance has become a major issue for computer engineers and software developers because the occurrence of faults increases the cost of using a parallel computer. On the other hand, the activities performed by the fault tolerance mechanism reduce the performance of the system from the user point of view.

This thesis presents RADIC (Redundant Array of Distributed Independent Fault Tolerance Controllers,) a fault-tolerant architecture to parallel computers, which is simultaneously transparent, decentralized, flexible and scalable.

RADIC is a fault-tolerant architecture that implements a fully distributed controller to manage faults. Such controller rests on dedicated processes, which share the user's resources in the parallel computer.

In order to validate the operation of RADIC, we created RADICMPI, a message-passing implementation that includes the elements of the RADIC architecture and complies with the MPI-1 standard.

RADICMPI served for to verifying the functionality of RADIC in scenarios with and without failures in the parallel computer. For the tests, we implemented a fault injector in RADICMPI in order to create the scenarios required to validate the operation of the RADIC distributed controller.

We also used RADICMPI to study the practical aspects of using RADIC in a real environment. This allowed us to evaluate the operation of our architecture in practical situations, and to study the influence of the RADIC parameters over the system performance.

The results proved that the RADIC architecture operated correctly and that it is flexible, scalable, transparent and decentralized. Furthermore, RADIC established a powerful fault-tolerant architecture model for message-passing systems.

Table of Contents

List of Figures

Chapter 1

Introduction

Any man made machine may fail and the computers are not an exception. Since the beginning of the computational science until our days, the computer designers and computer engineers had to deal with faults.

As a rule of thumb, all computer engineers know that when a computer system becomes more complex, the system's susceptibility to faults increases. Such rule is fully applicable to the parallel computers.

The number of nodes in the parallel computers has steadily increased since the debut of the parallel machines in the computer's world and, in spite of the quality improvement in each single component of the nodes, one may argue that, as certain as the sun will rise tomorrow, any current large computers will suffer some kind of failure.

A fast look in the current "Top500 Supercomputer Sites" list (November/2006) shows that the Top100 supercomputers have more than 1000 nodes and the top computer has more than 130000 nodes [Top500.Org, 2006]. Currently, there is no signal that the current trend of increasing the number of nodes in parallel computers will stop. Such trend forces the computers engineers and software programmers to increase the amount of effort dedicated to deal with faults.

Dealing with fault tolerance has become a major task for computer engineers and software developers because the occurrence of faults increases the cost of using a parallel computer. However, the inclusion of fault tolerance in a system increases the complexity of such system from the user point of view. Furthermore, the operation of the fault tolerance mechanism interferes in the operation of the distributed parallel

application and such interference often appears as a performance loss for system's users.

In this thesis, we proposed a fault tolerant architecture to parallel computers. In the development of such architecture, we assumed that, in order to attend the user expectations, a fault-tolerant architecture for modern parallel computers must simultaneously be transparent, decentralized, flexible and scalable. Currently no fault tolerance solution satisfies all these requirements simultaneously; because of this, we have developed the RADIC architecture.

RADIC is the acronym for "Redundant Array of Distributed Independent Fault Tolerance Controllers". RADIC is a fully distributed fault tolerance architecture that implements a distributed controller to manage faults. The RADIC controller rests on dedicated processes, which run together with the parallel application processes, i.e., they share the resources allocated for the user in the parallel computer. Therefore, the RADIC controller works as a distributed parallel application.

Since RADIC operates in a fully distributed way, without hierarchy or any global coordination between their processes, it is expandable and it mitigates the interference of the fault tolerance mechanism over the application's scalability.

RADIC is transparent for the system administrators, because its operation does not require any human intervention in order to maintain the parallel application execution in case of a failure in a node of the parallel computer. Furthermore, RADIC also does not require any change in the parallel application's code, what make it completely transparent from the programmer point of view.

RADIC is flexible because it allows different structures and relationships between its operational elements, facilitating the adaptation of the fault tolerance mechanism to the structure of the parallel computer. The flexibility is also present in adjusting the RADIC parameters, which may be set individually for each process that is been protected, or combined for a group of processes.

The RADIC operation bases on the classical pessimistic message-log rollback-recovery protocol. We choose this protocol because it allows that the fault

tolerance mechanism of a process operates without coordination with the fault tolerance mechanism of the other processes. This independence between the processes is the key feature of the distributed controller.

The distributed controller for fault tolerance implemented by RADIC executes the major activities of a fault-tolerant architecture based on such protocol. The RADIC activities are:

- it stores state information and deterministic events of the parallel application while the parallel computer is operating well;
- it monitors the parallel computer nodes in order to detect faults;
- it recovers faulty processes in a survivor node;
- it masks faulty nodes so the processes of the parallel application are not affected by the recovering of a faulty process in a different node from the one it was originally running.

In order to test the RADIC architecture in message-passing parallel computers, we implemented a prototype based on the MPI standard [Argonne National Laboratory, 2007a] and used it to execute a set of applications under different fault conditions. The RADICMPI implementation served also as a test platform to assess the interference of the fault tolerance mechanism on the parallel computer operation and to test the robustness of the RADIC concepts in a practical environment.

1.1 The RADIC key features

A fault-tolerant architecture driven by the modern parallel computers characteristics must simultaneously be transparent, decentralized, flexible and scalable. Now, we justify that such features are important because they attend to an important group of user's expectations.

The first user expectation is, of course, that his/her **parallel application correctly finishes** independent of faults in the parallel computer. This is obviously the main obligation of any fault-tolerant architecture for parallel computers and demands no further discussion.

The next requirement is that the fault-tolerant architecture must be easy to use. Easiness of using indicates that the fault-tolerant architecture must be as **transparent** as possible, i.e., that the user neither need to change his/her application code in order to use the fault-tolerant architecture nor need to concern about how the fault tolerance mechanism operates.

Any user, or system administrator, wants to pay the lowest price for the best fault-tolerant architecture and such price relates strictly to the efficiency of the fault-tolerant architecture. One important feature that determines the efficiency of a fault-tolerant architecture is the allocation of resources dedicated to implement the fault-tolerance. Architectures that use **decentralized** resources are more efficient because they avoid the bottleneck caused by centralized resources and allow a better distribution of the fault tolerance activities among the elements of the parallel computer.

Another interesting feature of a fault-tolerant architecture should be the possibility of sharing the same resources in the nodes that the user uses to compute his/her application, instead of requiring some dedicated or centralized nodes to operate. This feature would facilitate the allocation of the fault tolerance resources and would simplify the construction of the fault tolerance mechanism in the system. Furthermore, by avoiding the using of elements dedicated only to fault tolerance, the architecture will use the computers resources more efficiently.

The user also wants that the fault tolerance mechanism **scales** as well as, or better than, his/her parallel application and that the fault-tolerant architecture does not compromise his/her application's scalability.

Finally, from the point of view of the system administrator, **flexible** fault-tolerant architecture must allow different combinations of their operational elements, in order to better adapt its operation to the parallel computer structure. Furthermore, the adjusting of the fault tolerance parameters is also an important requirement for systems administrators, since they may tune the fault-tolerant architecture to the system requirements of the parallel computer.

In order to support such affirmations, let us suppose that a user needs to execute his/her program in a cluster. Obviously, the user expects that once s/he has started his/her program execution, such program will correctly finish before time bound.

Now let us suppose that a node fails before the end of the program execution. Assuming that the user would detect the failure instantly (let us figure out that s/he is in front of her/his terminal until the end of the program), we may assume two scenarios:

- Scenario 1: Once the user detects the problem, s/he simply either waits that someone fixes the faulty node and restart the program or re-launches the program with a node less.
- Scenario 2: The user loses all his/her work because even if s/he restarts the program immediately after s/he detects the fault the program will not finish in the expected time.

In spite of the unpredictable reaction that our "friendly" user might have in scenario 2, we may affirm that, in both scenarios, a single node failure has thrown away some working hours of the cluster. It is easy to see that a fault-tolerant architecture is necessary in order to mitigate the cost caused by the loss of the computational time. Furthermore, it is reasonable to assume that the user will prefer a transparent fault-tolerant architecture that does not demand any modification in her/his algorithm, instead of worrying about how to implement fault tolerance in his/hear algorithm.

Finally, such scheme must be flexible in terms of internal structures and operational parameters, in order to offer a wide range of configurations. This would help the users and the system administrators to adjust in the system parameters in order to achieve the best system performance.

1.2 RADIC fundamentals

RADIC is a fault-tolerant architecture that uses a rollback-recovery protocol to assure that a parallel application running can correctly finish in spite of faults in the

nodes of the parallel computer in which it is running. The core of RADIC is a fully distributed fault-tolerant controller, which bases its operation on two groups of processes: protectors and observers.

Protectors are processes that RADIC creates in order to implement the fault detection mechanism, the stable storage and the recovery mechanism required by the rollback-recovery protocol. There is one protector process running in every node of the parallel computer. Observers are processes that RADIC creates in order to implement checkpoint and message-log mechanisms. There is one observer process attached to every process of the parallel application.

In Figure 1-1, we depict a brief example of how the elements of the RADIC controller work to manage faults in a parallel computer. The figure represents a draft of four nodes in a cluster. The P_i elements are application processes, The T_i elements are protectors (one for each node N_i), the O_i elements are observers (one for each process.)

The Figure 1-1a depicts some nodes of a typical cluster. We can see that a protector T_i monitors a successor T_{i+1} and is monitored by an antecessor T_{i-1} (the arrows indicate the protection relationship). An observer O_i uses its antecessor protector to store checkpoints and message-logs of its process Pi.

If a failure occurs, the antecessor of the faulty node detects it and recovery the faulty processes in its node. The Figure 1-1b represents cluster the after the completion of the recovery procedure started by the failure in node N_{i+1}. The node N_{i+1} has failed; thus, T_i has detected the failure and has recovered P_{i+1} in N_i.

In Figure 1-1, we can see that application processes continue the computation after a failure. The RADIC controller has transparently performed all the fault tolerance procedures required to assure that the application will correctly finish, using the same resources available to the parallel application.

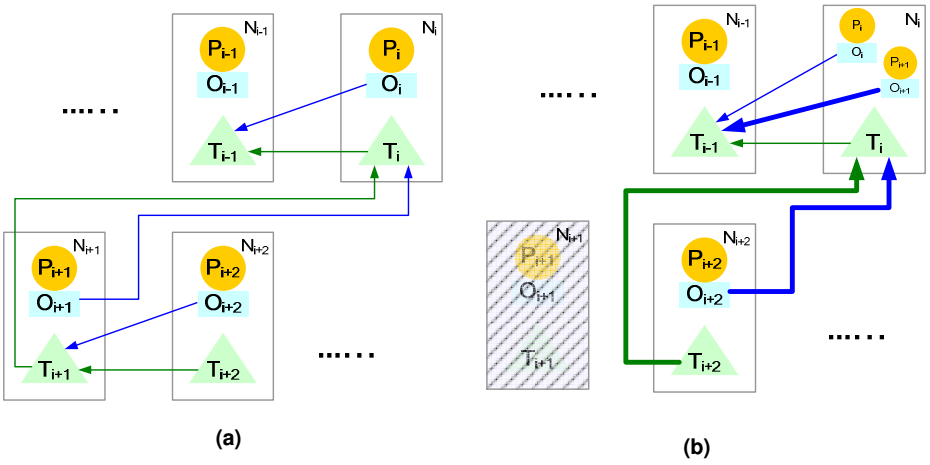


Figure 1-1: (a) A failure free cluster; (b) The same cluster after the management of a failure in node N_{i+1}.

1.3 How RADIC increases the availability of a system

The word *availability*, in terms of computer systems, is a measure of the time that the computer operates correctly. A simple equation serves to calculate the availability of a system [Marcus and Stern, 2003]:

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

In Equation 1, A is the degree of availability expressed as a percentage of the operation time, *MTBF* is the *mean time between failures* and *MTTR* is the *maximum time to repair*.

Availability is an important measure for critical systems. A scientific user needs that the computer is fully available to him/her, since his/her program begins until the program completes. In terms of parallel computers, this means that all components of the system must be available for the parallel program while it is executing, i.e., an availability of 100%. Theoretically, there is only one way of assuring an availability of 100%: reducing the MTTR to zero or increasing the MTBF to infinite.

Since a system with infinite MTBF is unfeasible, we shall focus on the MTTR. We can evaluate the MTTR as the maximum time a failure will exist in the system, i.e., the *failure's lifetime*. The *lifetime* of a failure has two parts: the time required detecting the failure and the time required fixing the failure in order to return the system to a full operational state. Here is where the schemes for fault tolerance play the major role. Does not matter in which level a fault-tolerant architecture is implemented, application level or system level, its more important role is to assure that the parallel application will correctly finishes independent of failures in the nodes of the parallel computer. In other words, the fault tolerance system strongly influences the MTTR.

Any fault-tolerant architecture that can manage failures without human intervention, i.e., automatically and transparently to the user or the system administrator, greatly reduces the MTTR. On the other hand, the MTTR of a non-automatic scheme is unpredictable because it depends on some human intervention.

RADIC operates automatically, i.e., without human intervention, in order to increase the availability by reducing the MTTR. A fault-tolerant system using RADIC has a MTTR lower than a fault-tolerant system using a non-transparent fault tolerance mechanism. Furthermore, because the RADIC mechanism bases on a group of atomic procedures, it is possible to calculate the MTTR of the system based on the specifications of the parallel computer structure.

1.4 How RADIC interacts with the application

Typically, there are two kinds of fault-tolerant architectures for parallel computers: fault tolerance in the application/algorithm level [Geist, 2002], and fault tolerance in the system level [Gropp and Lusk, 2004]. Other solutions mix both approaches, letting a part of the fault tolerance functions for the application and another part to the system.

Any fault-tolerant architecture has advantages and disadvantages according to the target system in which it works. A fault-tolerant architecture implemented in the

application level is independent of the cluster structure. However, it forces the programmer to concern about how his/her application will manage faults and demands a large effort in terms of algorithm development and software engineering.

On the other hand, fault-tolerant architectures implemented in the system level liberate the programmer of the concerning about fault tolerance, but such schemes demand a large effort to integrate them with the cluster architecture.

RADIC uses the system implementation level in order to be fully transparent to the parallel application. On the other hand, since RADIC recovers the faulty processes in the survivor nodes of the parallel computer, the parallel application may collapse if the structure that remains after a fault is insufficient to attend the requirements of the application.

Another interesting approach is to evaluate the Average Computational Capacity (ACC) of a system using RADIC, as defined by Koren and Krishna [Koren and Krishna, 2007]. The ACC is a function of how good the parallel application uses the available processors in a system. Therefore, the ACC depends on the probability $P_i(t)$ that exactly i processors, from the N total processor of the parallel computer, are operational at time t, and of the function $C(i)$ that defines how good the application adapts to the i available processors, as described in equation 2.

$$ACC = \sum_{i=1}^{N} C(i) P_i(t) \qquad \mathbf{2}$$

RADIC assures that all processes of the application will continue to execute after a failure, i.e., it creates a virtual machine capable to execute all process of the parallel applications in despite of node failures. Therefore, assuming that from the application point of view $P_i(t)$=1, the impact of the fault over the application is consequence of the how good the application adapts to the structure of the parallel computer after a failure.

1.5 RADIC and Message-passing

In the world of the parallel computers applied to computational science, the message-passing paradigm is currently the standard paradigm used by programmers in order to create parallel applications. In the last recent years, the MPI standard [Argonne National Laboratory, 2007a] has taken the place of the PVM standard [Oak Ridge National Laboratory, 2007] as the message-passing standard for parallel programming in clusters.

The MPI standard follows the fail-stop semantic, i.e., if any part of the parallel computer's structure fails then the application will stop. Therefore, according to the features that we have analyzed in paragraph 1.2, in order to a parallel program using MPI completes the parallel computer must be 100% available during the program's execution.

Now, let us adapt the concept of availability in order to better relate this concept with parallel applications that use message-passing. The goal of a fault-tolerant architecture is to increase the availability of the parallel computer for the parallel application. MPI programs demands 100% of availability, what means that MTTR (Equation 1) must be null. Therefore, for MPI programs, any fault-tolerant architecture must assure that the MTTR is null.

However, a null MTTR is unfeasible since the lifetime of a fault is never null. Therefore, the only way we achieve 100% parallel computer's availability for a parallel MPI program is masking the fault. To mask the fault means that an automatic fault-tolerant architecture must manage the parallel computer's structure in such a way that the parallel MPI program is not aware that the failure has occurred.

The masking mechanism may use two strategies. The first is to regularly save (automatically or not) the parallel application's state. Therefore, in case of a failure, the user can wait for the repairing of the parallel computer and restart the application from its last state saved. In this solution, the parallel application will always execute with all parallel computer's structure available. However, one cannot assure how long the execution will take in case of a failure because the MTTR is unpredictable.

The second strategy is to endow the fault tolerance mechanism with the capability of adjust the distribution of the parallel application processes to the parallel computer in case of a failure. Such strategy requires an automatic mechanism in order to detect faults, to adapt the parallel application to the new computer's structure and to assure that the parallel application correctly ends. In such strategy, the parallel programs never stop, but its execution takes a longer time if failures occur.

RADIC implements a fault-masking algorithm so the RADIC processes will calculate the location of a recovered process. The algorithm is part of the distributed controller for fault tolerance implemented by RADIC. With this algorithm, programs using MPI may continue their execution even if some nodes fail, because the RADIC architecture will assure that the faulty processes will recover in a different node and will resume their execution from a previous state.

1.6 Organization of this dissertation

This dissertation contains seven chapters. In the next chapter, we discuss the fault tolerance strategies for message-passing systems and explain the reasons to select the strategy used in RADIC.

Chapter 3 evaluates the recent fault tolerance message-passing implementation, analyzing their strong and weak points. Our idea is to show that none of them is capable to attend simultaneously to the four fundamental features offered by the RADIC.

Chapter 4 presents the concepts and describes the operation and the elements of the RADIC Architecture. The Chapter 5 talks about RADICMPI, a practical implementation of the RADIC ideas using the MPI standard. RADICMPI served as a test platform for the functional validation of the RADIC Architecture, which we describe in Chapter 6.

Chapter 7 presents the methodology and describes the experiments conducted with RADICMPI in order to evaluate the practical consequences of using RADIC in a system. Finally, in Chapter 8 we state our conclusions and suggest future works with the RADIC architecture.

Chapter 2

Fault Tolerance in Message-Passing Systems

Any scheme made to manage faults in parallel computers rests on some kind of redundancy. For message-passing systems, the traditional method used to implement redundancy is rollback-recovery.

The main difference between the different rollback-recovery protocols is the efficiency that they present in the absence and in the presence of failures. In this chapter, we evaluate such protocols in order to choose one that better adjusts to the requirements of scalability, transparency, flexibility and decentralization.

Rollback-recovery is a well know and mature technique used to manage faults in applications using message-passing and running on parallel computers. In this technique, when a failure is detected the application must rollback to a previous consistent state and restarts the execution. The rollback recovery mechanism saves all computation done until the moment in which the fault occurs in order to avoid that the application must restart from its beginning after a failure.

Any rollback-recovery strategy requires a set of activities in order to assure that the system can handle faults and that the application can correctly finishes in spite of faults. Such set of activities differentiates the operational phases of the several rollback-recovery protocols.

There are two major groups of rollback-recovery protocols for message-passing systems: checkpoint based and log based (Figure 2-1.) The first group uses only checkpoint of the application's processes. The second group includes the log of

determinant events in order to improve the efficiency of the rollback-recovery mechanism.

In the search of a rollback-recovery that satisfy our requirements of scalability, transparency, flexibility and decentralization; we first established the fundamentals of rollback-recovery in message-passing systems (paragraph 2.1). Next, we evaluated the two major protocol groups (paragraphs 2.2 and 2.3). Finally, we compared the protocols in order to select the one that best fulfill our requirements.

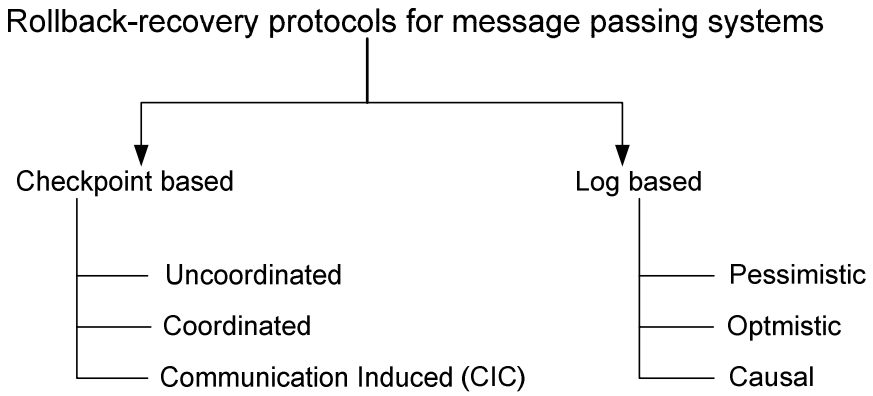


Figure 2-1: Types of rollback-recovery protocol for message-passing systems

The theoretical bases of this chapter follows the surveys made by Elnozahy [Elnozahy, *et al.*, 2002] and by Kalaiselvi & Rajaraman [Kalaiselvi and Rajaraman, 2000]. We selected these two texts because they present two complementary ways of describing the rollback-recovery protocols and because they have served as a basic reference in many publications about fault tolerance.

When more formal approaches were necessary in order to explain some concepts, we used the classical books of Jalote [Jalote, 1994] and Anderson & Lee [Anderson and Lee, 1981].

2.1 Rollback-Recovery Fundamentals

The bases of rollback-recovery rely on the periodical recording of the state of the parallel application (represented by the state of every process and every

communication channel). Upon a failure, the system "rolls back" to a previous state and resumes the execution. There are different strategies for deciding how to record the system state and how the system resumes its execution after a failure.

The key of any rollback-recovery scheme are the checkpoints. Checkpoints are intermediate states of a process that are stored in some memory element. The checkpoints must survive to a fault in the process so such process can restart from a previous state (a previous checkpoint).

One important matter in rollback-recovery is to decide about which strategy the system should use to take the checkpoints. For example, a fault-tolerant architecture based on rollback-recovery may use coordinated or uncoordinated checkpoints; each strategy has advantages and disadvantages according to the application behavior. Another important matter is how the cluster organization will interact with the fault tolerance strategy. For example, a system that stores all checkpoints in a central server will suffer more performance degradation as the number of nodes increases.

Any rollback-recovery scheme considers a distributed system as a collection of application processes that communicate through a network. Each process achieves fault tolerance by using a stable storage device, which must survive all tolerated failures, to save recovery information periodically during failure-free execution. Upon a failure, a failed process uses its saved information to restart the computation from an earlier state, thereby reducing the amount of lost computation.

A *rollback-recover protocol*, also called *fault-tolerance protocol*, manages the rollback-recovery functioning. In simple protocols, the recovery information can include only the states of the participating processes, called *checkpoints*. More sophisticated recovery protocols may require additional information, such as logs of the interactions with input and output devices, events that occur to each process, and messages exchanged among the processes.

In this thesis, we define a distributed parallel application based message-passing as a collection of processes that cooperate through messages sent via some network. Figure 2-2 shows a symbolic representation of a parallel application with four

processes. In that figure, a line represents a process execution and the arrow lines between the processes represent messages.

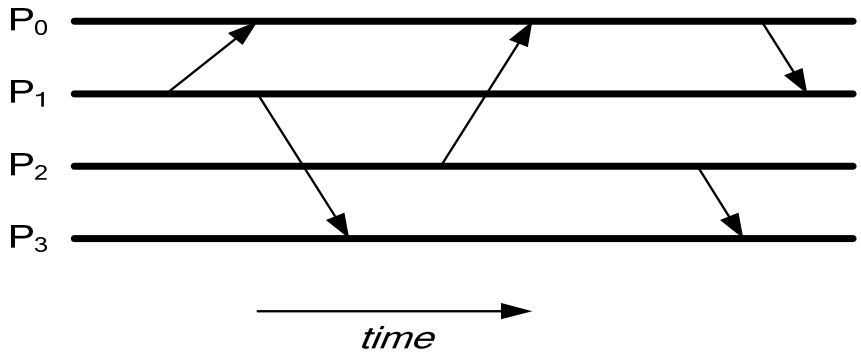


Figure 2-2: An example of a distributed application using message-passing

Each process's execution is a sequence of state's intervals triggered by a nondeterministic event. For the sake of the parallel applications used in this thesis, such triggers are the messages that arrive to the process. Inside each state interval the execution is deterministic, i.e., if a process starts from the same state and is subjected to the same nondeterministic events at the same locations within the execution, it will always generate the same output.

A rollback-recovery protocol should assure that the internal state of a system after a failure should be consistent with the observable behavior of the system before the failure [Strom and Yemini, 1985]. In order to satisfy this condition, a rollback-recovery protocol must save information about internal iterations among processes to assure that the system restart from a *consistent state* after a failure. Consistent system state and other important concepts follow.

2.1.1 Consistent System State

The state (or the global state) of a distributed system is represented by the state of each communication channel and each process in a given time. Therefore, one can model a distributed system as a sequence of system states in which some kind of

event triggers the transition from one state to another. Multiple events may occur in a given state and an event's presence, or its absence, depends on the particular execution of the system. This means that even if we restart a system many times from the same initial state and we give to it the same inputs, the system can follow a different sequence of states, and each sequence is a valid execution of the system. A consistent system state (or consistent global state) is one that may occur during a failure-free correct execution, i.e., one that could occur in one of the possible sequences of states in the system.

Checkpoints correspond just to processes' states and have no information about the states of the communication channels. Therefore, in order to record a consistent global state of a process, whenever the state of any process indicates that it has sent a message to another process, the state of the receptor must indicate that it has received this message [Chandy and Lamport, 1985]. This requirement is the major condition to the implementation of techniques to record global states of distributed systems.

2.1.2 Recovery Line

Figure 2-3 shows an example of two system's states. In this figure, the bars indicate the checkpoints of the processes. In Figure 2-3a, the system's state indicates that process P_2 has sent the message m_3 but process P_0 has not yet received it. In such situation, if P_0 fails and rolls back to the state represented by the checkpoint C_{00}, then the system goes to an inconsistent global state because the state of P_2 indicates that it has sent m_3 to P_0 and the state of P_0 does not indicate that it has received m_3.

The consistency of the global system's state depends on how the recovery protocol deals with in-transit messages. If the rollback-recovery protocol assumes that the message channels are reliable, then the global state in Figure 2-3a is inconsistent and m_3 is a *lost message*. On the other hand, if the rollback-recovery protocol assumes that the message channels are unreliable, this global state is consistent and m_3 is an *in-transit message*.

The example of Figure 2-3b also shows an inconsistent state because the state of process P_0 considers that P_0 has received m_3 but the state of process P_2 does not considers that P_2 has sent m_3. In this case, m_3 is an *orphan message*.

If a set of checkpoints of the system, i.e., a system global state, satisfy the following restrictions, then it is a *recovery-line* and the recovery protocol can use it as a recovery point [Jalote, 1994]:

a) The set contains only one checkpoint for each process;

b) For a given set, there is no send-event succeeding the recovery point of a sender process P, whose equivalent receive-event in the destination process Q occurs before the recovery point of Q in the set (no orphan messages).

c) For a given set, there is no send-event preceding the recovery point of a sender process P, whose equivalent receive-event in the destination process Q occurs after the recovery point of Q in the set (no lost messages).

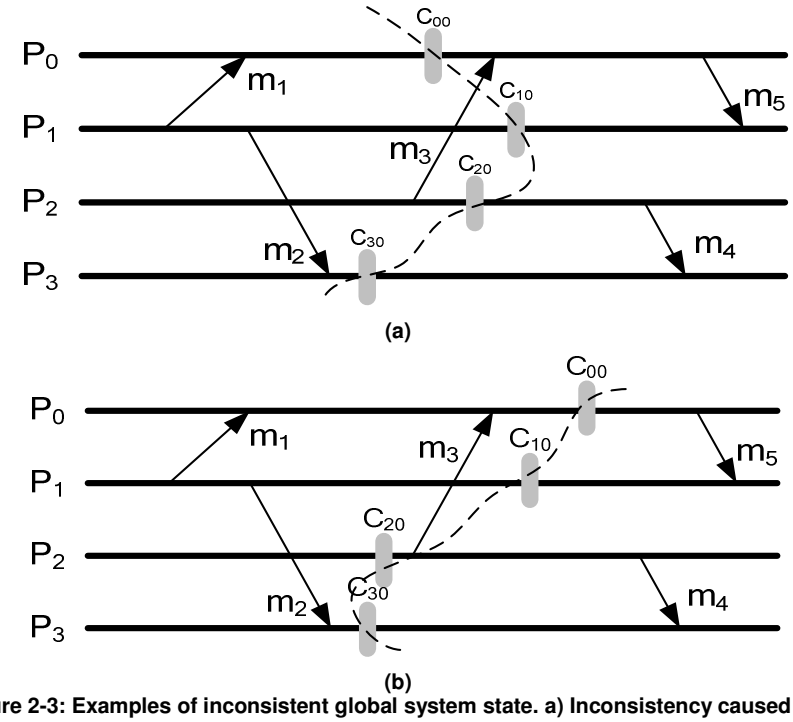


Figure 2-3: Examples of inconsistent global system state. a) Inconsistency caused by a lost message ; b) Inconsistency caused by a orphan message

2.1.3 In-transit Messages

In-transit messages are messages registered in the sender's state but not in the receptor's state. There are two ways to implement the rollback-recovery protocol in order to deal with in-transit messages.

- **An implementation based on a reliable communication protocol**. In this case, a reliable communication protocol ensures the reliability of message delivery during failure-free executions, but it cannot ensure the reliability of message delivery in the presence of failures. For instance, a conventional communication protocol will generate a timeout and inform to the sender that it cannot deliver the message whenever an in-transit message is lost because the intended receiver has failed. Once the fault tolerance mechanism recovers the receiver process, the system must mask the timeout from the sender process and make the in-transit messages available to the intended receiver process after it recovers.
- **An implementation based on an unreliable communication channel**. In this case, the recovery protocol need not handle in-transit messages in any special way. Indeed, the recovery protocol cannot distinguish the in-transit messages lost because of process failures from those lost because of communication failures in an unreliable communication channel. Therefore, the loss of in-transit messages due to either communication or process failure is an event that can occur in any failure-free, correct execution of the system.

2.1.4 Domino Effect

The *domino effect* [Randell, 1999; Russell, 1980] appears when the processes of a distributed application take their checkpoints in an uncoordinated manner. Figure 2-4 shows an execution in which processes take their checkpoints (represented by red bars) without coordinating with each other.

Each process starts its execution with 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 2-4 represents the recovery line of the system in case of a failure in P_0.

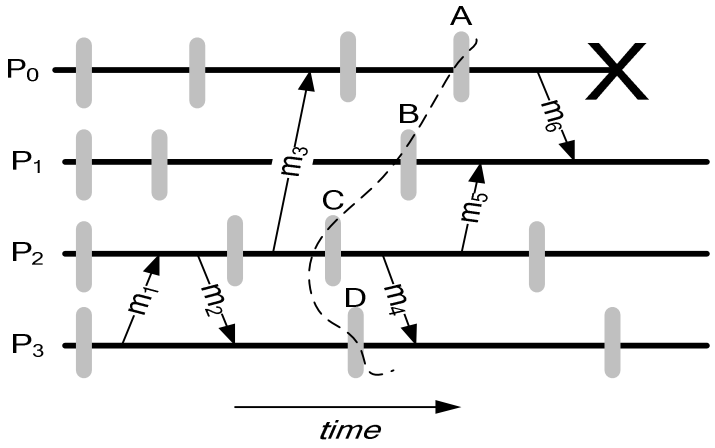


Figure 2-4: Domino effect

2.1.5 Event Logging

Event logging is the strategy used to avoid the domino effect caused by uncoordinated checkpoints. Log-based rollback-recovery protocol is family's name of the 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.

2.1.6 Stable Storage

Rollback recovery protocols use stable storage to save checkpoints, message logs, and other recovery-related information. The concept of stable storage in rollback recovery protocols is an abstraction and should not be confused with the disk storage used to implement it. Stable storage must ensure that the recovery data persist through the tolerated failures and their corresponding recoveries. This requirement allows different implementation's techniques of stable storage according to the number of failures a system can tolerate. A classification of such techniques according to the number of faults follows [Elnozahy*, et al.*, 2002]:

d) For only a single failure, a stable storage may consist of the volatile memory of another process [Borg*, et al.*, 1989];

e) For an arbitrary number of transient failures, stable storage may consist of a local disk in each host;

f) For non-transient failures, stable storage must consist of a persistent medium outside the host on which a process is running. One possibility is to implement this scheme using a network file system or to replicate the recovery information of a node into a different node.

2.1.7 Garbage Collection

The amount of recovery information (checkpoints and event logs) increases as the application progress. Since the amount of storage required to store the recovery information influences the cost of the fault tolerance scheme, some strategy is necessary in order to reduce the amount of storage, i.e., the storage cost, consumed by discarding useless information. From time to time, a subset of the stored information becomes useless for recovery. *Garbage collection* is the deletion of such useless recovery information from the system.

A typical approach to perform garbage collection is to identify the most recent recovery line, and discard all information relating to the events that occurred before that line. For example, processes that coordinate their checkpoints to form consistent states will always restart from the most recent checkpoint of each process. Therefore, the protocol can discard all previous checkpoints.

Garbage collection is an important pragmatic issue in rollback-recovery protocols for two reasons. First, because running a special algorithm to discard useless information incurs an extra overhead. Second, because the garbage collection reduces the amount of hardware resources used by the fault tolerance mechanism as the parallel application runs.

Different rollback-recovery protocols differ in the amount and nature of the recovery information they need to store in the stable storage. Therefore, each protocol differs in the complexity and invocation frequency of its garbage collection algorithm.

2.2 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, i.e., the most recent recovery line. Since such protocols do not rely on the PWD assumption, they do not care about nondeterministic events, i.e., 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.

The next topics explain the three categories of the checkpointing strategies used by the checkpoint-based protocols: *uncoordinated*, *coordinated* and *communication-induced*.

2.2.1 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] make 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.

During failure-free operation, the processes calculate the interdependencies among theirs checkpoints using a technique explained by Bhargava and Shu-Renn [Bhargava and Shu-Renn, 1988]. In this technique, each process piggyback checkpoint information in each message sent to another process. The destination process uses this information to calculate the dependency between it and the sender. If a failure occurs, the recovering process initiates the rollback by broadcasting a dependency request message to collect all the dependency information maintained by

each process. Upon a process receives this message, it stops its execution and replies with the dependency information saved on the stable storage as well as with the dependency information, if any, that is associated with its current state. The initiator then calculates the recovery line based on the global dependency information and broadcasts a rollback request message containing the recovery line. Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line.

There are two similar approaches to calculate the recovery line: the *rollback-dependency graph* [Bhargava and Shu-Renn, 1988] and the *checkpoint graph* [Wang, 1997]. Both are equivalent in that they always produce the same recovery line. By using such methods, one can determine the most advanced recovery line. This greatly simplifies the implementation of the garbage collection algorithm in order to liberate the space used by checkpoints located before this line.

2.2.2 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.

Tamir & Sequin propose a typical scheme that uses a central coordinator in order to start the checkpoint mechanism [Tamir and Sequin, 1984]. In this scheme, the coordinator broadcasts a request asking to all other process to make a checkpoint. Upon receiving a checkpoint request, the process suspend its execution, takes a *tentative* checkpoint and sends an acknowledge message to the coordinator. After the

coordinator receives the acknowledge messages from all process it broadcast a commit message. Upon receiving a commit message, all processes replace the old checkpoint with the *tentative* checkpoint and resume the execution.

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.

Another alternative is to use loosely synchronized clocks in order to facilitate checkpoint coordination. By using loosely synchronized clocks, all processes can trigger checkpoint actions at approximately the same time without a checkpoint initiator [Cristian and Jahanian, 1991; Tong, *et al.*, 1992].

The scalability of coordinated checkpointing is weak because all processes must to participate in every checkpoint. A strategy to reduce the number of processes involved is to take new checkpoints for only those processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint [Koo and Toueg, 1987].

2.2.3 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, i.e., 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* [Netzer and Jian, 1995]. 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]. An implementation can be found in the work of Hélary [Helary, *et al.*, 1997b].

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. Baldoni [Baldoni, *et al.*, 1998], Russel [Russell, 1980] and Wang [Wang, 1997] describe different implementation's methods of this type protocol.

2.2.4 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.

2.3 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 [Strom and Yemini, 1985]. 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 2-4, 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*.

An implementation called *K-optimistic* logging protocol is proposed in [Damani, *et al.*, 2003]. However, instead of a new protocol, the authors propose a new way to use the optimistic protocol by applying a constant K that defines a "grade of optimality". By varying K, one may chose the cost-benefit relationship of the message log protocol in terms of overhead in failure free scenarios and in case of failure.

2.3.1 Pessimistic log-based protocols

These protocols assume that a failure can occur after any nondeterministic event in the computation. This assumption is "pessimistic" since in reality, failures are rare. In their most straightforward 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.

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]:

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

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

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

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

Some examples of pessimistic log-based rollback-recovery protocols are found in the works described in [Borg*, et al.*, 1989; Elnozahy and Zwaenepoel, 1992; Johnson and Zwaenepoel, 1987; Juang and Venkatesan, 1991]. These implementations differ in the way they try to reduce overhead penalty under failure-free operation.

2.3.2 Optimistic log-based protocols

These protocols allow the apparition of orphans because to failures in order to reduce the failure-free performance overhead. However, the possibility of apparition of orphans processes complicates the recovery, garbage collection and output commit [Jalote, 1994]. In optimistic protocols [Sistla and Welch, 1989; Strom and Yemini, 1985] every process take checkpoint and message log asynchronously. 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 [Sistla and Welch, 1989]. 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.

2.3.3 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.

2.4 Comparing the rollback-recovery protocols

Table 1 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 2-1 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 2-1, 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 2-1: 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 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.

In this chapter, we presented the rollback-recovery protocols for message-passing systems and justified the pessimistic log-based protocol as the only one that simultaneously supports all four requirements for the modern fault tolerance schemes for parallel computers. In the next chapter, we present some recent fault tolerant message-passing implementations and show that, because of their architecture and rollback-recovery protocol, none of them can simultaneously to support the four key features that we have established for a modern fault tolerance solution.

Chapter 3

Fault Tolerant Message-passing Implementations

In the last chapter, we discuss the theoretical aspects of the rollback-recovery protocols for fault tolerance in message-passing systems. We also justified the adoption of the pessimist message-log protocol as the protocol that simultaneously attends to scalability, transparency, flexibility and decentralization.

Now, we are going to evaluate some legacy and current fault-tolerant message-passing implementations in terms of the same features. We analyzed that none of these implementations simultaneously supports these four of features, what creates an opportunity to the development of a new solution.

The message-passing paradigm is a paradigm largely used for development of parallel programs. The large acceptation of such paradigm has given place to the development of the PVM and MPI standards.

In the last recent years, the MPI standard has become the *de facto* standard for the development of distributed parallel applications using the message-passing paradigm. Therefore, we focused our attention in practical MPI implementations.

Because MPI uses a fail-stop semantic, several fault tolerance MPI implementations based on rollback-recovery have appeared, but we have verified that none of them could simultaneously attend to the four major requirements that we are looking for.

In this chapter, we first discuss some practical aspects related to the operation of fault tolerance mechanisms in message-passing systems. Then, we present traditional

and current fault-tolerant MPI implementation and summarize their features in order to show that none of them simultaneously offers the same set of features we are looking for: transparency, decentralization, scalability and flexibility.

3.1 Practical aspects

Any fault-tolerant architecture interacts with the application and the parallel computer structure. Such interaction yields practical consequences we should consider when evaluating a fault tolerance implementation. We divided these practical aspects in three main groups as shown in Figure 3-1.

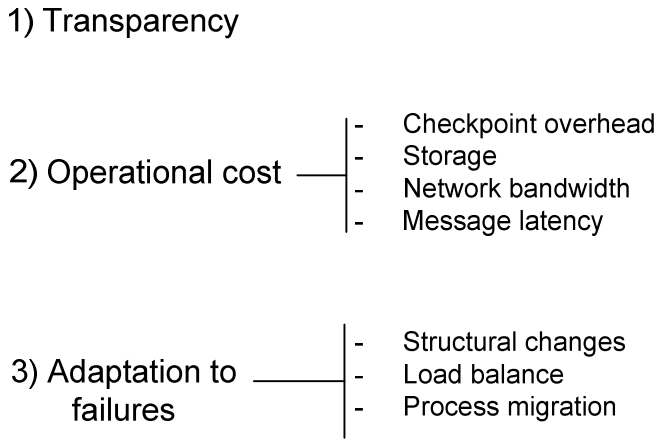


Figure 3-1: Practical aspects of fault tolerance for message-passing systems

3.1.1 Transparency

An important concern is how much software development effort is necessary in order to enjoy the fault tolerance benefits.

To embed fault-tolerance in the application yields the best benefits because the developer will have full control of how fault-tolerance will interact with the application and the cluster architecture. However, the cost of embedding fault tolerance in an application is worthy only when such application executes many times and when it is necessary to achieve the best possible performance for the application.

To embed fault-tolerance in the system is very interesting when the application development time is the main requisite. The common approach in such cases is to offer to the user, a library that implements the fault tolerance functionality in the MPI API. The development cost is low because the user just needs to recompile its applications with the library in order to gain access to the fault tolerance. The disadvantage of this method is that the application performance depends on the fault tolerance implementation and the way the fault tolerance interacts with the cluster architecture.

3.1.2 Operational cost

Any fault-tolerant architecture will have some associated costs. These costs relates to the resources consumed by the fault tolerance mechanism. For systems using fault tolerance schemes based on rollback-recovery, the following elements will influence the operational cost:

Checkpoint overhead

Checkpointing strongly contributes for computation time enlargement because of the cost of writing the process' state into the stable storage. Strategies as concurrent checkpointing [Goldberg, 1990] in combination with incremental checkpointing [Plank, *et al.*, 1995] can greatly reduce the checkpoint overhead. Furthermore, incremental checkpoint also reduces the storage space consumed by checkpoints.

The problem of determining the optimal checkpointing frequency and placement is a kind of optimization's problem subjected to constraints [Chandy, 1972]. However, the optimality of checkpoint placement is rarely an concern in practical systems because in practice the overhead of checkpointing is usually negligible, unless the checkpointing interval is relatively small, as observed by Elnozahy at [Elnozahy, *et al.*, 1992].

Stable storage and storage space consumed

Magnetic disks are still the preferred media used for logs and checkpoints and are the largest contributor for performance overheads. Communication overhead is much lower in comparison with the disk storage overhead.

The amount of space required by checkpoints and message logs strongly influences the efficiency of the fault tolerance scheme. The checkpoint interval and the application behavior (message pattern and processes state sizes) are the major parameters that determine the storage space consumed by the fault tolerance.

Network bandwidth consumed

Besides the normal application messages, many control messages will exist because of the rollback-recovery protocol and the fault detection scheme. Furthermore, taking into consideration that checkpoint and message logs of a process are saved outside the machine where the process is, delivering of these new elements also will contribute for a reduction of the total network bandwidth.

Message latency enlargement

Schemes using message-log increase the message latency because of the necessity of saving the logs into a stable storage. The total overhead will depend from the message-log protocol and the application's message pattern.

3.1.3 Adaptation to failures

After a failure, the whole system changes to a new configuration. This new configuration is consequence of the new the structure of the parallel computer and how the parallel application will interact with such new structure. A fault directly influences three major system's feature, as explained bellow.

Structural changing

After a node failure, the cluster configuration will depend of the remaining resources. The user can decide to reserve nodes for faulty node replacement; however, this solution can be inefficient if the probability of failures is small or if there are few nodes in the system, because the spare nodes do not participate in the computation.

A common solution in practical systems is to use all available resources and to recover the faulty processes in the remaining nodes of the cluster. This strategy leads to problems of load balancing and location-independency of the distributed application processes.

Load-balancing

The load-balancing problem depends of the ratio between the number of faulty nodes and the number of total nodes in the cluster. Load-balancing includes an additional overhead and the decision of include it or not in the fault-tolerant architecture is influenced by factors such as total application runtime, cluster MTBF, i.e. probability of failures, and the number of nodes in the cluster.

Migration of recovered processes

Rollback-recovery protocols should prevent the application processes from acquiring dependency of any location. The system must mask the actual location of the process (network identity, addresses and ports) for the communication protocol.

3.2 Recent fault-tolerant message-passing platforms

In the last ten years, several message-passing implementations included fault tolerance capabilities. Because MPI has become the *de-facto* standard for message-passing systems, we focused our attention in the fault-tolerant MPI implementations (Table 3-1). Some of these implementations do offer the requirements necessary to attend the fault tolerance expectations of the modern parallel computers. Nevertheless, at the end of this chapter we present a comparison that shows that none of them attends simultaneously to the four benefits we have established: transparency, flexibility, scalability and decentralization.

Table 3-1: Time line of the recent fault-tolerant MPI implementations

1996	1999	2000	2001
CoCheck	Starfish and Egida	FT-MPI	MPI/FT
2002	**2003**	**2005**	**2006**
MPICH-V1	MPICH-V2 and LA-MPI	LAM/MPI	MPICH-PCL

3.2.1 CoCheck

CoCheck tuMPI [Stellner, 1996] addresses fault tolerance application level and was one of the first efforts to incorporate fault tolerance into MPI. CoCheck used the Condor [Litzkow, *et al.*, 1988] library to checkpoint and then, if necessary, restart and rollback all MPI processes. Its main drawback was the need to checkpoint the entire application, which is prohibitively expensive in terms of time and scalability for large applications.

3.2.2 Starfish MPI

Unlike CoCheck that relies on Condor, Starfish [Agbaria and Friedman, 1999] uses its own distributed system to provide built in checkpointing. The main difference with CoCheck is that StarFish uses strict atomic group communication protocols, built upon the Ensemble system, in order to handle communication and manage state changes. This mechanism avoids the message flush protocol of CoCheck.

3.2.3 Egida

Egida [Rao, *et al.*, 1999] is an object-oriented toolkit designed to support transparent rollback-recovery. Egida specifies a simple language that serves to express arbitrary rollback recovery protocols. From this specification, Egida automatically synthesizes an implementation of the specified protocol by gluing together the appropriate objects from an available library of building blocks.

3.2.4 FT-MPI

FT-MPI [Fagg and Dongarra, 2000] has been developed in the frame of the HARNESS [Micah, *et al.*, 1999] metacomputing framework. The goal of FT-MPI is to offer to the end-user a communication library providing an MPI API, which benefits from the fault-tolerance in the HARNESS system. FT-MPI implements the whole MPI-1.2 specification, some parts of the MPI-2 document and extends some of the semantics of MPI for giving the application the possibility to recover from failed processes.

FT-MPI claims it survives the crash of n-1 processes in an n-process job. The processes can be re-spawn if required. However, it is still the responsibility of the application to recover the data-structures and the data on the crashed processes. FT-MPI uses collective operation to assure

FT-MPI contains the notion of two classes of participating processes within the recovery: *Leaders* and *Peons*. The leader tasks are responsible for synchronization, initiating the Recovery Phase, building and disseminating the new state atomically. The peons just follow orders from the leaders. In the event that a peon dies during the recovery, the leaders will restart the recovery algorithm. If the leader dies, the peons will enter an election controlled by the name service using an atomic test and set primitive. A new leader will restart the recovery algorithm. This process will continue either until the algorithm succeeds or until everyone has died [Fagg*, et al.*, 2005].

3.2.5 MPI/FT

MPI/FT [Batchu*, et al.*, 2001] uses a centralized coordinator that manages the fault tolerance mechanism in a transparent way. The coordinator runs in an nMR (n-Modular Redundancy) mode and has to monitor the application progress, act as a virtual channel routing every message transfer between process (much like the memory channels in MPICH-V1), recover faulty processes and replay messages from log to it until the faulty process reach a consistent state. Because of the centralized coordination scheme, the scalability of MPI/FT is limited.

3.2.6 MPICH-Vx

MPICH-V [Bouteiller*, et al.*, 2006] is composed by a communication library based on MPICH and a runtime environment. MPICH-V runtime environment is a complex environment involving several entities: Dispatcher, Channel memories, Checkpoint servers, and Computing/Communicating nodes. 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.

MPICH-V comes in three flavors. MPICH-V1 [Bosilca, *et al.*, 2002] is designed for 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, but requires a larger bandwidth for stable components to reach good performance.

MPICH-V2 [Bouteiller, *et al.*, 2003] is designed for homogeneous network large-scale computing (typically large clusters). Unlike MPICH-V1, it requires a small number of stable components to reach good performance on a cluster. It uses uncoordinated checkpoint protocol associated with sender based pessimistic message logging. Instead of channel memories, MPICH-V2 uses event loggers to assure the correct replace of messages during recovers. The computing node now keeps the message-log.

MPICH-VCL 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.

3.2.7 LA-MPI

LA-MPI [Graham, *et al.*, 2003] has two primary goals: network fault tolerance and high performance. Network fault tolerance is achieved by implementing a checksum/retransmission protocol. The integrity of delivered data is (optionally) verified at the user-level using a checksum or CRC. Data that is corrupt (or never delivered) is retransmitted.

LA-MPI offers a lightweight checksum/retransmission protocol, instead of the classic TCP/IP protocol. Such protocol allows the use of redundant data paths in the network leading to a high network bandwidth since different messages and/or message-fragments can be sent in parallel along different paths.

Currently LA-MPI delivers messages in the presence of I/O bus, network card and wire transmission errors, supports network card and path failures and guarantees delivery of in-flight messages after such a failure. Therefore, LA-MPI guarantees of end-to-end network fault tolerance for an MPI application. The fault-tolerance of application processes is not a feature in the current release.

3.2.8 LAM/MPI

The LAM/MPI [Sankaran, *et al.*, 2005] is an MPI implementation that allows MPI applications running to be checkpointed to disk and restarted later. LAM requires a third party single-process checkpoint/restart toolkit for actually checkpointing and restarting a single MPI process. Currently, LAM uses the Berkeley Labs Checkpoint/Restart package (Linux only) in order to implement a coordinated checkpoint protocol.

The approach adopted in LAM/MPI ensures that all the MPI communication channels between the processes are empty when at the checkpoint time. During restart, all the processes resume execution from their saved states, with the communication channels restored to their known (empty) states.

3.2.9 MPICH-PCL

MPICH-PCL is a very recent fault-tolerant MPI implementation that uses MPICH-2 as a standard message-passing platform [Coti, *et al.*, 2006]. MPICH-PCL follows the same architecture model implemented in the MPICH-Vx versions. Therefore, it requires the use of dedicated elements for checkpoint and messaging management.

MPICH2 is the MPI-2 implementation of Argonne National Laboratory [Argonne National Laboratory, 2007c]. MPICH-2 provides a platform to manage dynamic processes in the MPI Communication World. This feature makes MPICH-2 suitable to implement process migration and process recovery mechanisms.

3.2.10 Open MPI

Open-MPI [Gabriel, *et al.*, 2004] is a MPI-2 compliant implementation, produced by the combining technologies and resources from FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI, in order to build an open source MPI library. Open-MPI claims to offer a combination of the best features from each one of these platforms, including fault tolerance for distributed parallel applications.

Currently, no Open-MPI distribution offers fault tolerance capabilities. The authors of Open MPI claim that this feature will be available in parallel projects but not in the trunk implementation [Open-MPI Project, 2007].

3.3 Comparing the fault-tolerant MPI implementations

We compared the fault-tolerant MPI solutions present in this chapter using the four key features that we have elected for a modern fault tolerance scheme. Table 3-2 presents a summary of the implementation analyzed in this chapter. We use a + signal to indicate that the implementation fully attends and +/- signal to indicate that it partially attends to a requirement.

Table 3-2: A comparison of the recent fault-tolerant MPI solutions based on four relevant features required by the modern parallel computers

Solution	Scalable	Fully Decentralized	Transparent	Flexible
Cocheck			+/-	
Starfish			+/-	
Egida			+/-	+
FT-MPI	+		+/-	
MPI/FT			+/-	
MPICH-V1			+	+
MPICH-V2	+		+	+
LA-MPI			+	
LAM/MPI			+	
MPICH-PCL	+		+	+

As one can see, although all solutions are transparent to the user, none of them simultaneously presents the four features. Such lack of a fault-tolerant solution that could simultaneously present all four features has led us to the development of a new architecture, which we present in the next chapter.

Chapter 4

The RADIC Architecture

In Chapter 3, we analyzed several recent fault-tolerant message-passing implementations. The lack of a solution that simultaneously offers scalability, flexibility, transparency and decentralization has guided us in the development of a new fault-tolerant architecture, the Redundant Array of Independent Fault Tolerance Controllers - RADIC.

As any fault-tolerant solution for message-passing systems, the main goal of RADIC is to assure that a parallel-distributed application will correctly finish even if faults occur in some nodes of the parallel computer. Additionally, RADIC simultaneously attends to the four key features described in Table 4-1.

In this chapter, we first present the system model of the RADIC architecture. Then, we describe the RADIC functional phases and functional elements that compose the RADIC fault tolerance distributed controller. Next, we explain the RADIC operation. Finally, we describe the RADIC functional parameters and evaluate the configuration aspects of RADIC.

4.1 RADIC architecture model

RADIC establishes an architecture model that defines the interaction of the fault-tolerant architecture and the parallel computer's structure. Figure 4-1 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.

Table 4-1: The key features of RADIC

Feature	How it is achieved
Transparency	– No human intervention is required to manage the failure – No change in the application code
Decentralization	– No central or fully dedicated resource is required. All nodes may be simultaneously used for computation and protection
Scalability	– The 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

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.

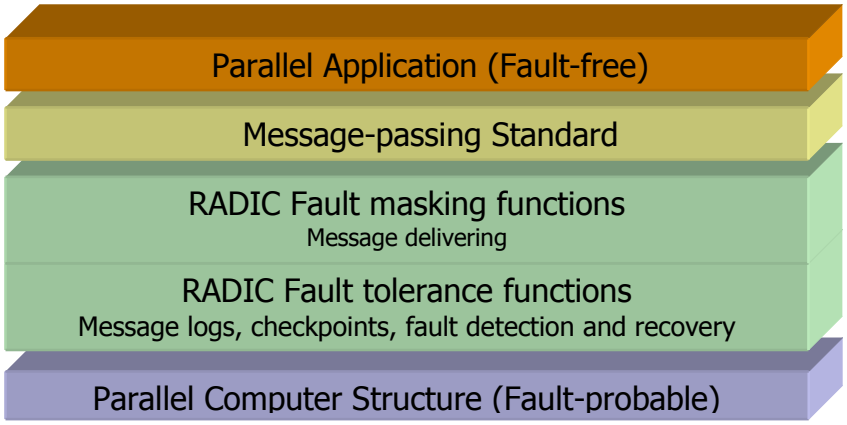


Figure 4-1: The RADIC levels in a parallel system

The operation of RADIC relies on some operational models: the parallel application model, the parallel computer model and the fault pattern. These models establish the premises that rule the operation of the RADIC fault tolerance mechanism.

4.1.1 Distributed parallel application model

A *distributed parallel application* consists of a set of concurrent processes that *cooperate* with each other to perform a task. The processes communicate only using a message-passing standard. Therefore, since our model considers that received messages are the only nondeterministic event, we model each process as a sequence of state intervals, each interval started by a received message.

Execution inside each interval is deterministic, i.e., if a process starts from the same state and receives the same sequence of messages at the same points within the execution, then this process will always generate the same results.

Communication channels and process are both *synchronous* [Jalote, 1994]. By synchronous we mean that whenever an element is working correctly, it always will perform its intended function in a finite and known (or predictable) time bound. Therefore, every communication channel will have a latency bound and every process will execute each of their state intervals in a time bound.

4.1.2 Parallel computer model

The distributed parallel application has P processes, and runs in a parallel computer with N nodes. In failure-free executions, all the N nodes are available. The application will endure a maximum number of failed nodes without collapsing. Processes from failed nodes will recover in some survivor node.

The communication network is fully connected, i.e., a process in a node can send a message to any other process. The network delivers messages in FIFO order. A channel represents the logical connection between two processes. A communication failure between two nodes is the event used to define a node failure.

The minimum structure required to maintain the parallel application running defines the maximum number of failures that the system can bear. Therefore, the requirements of the parallel application, i.e., the user requirements, are determinant to decide the maximum number of failures the system will support. A typical requirement could be the time when the application must complete. In such cases, the

maximum number of faults is one that does not reduce the computational power beyond a minimum that neither compromises the application execution time nor compromises the functionality of the fault tolerance controller.

4.1.3 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.

4.2 RADIC functional phases

In the Chapter 2, we explained the theoretical bases that driven the adoption of the pessimistic log-based protocol for the RADIC rollback-recovery mechanism.

As a transparent fault tolerance system, i.e., a system that can handle faults without human intervention, RADIC automatically performs a group of activities required by its rollback-recovery protocol. Each activity is inside one of the four general procedures required by an automatic fault tolerance mechanism, as described in Table 4-2.

Table 4-2: Operational phases of RADIC

Functional phase	Activity
Sate saving	– Storage checkpoints – Storage message logs – Garbage collection
Failure detection	– Monitor nodes using a heartbeat watchdog mechanism
Recovering	– Restart failed processes from their most recent checkpoints – Replay messages from the message log
Fault masking	– Assure that survivor processes can reach the recovered processes

Such phases occur concurrently with the parallel application execution and do not interfere in its results. Actually, the RADIC implementation operates as a parallel-distributed application that runs concurrently and shares the same resources used by the user's parallel application.

4.3 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.

4.3.1 Protectors

There is a protector process in each node of the parallel computer. Each protector communicates with two neighbors: an antecessor and a successor. Therefore, all protectors establish a protection chain throughout the nodes of the parallel computer. In Figure 4-2, we depict a simple cluster built using nine nodes (N_0-N_8) and the respective protectors of each node (T_0-T_8). The arrows indicate the antecessor←successor relationship.

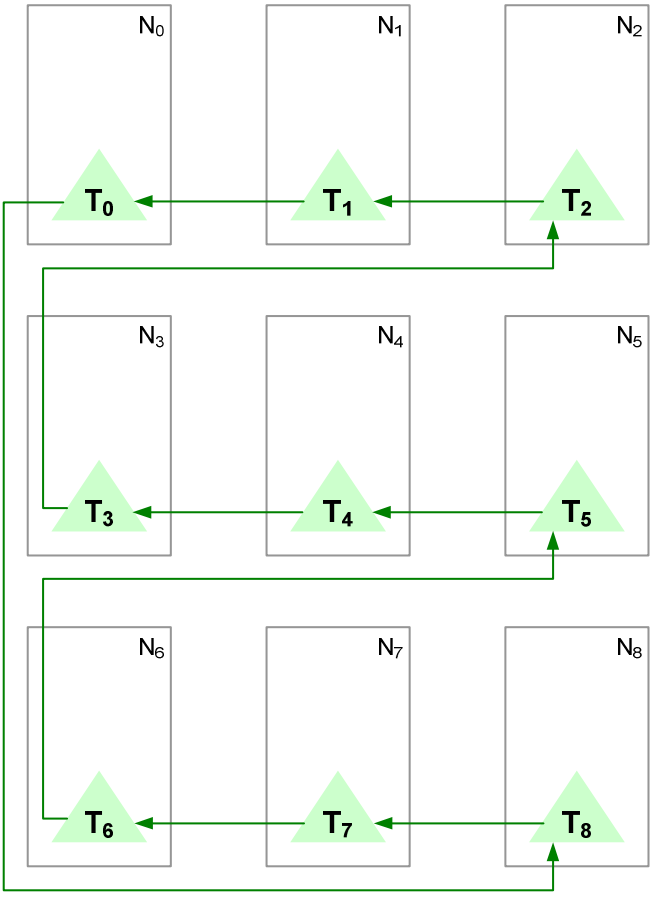


Figure 4-2: An example of Protectors (T_0-T_8) in a cluster with nine nodes. Green arrows indicate the antecessor←successor communication.

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 4-2 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 4-2, 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.

4.3.2 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 message-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 indicating the location of all application processes and their respective protectors and updates this table in order to mask faults.

4.3.3 The RADIC controller for fault tolerance

The collaboration between protectors and observers allows the execution of the phases of the RADIC controller. Figure 4-3 depicts the same cluster of Figure 4-2 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 4-3, 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 end of Chapter 2, 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.

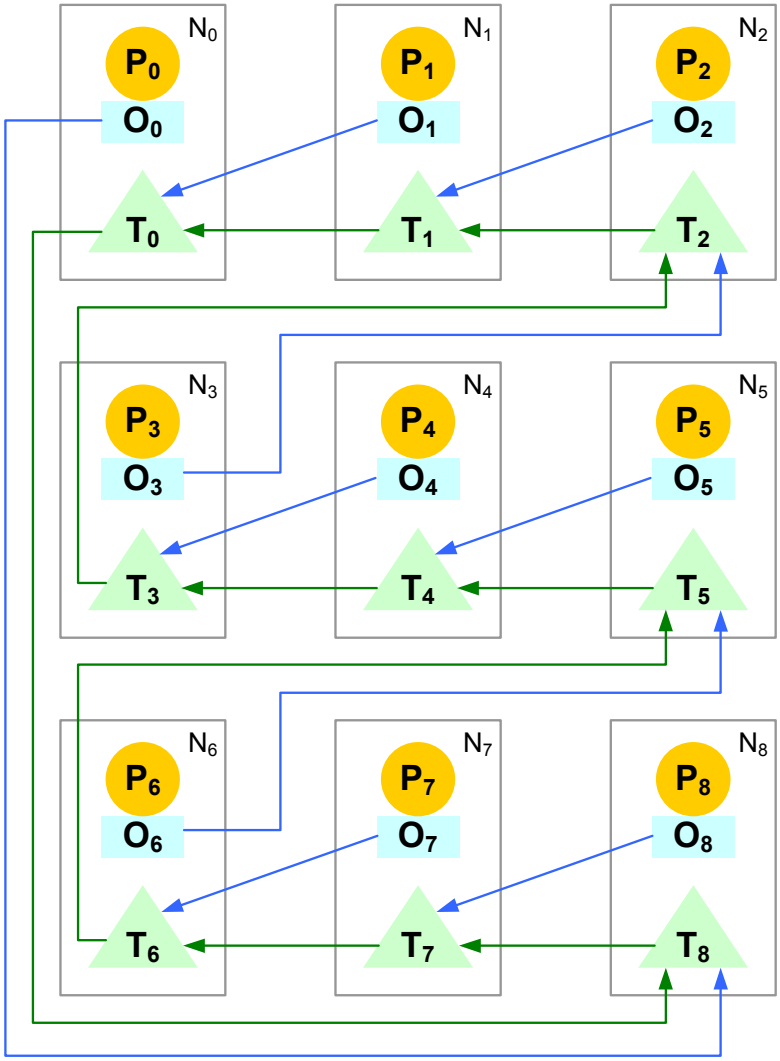


Figure 4-3: A cluster using the RADIC architecture. P_0-P_8 are application process. O_0-O_8 are observers and T_0-T_8 are protectors. O→T arrows represent the relationship between observers and protector and T→T arrows the relationship between protectors.

Table 4-3 explains how each RADIC element acts in each functional phase of the RADIC fault-tolerant architecture using the receiver-based pessimistic log rollback-recovery protocol.

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 in the message-passing mechanism.

Table 4-3: The role of each RADIC element in the phases of fault tolerance

	RADIC element	
Functional phase	**Protector**	**Observer**
State saving	– Receive and store checkpoints and message logs from observers – Make garbage collection	– Take checkpoints and message logs of its process and send them to a protector
Failure detection	– Monitor the neighbors – Establish a new neighbor in case of failure in the current neighbor; Inform local observers	– Detect failures in another application processes – Detect protector's failure – Listen to warnings coming from the local protector
Recovery	– Recover failed processes	– Manage the message log after recovering – Establish a new protector
Fault masking	– Recover the protection chain by reestablish the communication with a new neighbor	– Manage the mapping table

4.4 RADIC operation

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.

4.4.1 Message-passing mechanism

The message-passing mechanism is an observers' attribution. The protectors do not play any role in this mechanism.

In the RADIC message-passing mechanism, an application process sends a message trough 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 4-4 clarifies this process.

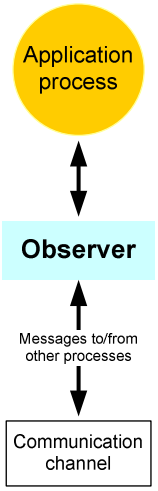


Figure 4-4: The message-passing mechanism in RADIC.

To discover the address of a destination process, each observer uses a routing table, called *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 4-4 represents a typical *radictable*.

Table 4-4: An example of *radictable* for the cluster in Figure 4-3

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

4.4.2 State saving phase

In this phase, protectors and observers collaborate in order to save snapshots of the parallel application's state. This phase 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 4-5 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 paragraph 4.4.3.

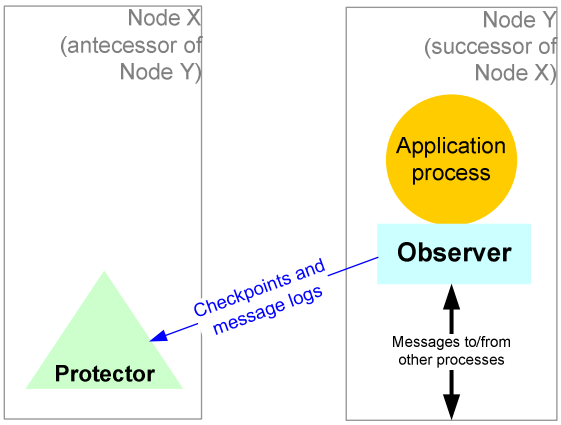


Figure 4-5: Relation between an observer and its protector.

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, similar to a RAID-1 scheme [Patterson, *et al.*, 1988]. Therefore, if a process fails, all information required to recover it is in a survivor node.

The difference between the legacy RAID-1 scheme and the redundant scheme implement in RADIC is that RADIC replicates processes' states and message logs instead of crude data chunks. Actually, RADIC takes care of saving the computation done by a process until a given time.

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 4-6 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.

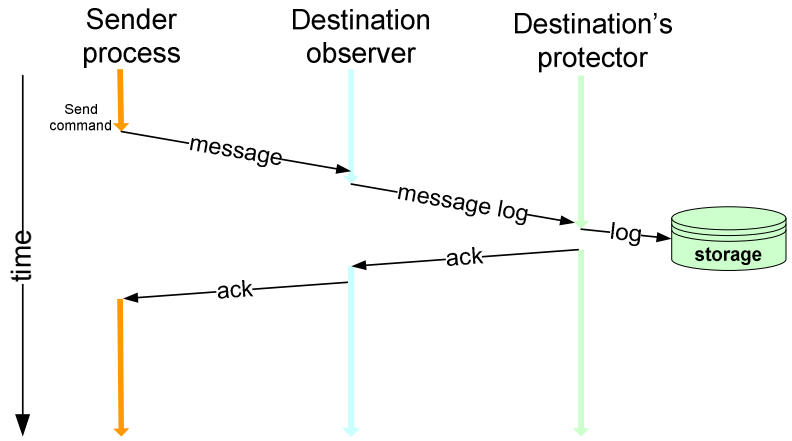


Figure 4-6: Message delivering and message log mechanism.

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.

4.4.3 Failure detection phase

The failure detection is an activity performed simultaneously by protectors and observers. Each one performs specific activities in this phase, 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 4-7 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 4-8 depicts three protectors and the heartbeat/watchdog mechanism between them.

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.

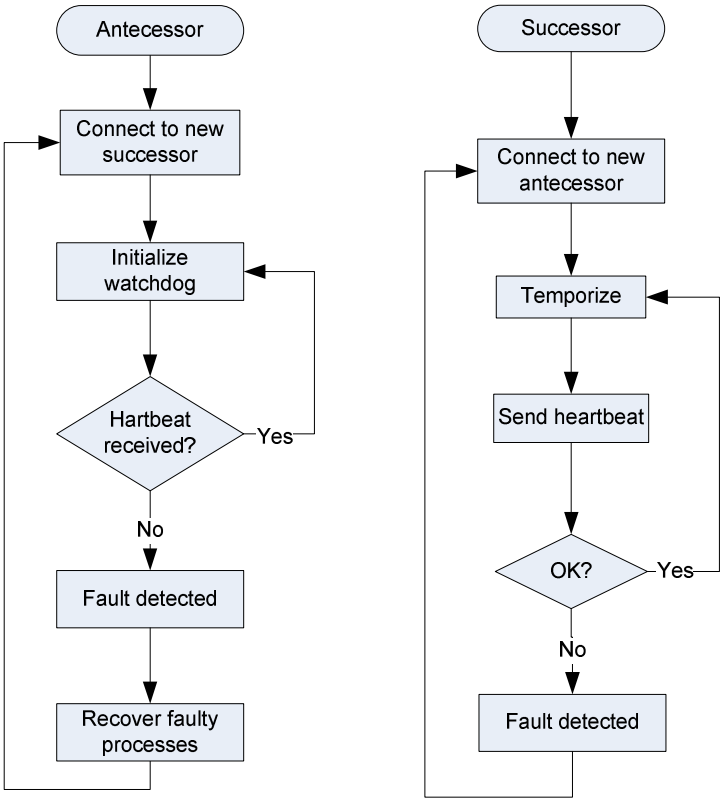


Figure 4-7: Protector algorithms for antecessor and successor tasks

How protectors deal with Byzantine problems

As explained in the previous topic, when a node fails two neighbor protectors of the faulty node detect such failure: the antecessor and the successor of the failed node. Depending on the network's topology of the parallel computer, the communication paths from a node to its antecessor may be distinct of the communication path from this node to its successor.

For example, in the configuration depicted in Figure 4-8, the protector T_X and T_Z will detect a failure in T_Y. If there is a communication problem between T_Y and T_Z but the communication between T_X and T_Y is fine, then T_Y and T_Z will have to agree about the failure (Figure 4-9a.) Such situation can be treated as a Byzantine problem [Lamport, *et al.*, 1982].

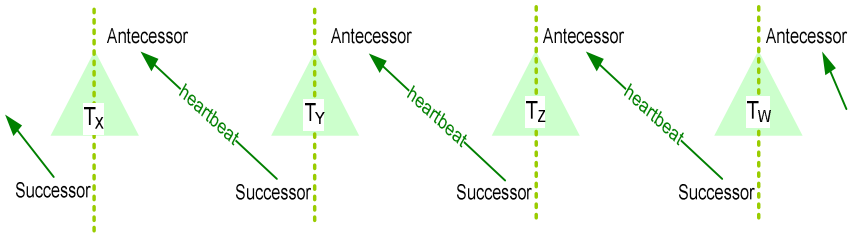


Figure 4-8: Four protectors (T_X, T_Y , T_Z and T_W) and their relationship to detect failures. Successors send heartbeats to antecessors.

To deal with such problem, when a protector detects a failure in its antecessor neighbor, it marks this antecessor as a suspect and contacts the suspect's antecessor in order to confirm the suspect's status. Then, according to the status received from the suspect's antecessor, the protector takes one the following actions:

a) If there is no answer, then the protector sends a suicide command to the local observers and then terminates, because it assumes that its node has a communication problem;

b) If the suspect's antecessor answers that the suspect is fine, then the protector assumes that it has a communication problem with its antecessor and does the same as described above in paragraph (a);

c) If the suspect's antecessor confirms that the suspect failed, then the protector fetches a new antecessor and sends a checkpoint command to the local observers;

Figure 4-9a shows the communication problem between T_Z and its antecessor T_Y, but each one keeps communicating with other neighbors. According to what we explained before, T_Z commits suicide because its antecessor is fine (Figure 4-9b.)

Concurrently, T_W fetches a new antecessor. Figure 4-10 represents the activities a protector performs to deal with the Byzantine problems.

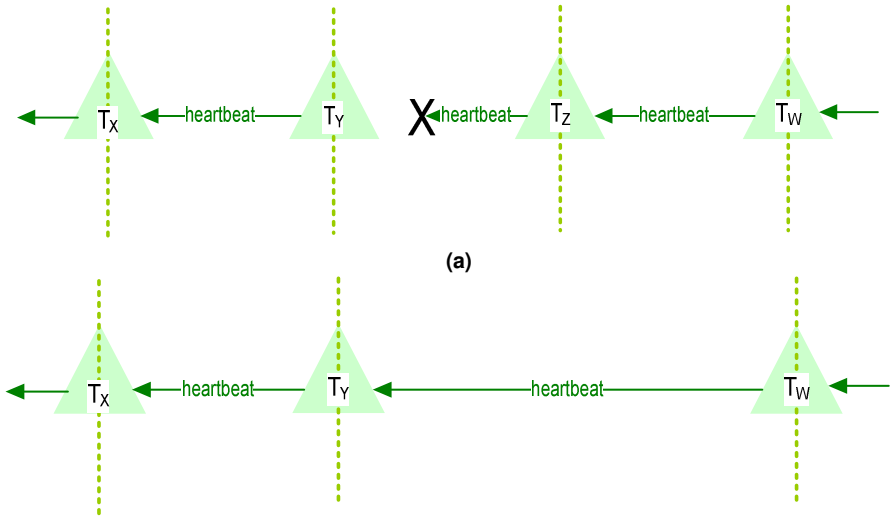


Figure 4-9: T_X is the antecessor of T_Y , T_Y is the antecessor of T_Z and so on. (a) Communication failure between T_Y and T_Z generates a Byzantine problem. (b) T_Z has committed suicide and T_W connects to T_Y.

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.

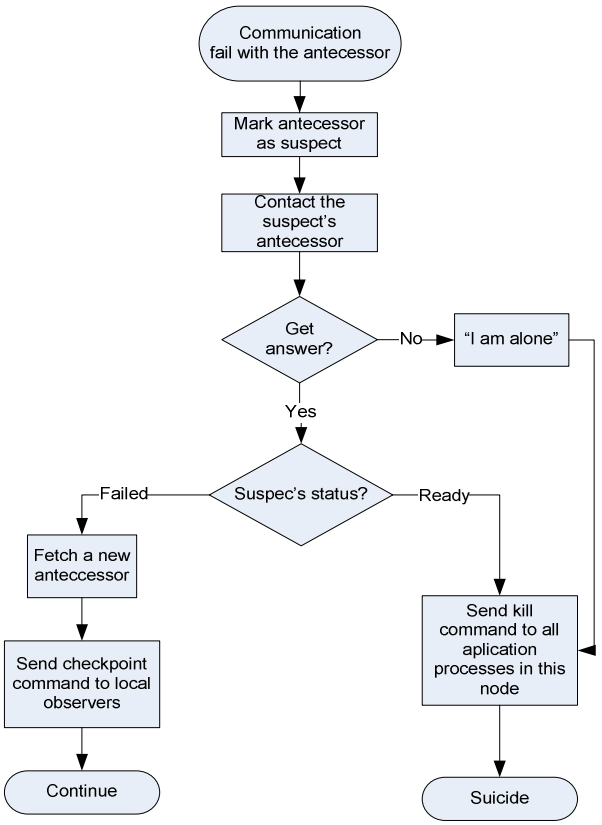


Figure 4-10: How a protector deals with Byzantine problems

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.

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 4-5.

Table 4-5: The *radictable* of each observer in the cluster in Figure 4-3.

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
. .	. .	. .

Analyzing Table 4-5, 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.

Practical issues about the heartbeat/watchdog cycle

The heartbeat/watchdog mechanism of RADIC transmits information throughout the communication channel and generates a communication traffic that affects the network bandwidth. The solution to reduce the traffic generated by the heartbeats is to enlarge the heartbeat cycle, therefore reducing the amount of information that watchdog/heartbeat mechanism transmits.

However, an enlargement in the heartbeat cycle reduces the sensibility of the failure detection mechanism; because the watchdog will take more time to detect a heartbeat missing. For example, if the heartbeat cycle is three seconds, in worst case the watchdog will take three seconds to warn about a problem. If we enlarge the cycle in order to reduce the network implication, let us say to nine seconds, the system will take three times more time to warn about a problem.

4.4.4 Recovery phase

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 4-6 explicates the atomic activities of each element.

When the recovery phase is finished, the RADIC controller's structure is reestablished and henceforth is ready to manage new failures. Figure 4-11 presents the configuration of a cluster from a normal situation until the recovery phase has finished.

Recovering failed application processes

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.

Table 4-6: 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

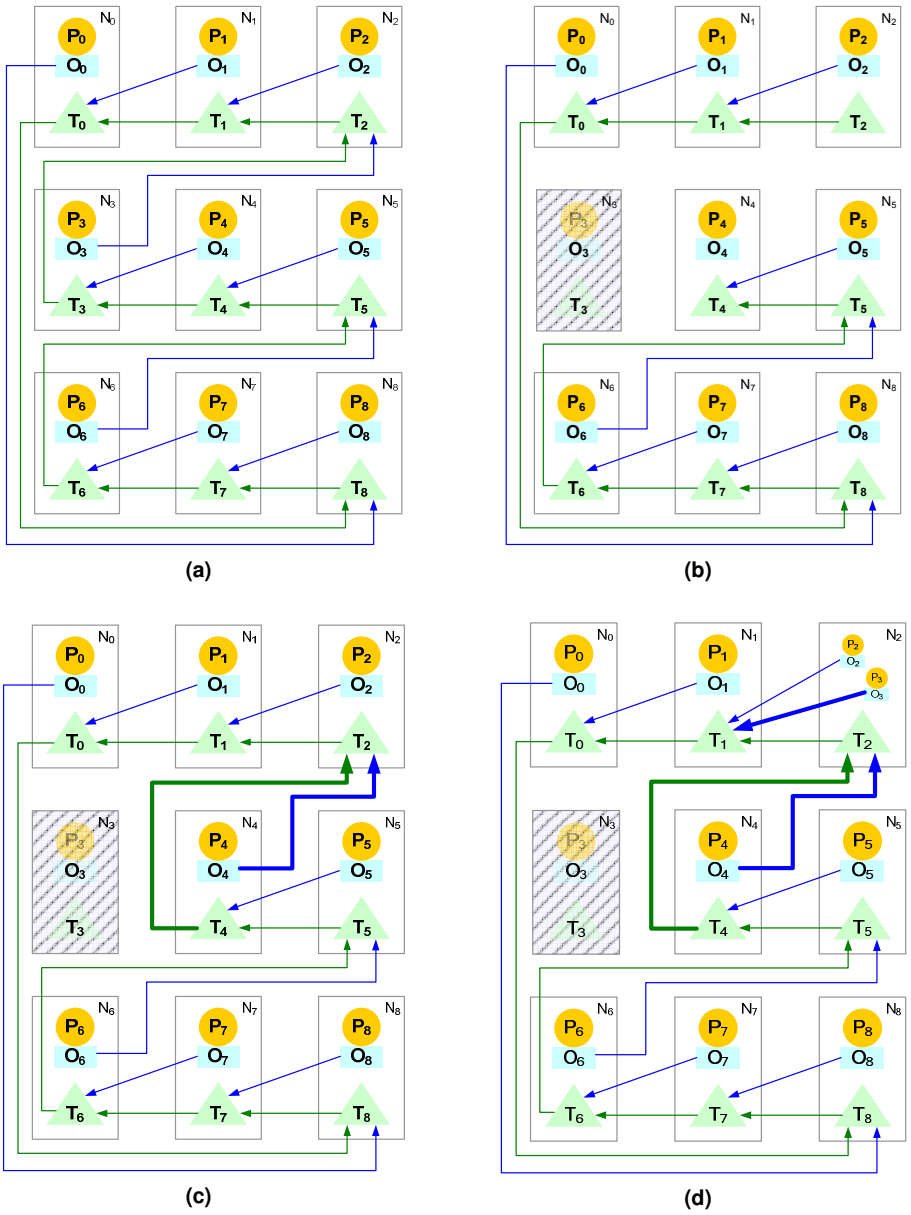


Figure 4-11: Recovering phases in a cluster. (a) Failure free cluster. (b) Fault in node N3. (c) Protectors T2 and T4 detect the failure and reestablish the chain, O4 connects to T2. (d) T2 recovers P3/O3 and O3 connects to T1.

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.

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.

4.4.5 Fault masking phase

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 paragraph 4.4.3, 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 4-7 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 4-7: The *radictable* of an observer in the cluster in Figure 4-3.

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 4-12 we represent four nodes of Figure 4-3 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.

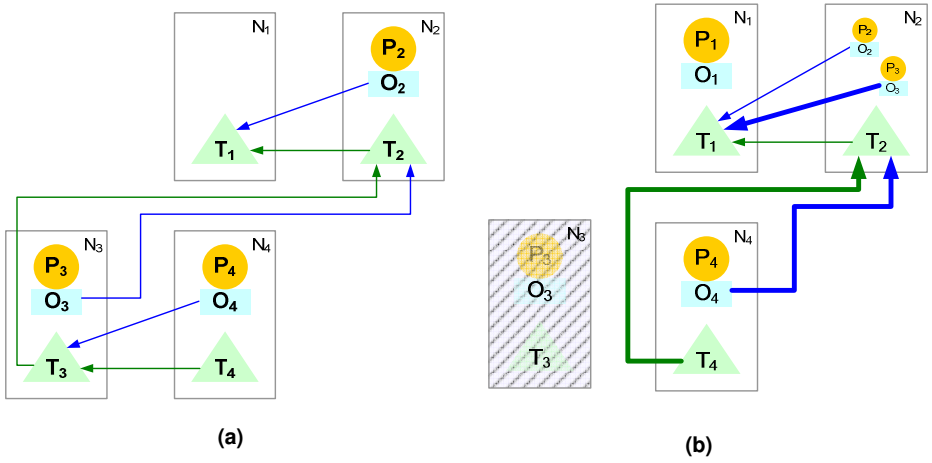


Figure 4-12: (a) A failure free cluster; (b) The same cluster after the management of a failure in node N_3.

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.

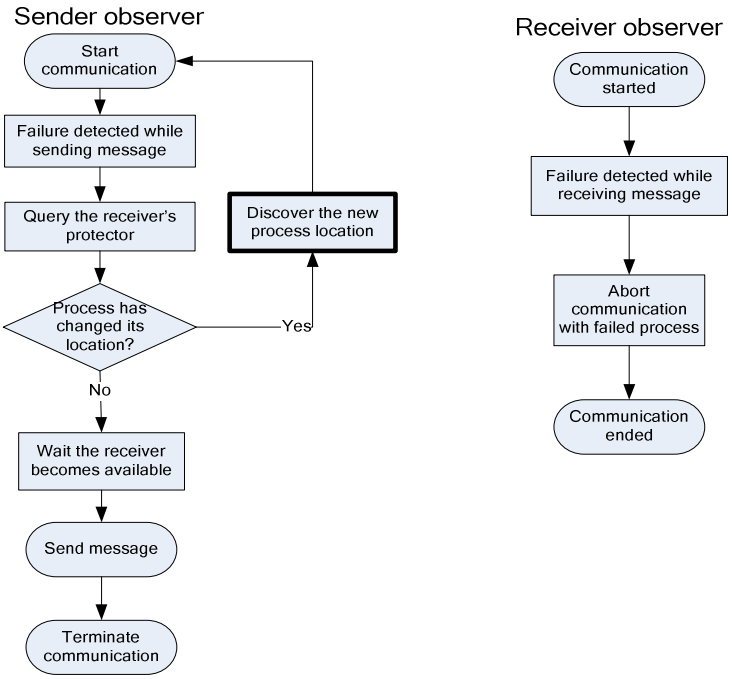


Figure 4-13: Fault detection algorithms for sender and receiver observers

Figure 4-13 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 4-12 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 4-8.

Table 4-8: Part of the original *radictable* for the processes represented in Figure 4-12a.

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.

Table 4-9: Part of he updated *radictable* of a process that has tried to communicate with P3 after it has recovered as shown in Figure 4-12b.

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

In our example, the updated *radictable* of a process who tries to communicate with the recovered process P3 has the information presented in Table 4-9. 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.

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.

4.5 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. The literature related to checkpoints contains several studies about how to estimate the optimal checkpoint interval in order to minimize the application run time in the absence of failures [Daly, 2006; Duda, 1983; Gelenbe, 1979]. Other works investigated the checkpoints under a practical point of view [Mandal and Mukhopadhyaya, 2003; Plank and Thomason, 2001; Ziv and Bruck, 1996].

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.

4.6 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.

4.6.1 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 paragraph 4.1.3 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 16, 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 16, 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.

4.6.2 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.

Two immediate advantages of the structural flexibility of RADIC are the possibility of using spare nodes and the clustering of protector's chain. In the first case, the parallel computer might have one or more spare nodes. Such nodes would assume the faulty processes in order to maintain the same number of computational

nodes. Therefore, the computational power remains the same the system had before the failure.

In the second 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 4-14 depicts an example of using two protectors' chains in our sample cluster.

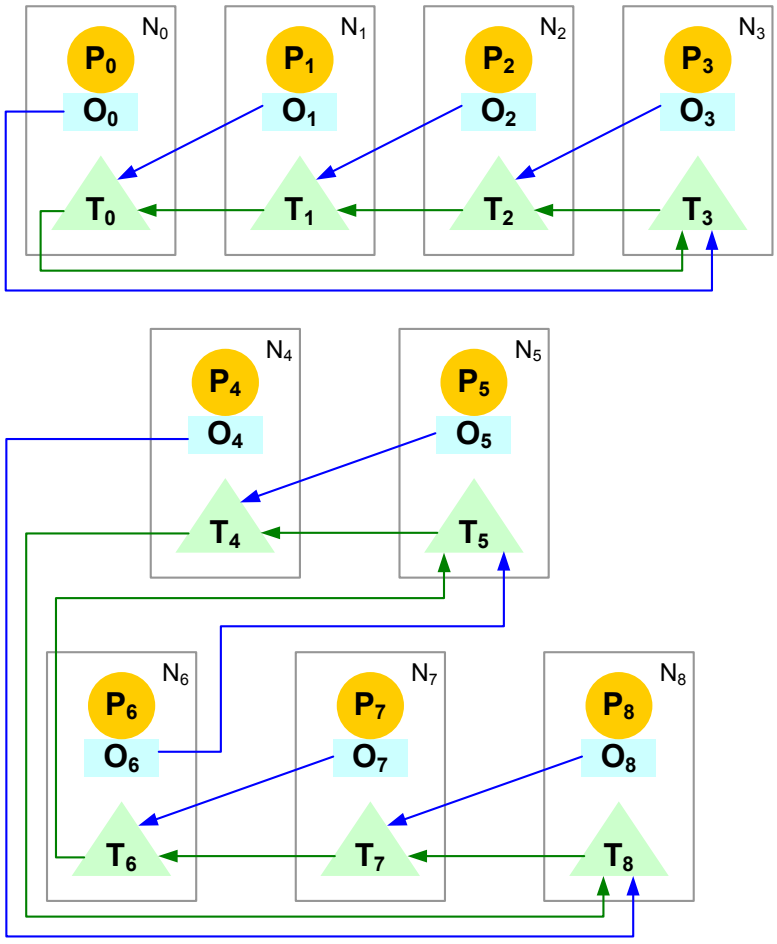


Figure 4-14: A cluster using two protectors' chain.

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 4-10 exemplifies the chain information in a typical *radictable*.

Table 4-10: The *radictable* of an observer for a cluster protected by two protectors' chains like in Figure 4-14.

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

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 4.3.1.) Therefore, at least three nodes must compose a protector's chain. We depicted such minimal structure in Figure 4-15, in which each protector has an antecessor (to which it sends the heartbeats) and a successor (from which it receives heartbeats.)

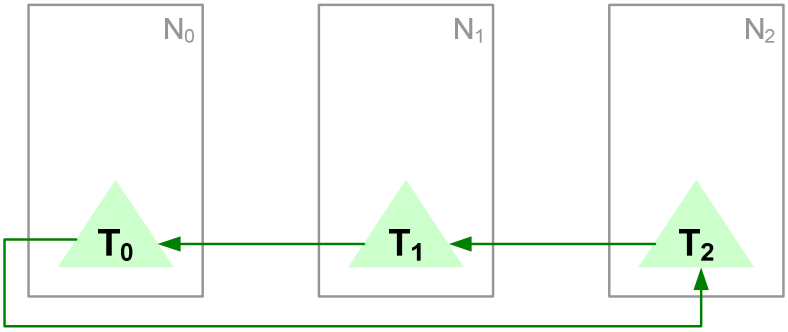


Figure 4-15: The minimum structure for a protectors' chain.

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 3 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.)

$$MaxFaults = Number\_of\_Protectors - 3 \qquad 3$$

In this chapter, we have defined the fundamentals and explained the operation of the RADIC fault tolerance architecture. In the next chapter, we are going to present a practical implementation of the RADIC architecture, which we used to perform functional and experimental tests.

Chapter 5

An implementation of the RADIC Architecture

Now that we have presented the functional aspects of the RADIC architecture, it is necessary to describe the method used to validate the concepts of this architecture.

To assure the correct operation of a fault-tolerant architecture for message-passing systems in the absence of failures is a relatively easy task, because the tests are limited to compare the results generated by parallel applications using the fault-tolerant architecture against the same parallel applications using a legacy message-passing implementation.

On the other hand, to evaluate the correct operation of a fault-tolerant architecture in the presence of failures is a cumbersome task. Many fault scenarios are possible during a program execution and the exhaustive test of all these scenarios requires a large amount of work. *Fault Injection* is the best strategy to face such challenge.

Fault injection is a strategy to inject faults in a controlled way. Once all fault scenarios are established, a fault injection mechanism, which can create such scenarios, executes inside the system together with the parallel application. With this mechanism, it is possible to create a specific fault scenario and monitor the behavior of the fault-tolerant architecture in such scenario.

The recent literature about testing fault tolerance architectures presents some works devoted to test fault-tolerant systems. The FAIL-FCI framework [Hoarau and Tixeuil, 2005] is an example of framework for test distributed systems. FAIL is a fault injection language that rests on FAIL-FCI in order to evaluate message-passing implementations [Hoarau, *et al.*, 2006]. Other works dedicate to testing of fault

tolerance schemes in Grids, like the work published by Sébastien Tixeuil in [Tixeuil, *et al.*, 2006].

The validation of the RADIC functional requirements depended of a message-passing implementation that could offer a suitable platform to the supporting of the fault tolerance mechanisms of RADIC.

The first issue was to define a messaging-passing paradigm. The MPI standard was the natural choice for our message-passing implementation, because it has become a widely adopted standard in the modern message-passing applications.

At the time we started our work, all available MPI implementations had a fail-stop behavior; therefore, we decided to create a new message-passing implementation that could support a node failure without stopping.

The creation of our own MPI implementation also allowed a better the control of the messaging-passing mechanism. This has facilitated the implementation of the message-log and the recovery mechanisms. Furthermore, we saved many time in software development, because we avoided the typical effort required in the understanding and modification of softwares made by other developers.

With these issues in mind, we developed a MPI implementation to evaluate the concepts of RADIC. This implementation, called RADICMPI, has served for two purposes: evaluate the functionality of RADIC as a fault-tolerant architecture , and assess the influence of the RADIC fault tolerance mechanism in practice.

In this chapter, we present the functional tests, i.e., the tests performed in order to evaluate the functional requirements of RADIC. First, we explain RADICMPI and the implementation of the RADIC distributed controller. Next, we define the operation of the fault injection mechanism and our test methodology. We conclude analyzing the results of our tests.

5.1 RADICMPI

RADICMPI is a MPI implementation created to test the functional and experimental evaluation of the concepts of the RADIC Architecture.

For the development of RADICMPI, we decided to rest only on open source softwares. Such decision has satisfied two requirements: a) it gave us freedom to adapt any software feature and b) it maximized the use of software tools accepted in the research world.

Because of this, our current RADICMPI implementation rests on the Linux operating system running on Intel IA-32 architectures. Our checkpoint mechanism rests on the popular BLCR, the Berkeley Labs Checkpoint/Restart library [Hargrove and Duell, 2006]. The message-passing mechanism and all messages exchanged between the RADIC elements use legacy TCP/IP sockets.

The operation of RADICMPI should attend to the fundamental features offered by RADIC: transparency, flexibility, scalability and decentralization. Therefore, these four features drove the design of RADICMPI.

Figure 5-1 depicts the several software levels of a parallel application using RADICMPI. This figure shows the transparent behavior of RADICMPI, because each application process relates only with a classical MPI API and has no contact with the RADIC API. This means that the application does not takes care of any fault tolerance activity.

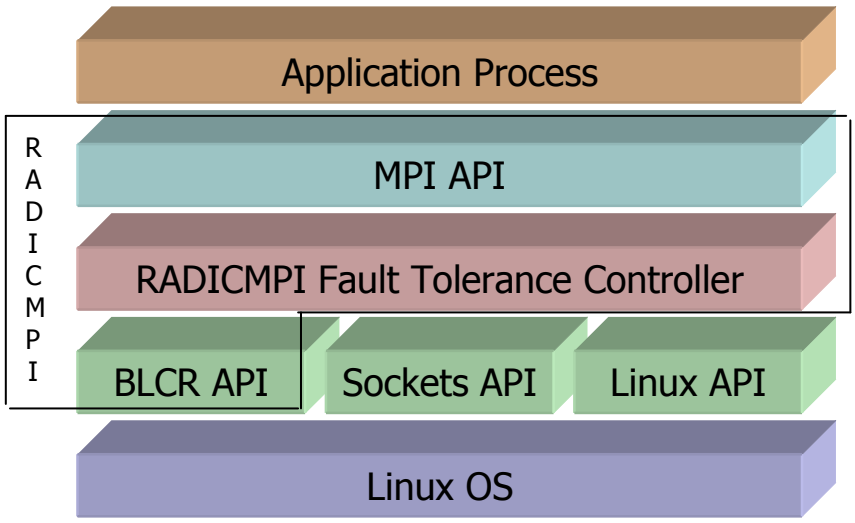


Figure 5-1: Software levels or the RADICMPI

In Table 5-1, we summarized all RADIC fundamental features and how RADICMPI implement them.

In order to implement the transparency, we have created a software environment in which a programmer would only need to re-compile their application codes in order to use RADICMPI. The software environment has two elements: a software library (RADICMPI library) and an execution environment (RADICMPI scripts).

Besides a transparent operation, RADICMPI also must comply with the other three features of RADIC. The operation of RADICMPI does not require any central or dedicated node. All RADIC functional elements, protectors and observers, which implement the RADIC controller for fault tolerance, execute in the same nodes where the processes of the parallel application are. This behavior complies with the decentralization requirement.

The number of nodes in the cluster does not modify the functionality of the RADIC controller as well as the messaging passing mechanism. Therefore, only the behavior of the application determines the scalability of the system. The number of cluster's nodes does not compromise the scalability of RADICMPI.

Table 5-1: How RADICMPI satisfies the RADIC features

Feature	How it is implemented by RADICMPI
Transparency	– A software library implements the fault tolerance and message-passing mechanisms – The programmer only has to re-compile the application code
Decentralization	– The protectors and observers execute in the same nodes used by the application processes – No central or special node required
Scalability	– The operation is not affected by the number of nodes in the parallel computer
Flexibility	– Fault tolerance parameters are defined by the user when it launches the parallel application – It is possible to implement other checkpoint policies

It is possible to modify checkpoint interval, the watchdog interval and the structure of the protection chain to comply with the user's requirements. Furthermore, the current RADICMPI implementation uses a typical time-driven checkpoint policy, but it is possible to implement other checkpoint policies, like a policy driven by the message-log size or by process' synchronization.

5.2 RADICMPI library

RADICMPI is a multithread library. Three threads will exist when we execute a program compiled with the RADICMPI library: the program main thread, the observer thread and the checkpoint thread.

Figure 5-2 depicts the threads of RADICMPI and the relationship between them. The main thread executes the user program and sends communication commands to the observer thread, which implements the fault tolerance and the message-passing mechanism. The observer thread sends checkpoint commands to the BLCR thread, which implements the checkpoint mechanism. Because RADICMPI works transparently to the application program, no relation is required between the user program (main thread) and the checkpoint mechanism (BLCR thread),

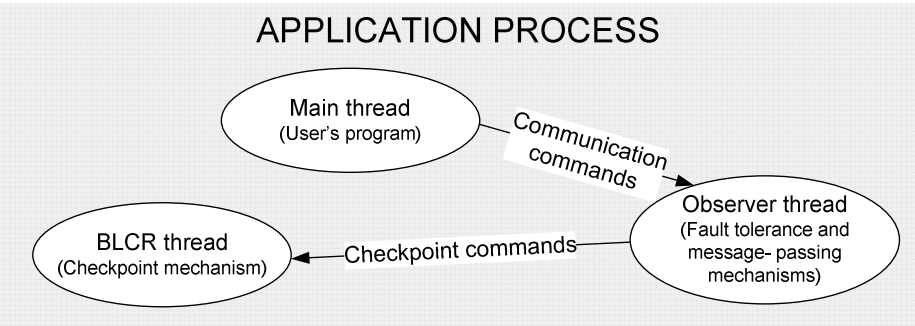


Figure 5-2: Threads of RADICMPI and their relationship

5.2.1 MPI API

RADICMPI offers a subset of routines and constants defined by the MPI-1 standard. The syntax of all RADICMPI functions strictly follows the MPI-1 specification in order to comply with the transparency requirement.

The MPI API level completely isolates the application process from all message delivery activities. RADICMPI contains 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.

MPI_Init

The MPI_Init creates the observer thread and the BLCR thread. This function also manages the arguments received by the application program in two sets of arguments: the application's arguments and the observer's arguments.

MPI_Finalize

The MPI_Finalize routine simply sends a finalize command to the observer. When the observer receives this command, it informs to the local and remote protector that it is going to finish and then it terminates.

MPI_Send

MPI_Send drive the observer to send the message stored in a buffer informed by the application process. The process blocks until the observer delivers the message to the communication channel.

MPI_Recv

MPI_Recv requests a message to the observer. The observer puts the message in the buffer allocated by the process. The function blocks until it receives the message from the observer.

MPI_Sendrecv

This function is just a sequential calling of the MPI_Send function followed by the MPI_Recv function.

Auxiliary MPI routines

MPI_Comm_rank, MPI_Comm_size, MPI_Wtime, MPI_Type_size and MPI_Get_processor_name execute locally, i.e., they do not require any interaction between the local process and other processes.

MPI_Comm_rank, MPI_Comm_size get their data directly from the observer. The observer receives this information from the RADICMPI environment, at the time the application starts.

MPI_Wtime, MPI_Get_processor_name and MPI_Type_size get their information from the operating system.

5.3 RADICMPI Controller for fault tolerance – Observer

The RADIC library contains half of the RADIC distributed controller elements: the observers. Each observer works as a thread of the application process. This thread manages the message-passing between the parallel application processes and implements the fault tolerance functionalities.

Figure 5-3 depicts the external and internal sources of the events that drive the observer operation, the elements in the top of the figure are the ones that generate external events and the elements in the figure bottom are the ones that generate internal events.

External events correspond to MPI commands sent by the application process; messages, which comes from the remote application processes of the parallel application; and initialization and recovery commands. Internal events are consequence of the operation of the state saving phase of the fault tolerance mechanism (checkpoints and message logs) or of the fault detection mechanism.

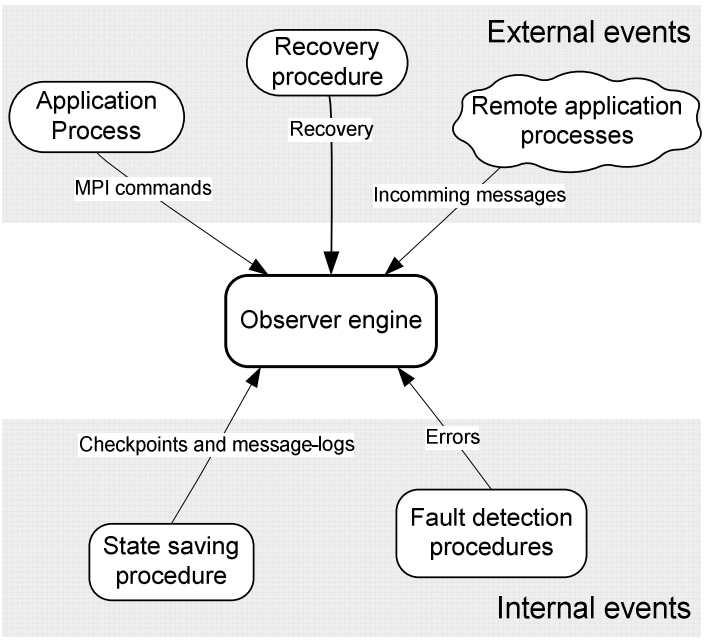


Figure 5-3: The observer engine and the external and internal event generators in RADICMPI

Incoming-messages (Remote application processes events)

The RADICMPI message-passing mechanism rests on TCP/IP sockets. Each observer sets up a TCP server socket for accepting connections from other observers. 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.

After an observer accepts a connection from other observer, it first waits for a header that identifies the message that is about to come. The header contains the message logical clock (which identifies the message sequence), the sender and receiver identification, the MPI tag of the message and the message's size.

The receiver than compares the logical clock of the received message against the logical clock that counts the messages received from the sender in this *radictable*. If the received logical clock is greater than the current counter, it means that the message is new. Then, it returns an ACK message to the sender and prepares to

receive the message. Otherwise, it returns an NAK message indicating that the message is old (the sender has recovered and replayed an old message).

If the message is new, the observer receives it through the TCP socket, puts the message in a temporary buffer, logs the message in its protector, returns a new ACK message to the sender and closes the TCP socket. Then it increments the logical clock, the counts the messages received from the sender, in its *radictable*. Figure 5-4 represents all sequence of steps followed by the observer in response to an incoming message event.

In this section, for the sake of simplicity, we did not explain how the fault detection procedure interacts with the message-incoming procedure. We will clarify this interaction in the section that explains the fault detection procedure.

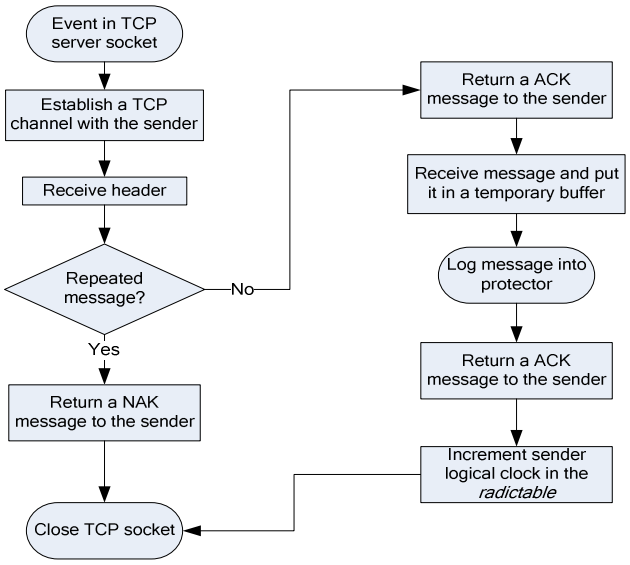


Figure 5-4: How the observer engine deals with incoming messages (error situations are not represented in this diagram)

MPI_command events

The application process generates MPI_commands to the observer. The observer deals with each MPI_command as an individual event. For the application process,

the messaging-passing mechanism and the fault tolerance mechanism are completely transparent.

– MPI_Init command

The MPI_Init function starts the observer thread. The first task of every observer is to create a TCP server socket in order to accept connections of all other application processes.

Next, the observer communicates with the local protector (the protector that is running in its node) in order to ask the information about the location of its remote protector. The remote protector is the one in which the observer stores checkpoints and message-logs of its application process, namely the neighbor antecessor of the node where the observer is running. After discover its remote protector, the observer communicates with it in order to inform that it is ready.

Next, the observer starts the procedure to mount its *radictable*, which contains all information about the location of the other application processes and their respective observers. In the current RADICMPI implementation, the leader observer, i.e., the observer with rank equal to zero, is in charge of requesting the location information and the protector information from all other observers. The leader observer then builds a complete *radictable* and broadcasts it to all other application processes. At the end of the initialization phase, all observers have the same *radictable*.

Is also in the initialization procedure that the observer starts the checkpoint thread using the BLCR API. In Figure 5-5, we represent the steps followed by the observer thread after it starts through an MPI_Init command.

– MPI_Recv command

The observer puts all incoming-messages in a temporary buffer. When the application process executes an MPI_Recv command, the observer uses the sender information and the MPI tag of the message to determine if the message is already in its buffer or not.

If the requested message is not in the temporary message buffer (because such message has not arrived yet), then the observer blocks the application process until

the message arrives. Otherwise, it immediately delivers the message to the application process. Figure 5-6 depicts the steps followed by an observer after it receives an MPI_Recv command.

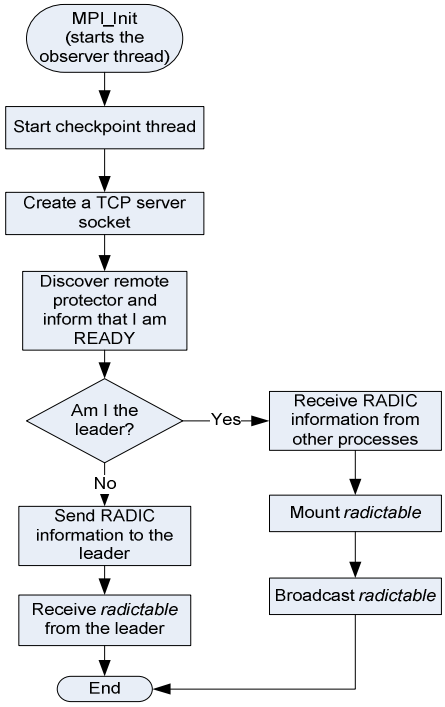


Figure 5-5: How the observer engine responds to an MPI_Init command

– MPI_Send command

When a process executes an MPI_Send its observer looks for the destination address in its *radictable*.

The sender observer contacts the destination and starts the transmission procedure. First, the sender transmits a header in order to identify the message for the destination. The header contains the message logical clock (which identifies the message sequence), the sender and receiver identification, the MPI tag of the message and the message's size. Then it waits for the confirmation from the destination.

Two kinds of messages may return from the destination: an ACK message or a NAK message. If a NAK message returns, the sender knows that the receiver already

has received that message. Then, the sender observer increments the send logical counter related to the respective receiver in its *radictable* and terminates the MPI_Send. This avoid that a recovered observer retransmit old messages through the communication channel.

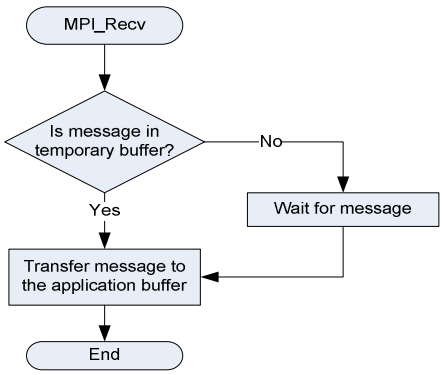


Figure 5-6: How the observer engine responds to an MPI_Recv command

If the destination returns an ACK message, then the sender knows that it may send the message. Then, it transmits the message and waits for a new ACK message from the destination. When the ACK comes, the sender observer increments the logical counter for send messages in the correspondent *radictable* line related to the destination.

Figure 5-7 represents the steps followed because and MPI_Send command. In this figure, for the sake of simplicity, we do not represent the error conditions. We explain such conditions in the fault detection procedure.

– Auxiliary MPI commands

The management of MPI commands that do not involve communication is very simple. Some functions require the use of services from the operating system in order to complete the command.

The MPI_Wtime and MPI_Get_processor_name functions are the ones that depend of the operating system to get the information required to complete the command. The observer calls the respective function of the operating system and manipulates the information in order to respond to the command. For example, for the

MPI_Wtime command, the observer accesses the wall clock of the operating system and adapts the information to the format required by the function.

The MPI_Comm_size and MPI_Comm_rank use information of the RADICMPI environment, namely the number of process in the communicator and the id of the application process.

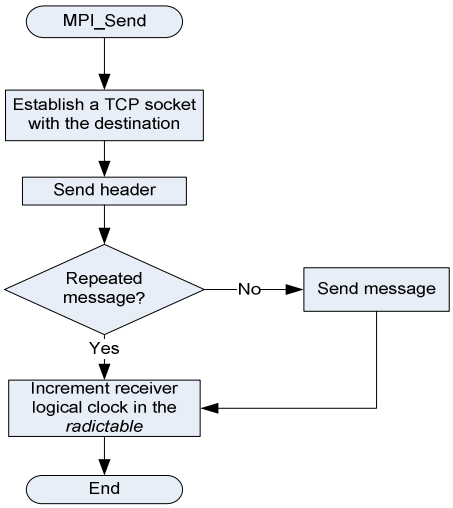


Figure 5-7: How the observer engine responds to an MPI_Send command

State saving events

The states saving events are events created by the fault tolerance mechanism. They correspond to message-log events and checkpoint events.

– Message logs

One of the steps performed by an observer when it is dealing with incoming-messages is to log new received messages in its remote protector, before sending an ACK message to the sender in order to confirm the message reception, as shown in Figure 5-4. Therefore, a message log event is an internal events generated when the observer receives a new message.

The communication between the observer and its protector is simpler than to the communication between two observers. The observer first establish a communication

socket with the protector, sends a message header to identify the message, sends the message, and then waits for an ACK message from the protector, which informs that the message was correctly logged. After the acknowledge message comes, the observer closes the TCP socket and terminate the log procedure. We depict the steps of the message-log procedure in Figure 5-8 without consider the error situations. Such situations are part of the fault detection procedure.

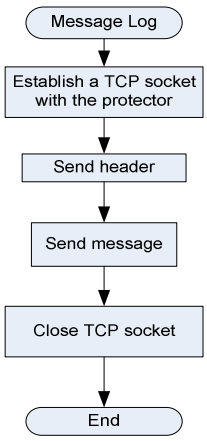


Figure 5-8: Steps of the message log procedure

– Checkpoint events

Checkpoint events are internal events started by time counter in the observer. The interval between two checkpoints is function of the checkpoint interval defined by the user in the system startup. The observer initializes its counter initializes when it starts, when it recovers and after a checkpoint completes.

The checkpoint mechanism is function of the BLCR checkpoint library. Because this library does not perform checkpoint of opened TCP sockets, a lock mechanism blocks the checkpoint procedure if the observer is communicating with other process. Similarly, the observer never initiates nor accepts communications while a checkpoint is occurring.

Because the processes in the RADIC architecture do not coordinate their checkpoints, a fake error condition could occur if a process tries to communicate with

a destination that is making a checkpoint. The strategy to deal with this fake error condition is in the fault detection procedure.

In the first step of the checkpoint procedure, the observer opens a TCP socket with its protector and informs that it is going to checkpoint. Then, the observer the observer close the TCP server socket that accepts communication of other observers, to avoid new connections from this point, and checks if there are active communications. If some communication is active, the observer waits until it concludes. This checking is necessary to assure that there are no open TCP channels with other observers when the checkpoint mechanism starts.

Following the checkpoint procedures steps, the observer orders that the BLCR library take a checkpoint of the whole application process state, including the observer itself. The BLCR checkpoint functions take control of the system, execute the checkpoint and transmit the checkpoint to the protector via the active TCP socket.

When the BLCR library concludes the checkpoint, it returns the control to the observer. Then, the observer closes the TCP socket for transmission. This activity signalizes to the protector that the checkpoint has finished. The protector confirms the correct checkpoint reception closing the socket with the observer.

Finally, the protector reopens the TCP server socket for accept new connections from other observers and finalizes the checkpoint procedure. Figure 5-9 represents the steps of the checkpoint procedure without error conditions.

The recovery event

According to the operation of the RADIC architecture, the recovery procedure corresponds to restarting a process since its previous checkpoint. In RADICMPI, when the BLCR checkpoint function takes a checkpoint, the checkpoint contains the states of the observer and of its application process, because their threads share the same memory space.

The start of recovery mechanism is not an observer attribution, because are the protectors who have to detect the failed nodes and restart faulty processes. The

recovery mechanism of RADICMPI rests on the restarting procedure of the BLCR library.

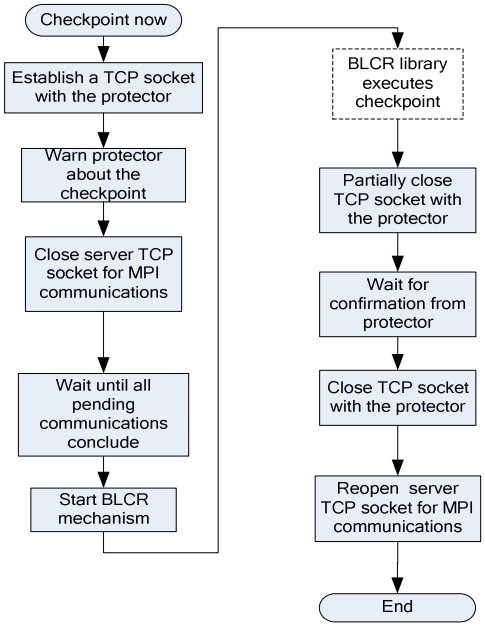


Figure 5-9: Observer checkpoint procedure without error conditions

When BLCR restarts a process since a checkpoint, it signalizes to the process that it is restarting. An observer relies on such signal in order to identify that it is restarting since a checkpoint, i.e., it is recovering from a fault.

Because the process is recovering in a node different from the one in which it was before the fault, the first observer concern is to update the information about its remote protector. For this, the observer communicates with the local protector in order to discover who its antecessor neighbor is.

The observer then informs to its new remote protector that it is recovering and to the local protector that from now on it will use another protector. It is important to warn to the local protector that it is not in charge of the recovered observer anymore, because the observer is now running in the same node. This breaks the relation between a recovered process and its previous protector.

The next step is to deal with the message log of its process. The observer copies all messages from the message log to temporary buffer used to store incoming messages. Therefore, when the application process requests those messages again, they already are in the reception buffer.

The observer then gets the messages from the message log, which is present in the node it has recovered, and copies these messages to the buffer used to store incoming messages. Therefore, when the recovered process requests an old message (MPI_Recv), such message is already in the observer's buffer. If the recovered application process tries to retransmit messages, the destination observer discards such as we have previously explained in the procedure that deals with incoming messages.

Finally, the observer reopens the TCP server socket in order to communicate with other application processes and informs to its remote protector that it is again ready to communicate. Figure 5-10 represents the recovery procedure. Again, we do not represent error events because they are responsibility of the fault detection mechanism.

5.3.1 How the RADICMPI observers detect and manage faults

As we depicted in Figure 5-3, the fault detection procedures generate internal events to the observer engine. In RADICMPI, the fault detection procedures are present inside all other procedures, i.e., when the observer engine treats an event, the procedure who deals with the event includes the functions to detect and to deal with the error events. We may define the RADICMPI fault detection procedure as a collection of all the individual fault detection procedures inside the observer engine.

The fault detection mechanism bases on communication failures and communication timeouts in order to detect a fault. Basing on such assumption, we may define two classes of communication failures that exist in the observer context: failures between two observers and failures between observers and protectors.

Because the fault detection phase of the RADIC architecture always starts the fault-masking phase, in RADICMPI we coupled the fault masking functions together

with the fault detection functions. Therefore, whenever some procedure detects a failure, the fault masking procedure immediately starts to adapt the observer to the new cluster configuration.

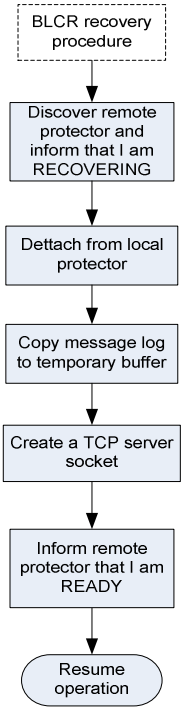


Figure 5-10: How observer recovers

Failures detected when two observers are communicating

These failures can occur in two different situations: the observer detects a problem before it starts a communication (valid only for the sender observer) or the observer detects a problem during a communication (valid for both observers implicated in the communication).

– The sender observer detects a failure before it initiates the communication

This situation occurs when the destination does not accept the sender because one of the following reasons: a) the destination is checkpointing and its TCP server socket is closed; or b) the destination node has failed and the destination process is running in another node.

In both cases, the observer first contacts the protector of the destination process in order to discover what is happening. If the destination's protector answer that the process is checkpointing, the sender observer keeps querying the protector until it answer the nodes is ready. Then it tries to communicate with the destination again.

If the protector of the destination process responds that it is not responsible by that process, the sender observer admits that the destination process has failed and, using its *radictable* and the algorithm explained in paragraph 4.4.5, calculates the new address of the destination process and retries the communication.

- An observer detects a failure after the communication has been started

In this case, the behavior of the observer depends of its role in the communication. If it is receiving the message, it simply aborts the communication because it knows that the sender will restart the communication again. If it is sending the message, it assumes that the destination has failed and uses its *radictable*, together with the algorithm explained in paragraph 4.4.5, to calculate the new address of the destination process in order to reestablish the communication.

Failures detected when an observer is communicating with a protector

As for the communication between two observers, the fault detection between a protector and an observer also relies on communication failures and communication timeouts. In this paragraph, we evaluate the mechanism since the observer view.

The communication between an observer and a protector occurs in two different phases: in the state saving phase and in the fault-masking phase. Therefore, the operation of fault detection mechanism depends of the current operation that the observer is executing.

- An observer detects a failure with its remote protector in the state saving phase

In this case, the observer immediately communicates with its local protector in order to determine who its new protector is. Then it immediately takes a checkpoint, in order to reestablish the protection, and concludes the communication.

In RADICMPI, the observer also detects a failure in its remote protector when the local protector sends a special warning message to it. This mechanism protects the observer of being unaware that its protector has failed, because it has stayed a long period without needing to communicate with the protector, for example inside a checkpoint interval in which no message log occurred.

– An observer detects a failure with a protector of a destination process

This situation occurs when a sender observer detects a communication error with a destination process, and needs to contact the destination protector to obtain information about the destination status.

In RADICMPI, the operation fail detection and fault masking functions are the same as with the observer detects a failure in a destination process. The observer uses the *radictable* and the search algorithm explained in paragraph 4.4.5 to calculate the location of the protector it needs to communicate.

When it finds the required protector, it updates the protector information in its *radictable* and reestablishes the communication.

5.4 RADICMPI Controller for fault tolerance - Protector

The protectors are the other half of the RADIC controller for fault tolerance. They operate like a distributed stable storage and as a distributed fault tolerance detector.

In the current implementation of RADICMPI, the protector has three main threads: the successor thread, the antecessor thread and the thread that deals with the observers.

The protectors execute as separated programs, they do not share the memory space of the application processes. Figure 5-11 depicts the structure of the protectuion program. The antecessor and the successor threads exist while the observers that relates to the protector are running. In the figure, this dependency corresponds to the

two arrows that connect the successor thread and the antecessor threads with the finalization procedure.

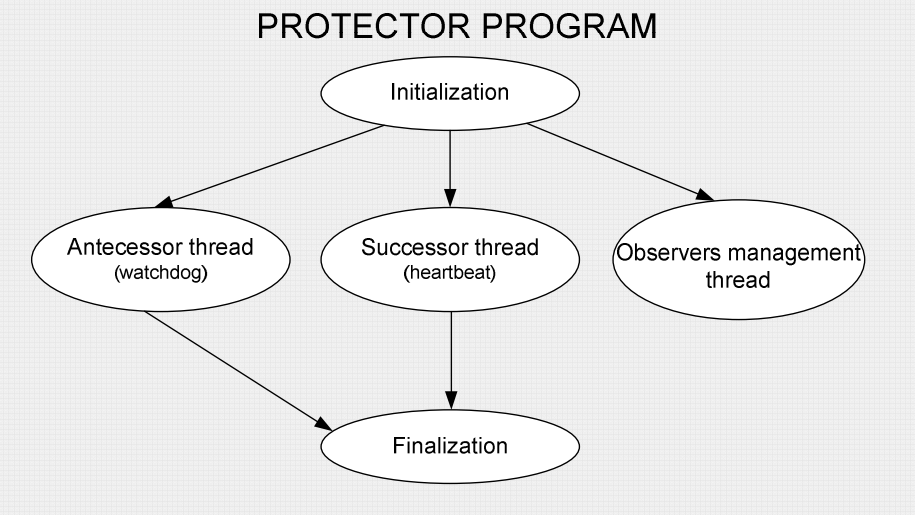


Figure 5-11: The protector program in RADICMPI

5.4.1 Initialization

The initialization procedure starts the all the protectors threads. Each protector receives the address of its neighbor antecessor and the address of its pre-antecessor (the antecessor of its antecessor). The user must design the protector chain and inform to each protector who its antecessor is and who its pre-antecessor is.

Another parameter that user must define is the watchdog/heartbeat interval, that will be used by the successor and antecessor threads in order to execute the fault detection mechanism.

5.4.2 Observers management thread

This thread creates a TCP server socket exclusively to accept the observers that will send checkpoints and message-logs for the protector or that will query about the status of other processes. The protector maintains a list with the status information of all observers that are using it as stable storage. The protector updates the list

according to the activities of the observers that it is protecting. Using this list, the protector answers to the queries about the status of each observer it is protecting.

All checkpoints and message logs of each observer that is using the protector as a stable storage are stored in the disk of the protector's node. The pessimistic log protocol used by the RADIC Architecture greatly simplifies the garbage collection procedure. Whenever a protector receives a new checkpoint, it automatically discards any earlier checkpoint and message log of the application process. This occurs because in the pessimistic protocol every checkpoint is a snapshot of the current state of the application process and it is not necessary any other information besides the current checkpoint to recover the process to this current state.

The observer management thread also accept connection from local observers, i.e., from observers that are running in the same node of the protectors. The thread maintains a list of these observers while they are active, and uses this list to warn the local observers about a failure in the antecessor protector.

5.4.3 Antecessor thread

The antecessor thread waits for the connection of a successor protector. This thread creates a TCP server socket to accept the connection of the successor protector and to establish the watchdog mechanism. The TCP socket between an antecessor and its successor is always active. It only closes in the finalization procedure.

Once the successor has connected, the thread establishes a watchdog mechanism based on the heartbeat cycle. The heartbeats sent by the successor reset the watchdog. For each heartbeat signal received the antecessor thread returns an acknowledge message (ACK message) to the successor protector.

If a heartbeat fails, the antecessor thread starts the fail confirmation function. The confirmation mechanism consists in wait an additional watchdog cycle to assure that the heartbeat is missing.

If the heartbeat does not arrive, the antecessor thread starts another confirmation mechanism, the one that checks for Byzantine failures. The algorithm in RADICMPI

is the one explained in the Failure Detection Phase of the RADIC Architecture, which we depicted in Figure 4-10.

When the antecessor thread finally confirms a failure, it takes care of recovering the faulty process from the faulty node. The recovery procedure is a parte of the BLCR library API. The protector simply calls the BLCR recovery function passing the name of the checkpoint file of the process that the function has to recover.

Concurrently with the recovery mechanism, the antecessor thread waits for a new successor connects and then restarts the watchdog function.

5.4.4 Successor thread

When the successor thread starts, it immediately fetches this antecessor in order to start the heartbeat mechanism. It establishes a TCP socket with the antecessor and keeps this socket opened until it the thread completes.

The successor threads regularly send heartbeat signals to the antecessor protector via the opened TCP socket in order to reset the antecessor watchdog. For each heartbeat sent the thread waits for an acknowledge message (ACK message). If the ACK message does not come, or if some communication error occurs the successor thread starts the fault confirmation function.

The fault confirmation mechanism consists in retrying to send the heartbeat to the antecessor protector. If it fails again, successor thread starts another confirmation mechanism, the one that checks for Byzantine failures. The algorithm in RADICMPI is the one explained in the Failure Detection Phase of the RADIC Architecture, which we depicted in Figure 4-10.

When the successor thread finally confirms a failure, it warns all local observers that the antecessor protector has failed. Then it connects to the pre-antecessor of the failed antecessor and reestablishes the heartbeat mechanism. The successor thread does not take part of the recovery procedure.

5.4.5 Finalization procedure

The finalization procedure initiates when all observers that relate with a protector terminate, i.e., the protector only can terminates after all local observers in this node terminate and all observers in the successor node terminate.

When the successor thread is ready to terminate, it sends a STOP signal to the antecessor protector using the heartbeat TCP socket and then terminates. Similarly, the antecessor thread only terminates after it receives the STOP signal in the watchdog TCP socket.

```
angelo@aoclp9:~/radic/examples> radiccc -help
This is a program to compile or link MPI programs using RADICMPI
It should be used just like the usual C compiler
For example,
   /users/angelo/radic/bin/radiccc -c foo.c
and
   /users/angelo/radic/bin/radiccc -o foo foo.o
or by combining compilation and linking in a single command
   /users/angelo/radic/bin/radiccc -o foo foo.c
In additional, the following special options are supported
    -show      - Show the commands that would be used without
                 runnning them
    -compile-info - Show how to compile a program
    -link-info - Show how to link a program
    -help      - Give this help
    -echo      - Show exactly what this program is doing.
                 This option should normally not be used.
```

Figure 5-12: The radiccc command

5.5 Using RADICMPI

Two scripts help the user to compile and to run his/her MPI applications using RADICMPI. These scripts have a similar syntax to their similes in the MPICH-1.2.7 implementation.

5.5.1 Compiling the sources

The compilation script is *radiccc*. This script serves to generate binary files from C or C++ source codes. *Radiccc* compiles the application code and links it with the RADICMPI and the BLCR libraries in order to generate a binary file that is ready to use the RADIC functionalities. Figure 5-12 contains a screen copy of the exit of the

radiccc command using the *–help* option. The figure contains the instructions about how to compile a program and about additional options that help the user in the compilation task.

5.5.2 Running the parallel program

The execution script is *radicrun*. This script spawns the application processes and the protectors throughout the cluster nodes.

First, it launches all protectors in order to build the protectors' chain. The current RADICMPI implementation uses the nodes described in the *machinefile* to form the protector's chain. Each node antecessor is the node in previous line of the file. To clarify how a protector chooses its antecessor, in Figure 5-13 represents the first lines of the *machinefile* for the cluster depicted in Figure 4-3. Using this information, the radicrun script knows that the antecessor of Node 2 is the Node 1; the antecessor of Node 1 is node 0 and so on. The antecessor of Node 0 is the last node used by the parallel application.

```
Node 0
Node 1
Node 2
Node 3
…
```

Figure 5-13: Initial lines of the machinefile for the cluster in Figure 4-3

After launches all protectors, the *radicrun* scripts launches the application processes. The binary of an application process contains the BLCR and the RADICMPI libraries.

Figure 5-14 contains a screen copy of the exit of the *radicrun* command. The *radicrun* accepts several specific options. The –ckpt option is relate to configuring of the RADIC checkpoint interval. The test options where used to activate special event log functions used for functional and debugging purpose (*-i, -olog* and *–tlog*).

The –justprot option make that *radicrun* only launches the protectors. It served only to debugging purposes. The –nomlog option turns off the message log mechanism, in order to assess the impact of message-logs in terms of execution time.

```
angelo@aoclp9:~/radic/examples> radicrun -help
radicrun [radicrun options...] <progname> [program options...]

radicrun options:
    -h
            this help
    -rsh
            use RSH instead of SSH as remote shell
    -machinefile <machine-file name>
       take the list of possible machines to run from the
       file <machine-file name>.  This is a list of all available
       machines; use -np <np> to request a specific number of machines.
    -machinedir <directory>
            look for the machine files in the indicated directory
    -nolocal
            don't run on the local machine
    -np <np>
            specify the number of processors to run on
    -t
            testing - do not actually run, just print what would be
executed

RADIC options:
    -ckpt <seconds>
           activates the RADIC FT mechanism and indicates the
           checkpoint interval

Test options:
    -justprot
           run only the protector array
    -nomlog
           deactivate the RADIC message log mechanism
    -i
           ativate the time measure instrumentation
    -olog
           enable event logs for observers
    -tlog
           enable event logs for protectors

On exit, radicrun returns a status of zero unless radicrun detects a
problem.
```

Figure 5-14: The radicrun command

In this chapter, we presented and explained RADICMPI. RADICMPI is a practical implementation of the RADIC architecture based on the MPI message-passing standard. We used RADICMPI to perform functional and experimental tests with the RADIC architecture, as we will explain in the next two chapters.

Chapter 6

Functional Validation of the RADIC Architecture

Now that we have presented the functional aspects of the RADIC architecture, it is necessary to describe the method used to validate the concepts of this architecture in practice.

Testing the functionality of a fault-tolerant architecture includes two major phases: to evaluate the system operation in the absence of failures, and to confirm that the system operates correctly in the presence of failures. To assure the correct operation of a fault-tolerant architecture for message-passing systems in the absence of failures is a relatively easy task, because the tests are limited to compare the results generated by parallel applications using the fault-tolerant architecture against the same parallel applications using a traditional message-passing implementation.

On the other hand, to evaluate the correct operation of a fault-tolerant architecture in the presence of failures is a cumbersome task. Many fault scenarios are possible during a program execution, and the exhaustive test of all these scenarios requires a large amount of work. In order to face such challenge, we had to develop a Fault Injection mechanism for RADICMPI.

Fault injection is a strategy to inject faults in a controlled way. Once all fault scenarios are established, a fault injection mechanism, which can create such scenarios, executes inside the system together with the parallel application. With this mechanism, it is possible to create a specific fault scenario and monitor the behavior of the fault-tolerant architecture in this scenario.

In this chapter, we present the tests performed in order to evaluate the functional requirements of RADIC. First, we will explain the theoretical bases of the fault injection mechanism that we have built in order to perform the tests. Next, we define the operation of the fault injection mechanism and explain our test methodology and the tests performed. Finally, we conclude with an analysis of the tests results.

6.1 Test platform

Because RADIC creates a distributed controller that shares the same cluster structure in which the parallel application is running, failures in nodes affect the operation of the RADIC controller as well as affect the parallel application. Since the RADIC controller must continue to operate after a failure, we had to assure the proper operation of such controller under faults.

Such requirement imposed an additional evaluation criterion. Now, besides we confirm that the RADIC controller was capable to manage faults and assure the correct ending of the parallel application, we must also assure that the RADIC controller was capable to endure faults in its own structure, and to continue its correct operation until the application completion.

In order to attend to both criteria: to assure that the RADIC architecture attends to the functional requirements and that the RADIC controller is able to continue its operation after failures, we created a test platform that contains a test protocol, a fault injection mechanism and a test environment.

6.1.1 Test protocol

Using the fault injection approach, we define a test protocol to rule the test procedures. Such protocol defined the following steps for each test procedure, as depicted in Figure 6-1. The protocol defines the test parameters, the data to measure, and the expected test results. After the execution of the tests, we analyze the data measured and compare them against the expected results.

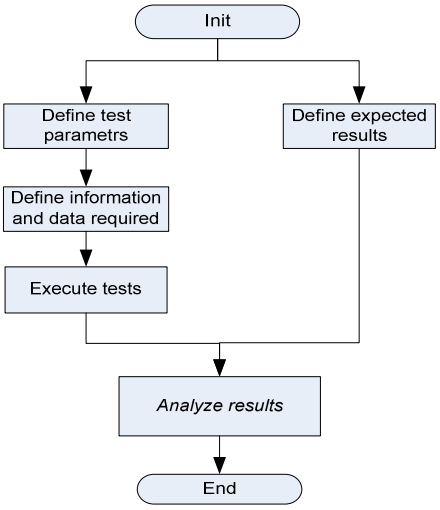


Figure 6-1: The test protocol for RADIC

In all tests, we validated the correct operation of the RADIC architecture comparing the results of the programs using RADICMPI against the same programs compiled and executed using the MPICH-1.2.7 MPI implementation, available at Argonne National Laboratory [MPICH, 2007]. If the results were the same, this meant that the application correctly ended in both cases, i.e., our architecture was operating correctly.

6.1.2 The Fault Injection mechanism of RADICMPI

Fault injection is the artificial generation of faults in a system in order to observer the system operation under fault conditions [Kopetz, 1997]. The fault injection mechanism must be able to generate faults as if they occurred in a real environment so the system acts as it is in a normal operation environment.

The fault injection approach is used achieve two goals: Testing and Debugging; and Dependability Forecast [Avresky, *et al.*, 1992]. In testing and debugging, the goal is to force fault situations that otherwise will be very difficult to verify in practice since fault are unpredictable events. In dependability forecast, the idea is to collect information about the behavior of the fault-tolerant system and relate them to the

expected fault model (types and distribution) of the environment. Such relationship serves to model the dependability of the system in the envisioned environment.

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 probabilistic distribution follows a given fault model or criteria for the system under test [Avizienis*, et al.*, 2004].

A fault injection mechanism may physically actuate over the hardware level or operate into the software level. In physical fault-injection, the target system is subject to hardware faults or adverse environment behavior that compromises the correct operation of the computer hardware. In software fault-injection, a fault injection algorithm interferes (according to a specific fault pattern or randomly) into the state of the system in order to mimic the effects of hardware fault or software design faults.

The recent literature about testing fault tolerance architectures presents some works that use fault injection in order to test fault-tolerant systems. The FAIL-FCI framework [Hoarau and Tixeuil, 2005] is an example of framework for testing distributed systems. FAIL is a fault injection language that rests on FAIL-FCI, a fault-tolerant test platform for clusters, in order to evaluate message-passing implementations [Hoarau*, et al.*, 2006]. Other works dedicate to testing of fault tolerance schemes in Grids, like the one published by Sébastien Tixeuil in [Tixeuil*, et al.*, 2006].

The mechanism that we implemented to inject faults 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.

We implemented the mechanism in the 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.

In Figure 6-3, we depict the interaction of the fault injection thread with the observer program. The figure shows that the fault injection mechanism operates as an additional thread, which executes together with the observer threads. The thread appears as a bold object in the figure. In the observer program, the fault injection thread interacts with the observer engine, in order to create the fault scenarios according with the events managed by the observer.

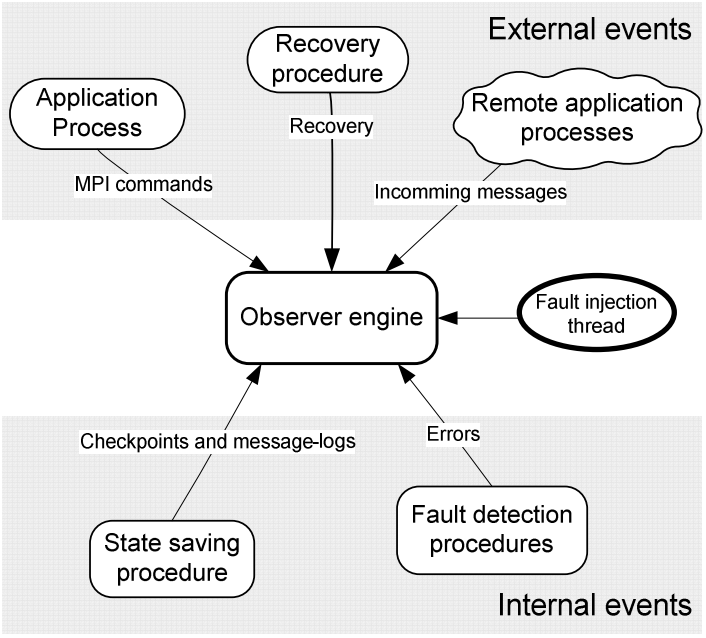


Figure 6-2: The fault injection thread inside the observer program.

In Figure 6-3, we depict the interaction of the fault injection thread with the protector program. The figure shows that the fault injection mechanism operates as an additional thread that executes together with the antecessor, successor and observer management threads. The thread, which appears as a bold object in the figure, interacts with all other threads in order to create the fault scenarios according with the events managed by the protector.

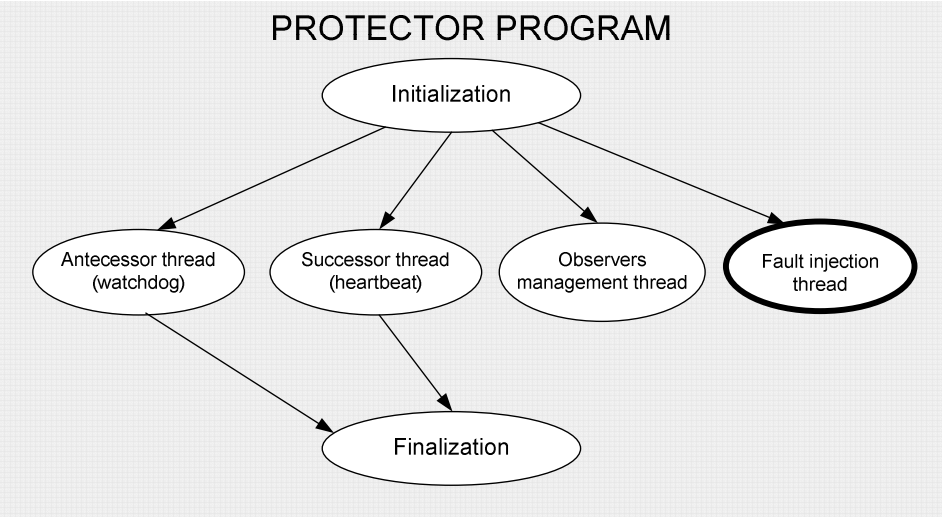


Figure 6-3: The fault injection thread inside the protector program.

6.1.3 The internal event-log mechanism of RADICMPI

To monitor the correct operation of the RADIC controller for fault tolerance, we implemented an event monitor that registered all activities of the observers and protectors that are the core elements of the RADIC controller. We built the internal event-log mechanism inside the program codes of observers and protectors, in order to monitor every single activity required to confirm the correct operation of the RADIC mechanism.

There are two types of event log mechanisms: one for debugging purposes and other for time measurement purposes. The user starts these mechanisms using the options of the *radicrun* command, as described in Figure 5-14. The *-i* option activates the time log mechanism. The *-olog* and *-tlog* options activate the debug log mechanism for observers and protectors, respectively.

Log Mechanism for Debugging

As its name indicates, the debug log mechanism served to help us in the development of the RADICMPI software. The mechanism registers 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 (the Figure 6-4 depicts a database of a protector).

Table 6-1: Fields of the debug log database

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

In Figure 6-4, there is an example of a typical debug log database. Each register has five fields that correspond to the columns in the figure. The figure refers to the log base of the protector 0, identified by T0 in the first column. The following columns correspond to the Event ID, the Event time, the name of the function that generated the event and the name of the event.

Log Mechanism for Time Measurement

The time log mechanism registers the activities of the RADIC elements in a temporary database inside the volatile memory. Each field of the database represents a event data, as described in Table 6-2. Store the log data in the volatile memory reduces the interference caused by the log mechanism over the execution of the program, because these memories are faster than disks.

The mechanism flushes the buffers to the program standard output immediately before the program ending. Figure 6-5 depicts an example of the output generated by the event log mechanisms. Each line corresponds to a register of the database. Each column corresponds to a field of the register according to the description in Table 6-2.

6.1.4 Clusters used in the test

Table 6-3 describes the clusters used to execute the tests. The diversity of clusters served to assure that our architecture was capable to adapt to different systems. We present in this thesis the test results obtained with the cluster named Homogeneous II, because it allowed the larger one.

```
T0  37      0  main                    : ANTE=aoclp24
T0   2      0  watchdog_thread         : Thread created
T0   2      0  heartbeat_thread        : Thread created
T0   8      0  link2neighbour          : successor Connection refused
T0   8      0  connect                 : successor OK
T0   8      1  connect                 : antecessor OK
T0  33      1  main                    : PRE_ANTE= 192.168.65.23 -
ANTE= 15 - SUC= 1
T0  45      1  obs_managing_thread     : Command 24 from observer 0
T0  10      1  manage_local_observers  : + obs 0 from node -1. 1
observers attached
T0  45      2  obs_managing_thread     : Command 12 from observer 1
T0  10      2  manage_observers_list   : + obs 1 from node 1. 1
observers attached
T0  45      4  obs_managing_thread     : Command 19 from observer 1
T0  14      4  storage_message_log     : Logging 4500000 Bytes of
message 0 from source 5
T0  15      4  storage_message_log     : Total log files is now 4500032
Bytes
T0  36      4  obs_managing_thread     : Reception finished : 4500000 B
/ 0.453168 s
T0  45    172  obs_managing_thread     : Command 19 from observer 1
T0  14    172  storage_message_log     : Logging 4500000 Bytes of
message 0 from source 2
T0  15    173  storage_message_log     : Total log files is now 9000064
Bytes
T0  36    173  obs_managing_thread     : Reception finished : 4500000 B
/ 0.429880 s
T0  45    185  obs_managing_thread     : Command 19 from observer 1
T0  14    185  storage_message_log     : Logging 4500000 Bytes of
message 1 from source 5
T0  15    185  storage_message_log     : Total log files is now
13500096 Bytes
T0  36    185  obs_managing_thread     : Reception finished : 4500000 B
/ 0.393857 s
T0  45    302  obs_managing_thread     : Command 13 from observer 1
T0  45    302  obs_managing_thread     : Command 18 from observer 1
T0  45    302  obs_managing_thread     : Command 25 from observer 1
T0   2    302  recv_ckpt_by_socket     : /tmp/angelo/radic/1.ckpt
T0   2    302  recv_ckpt_by_socket     : Receiving checkpoint...
T0   2    304  recv_ckpt_by_socket     : Checkpoint successfully
received
```

Figure 6-4: An example of the debug log database of a protector

6.1.5 Algorithms used in the tests

To conduct the tests, we needed a collection of algorithms that could help us to validate the functionality of RADIC and to assess the influence of the RADIC controller for fault tolerance over the cluster and the parallel application. For this, the algorithms should attend to two requisites:

- Have a deterministic runtime and communication-to-computation ratio, in order to facilitate the assessment of the influence of the RADIC parameters over the behavior of the parallel application.
- Enable the control of the message pattern, in order to facilitate the configuration of the fault scenarios;

Table 6-2: Fields of the debug log database

Column	Field name	Description
1	Source ID	Rank of the source element. To differentiate observers from protectors, the rank of protectors appeared increased by 1000
2	Event time	Elapsed time (in seconds) since the program startup
3	Event group	Identifies the events that are part of an atomic operation like, for example, an MPI_Recv or a checkpoint.
4	Event type	Identifies the event type.
5	Message	A brief description of the event
6	Destination ID	Rank of the destination element. To differentiate observers from protectors, the rank of protectors appeared increased by 1000
7	Event size	Event size (in bytes). A -1 indicates that the field is not used
8	Checkpoint size	Protectors: Total size (in bytes) of all checkpoint files in the node Observers: Not used (always -1)
9	Message log size	Protectors: Total size (in bytes) of all message log files in the node Observers: Not used (always -1)

The first requirement serves to control the behavior of the parallel application in terms of its execution time, in order to evaluate the impacts of RADIC over the application performance.

```
1,  4.629484,   0,  17,RECV_FIN,   0,       4,       -1,       -1
1,  4.629488,   2,  16,RECV_INI,   0,      -1,       -1,       -1
1,  4.884998,   3,   6,MLOG_INI,1000,      -1,       -1,       -1
1,  5.312595,   3,  11,MLOG_FIN,1000,       4,       -1,       -1
1,  5.320664,   2,  17,RECV_FIN,   0, 2880000,       -1,       -1
1,  5.320676,   4,  16,RECV_INI,   0,      -1,       -1,       -1
1,  5.568334,   5,   6,MLOG_INI,1000,      -1,       -1,       -1
1,  5.814287,   5,  11,MLOG_FIN,1000, 2880000,       -1,       -1
1,  5.823356,   4,  17,RECV_FIN,   0, 2880000,       -1,       -1
1, 93.276664,   6,  12,SEND_INI,   0,      -1,       -1,       -1
1, 93.318559,   6,  15,SEND_FIN,   0,       4,       -1,       -1
1, 93.318563,   7,  12,SEND_INI,   0,      -1,       -1,       -1
1, 93.996105,   7,  15,SEND_FIN,   0, 2880000,       -1,       -1
```

Figure 6-5: A typical set of registers in the time log database of an observer

The second requirements are necessary to configure the different experiments that will confirm that RADIC is capable to operate as a scalable, decentralized, flexible and transparent fault-tolerant architecture.

Table 6-3: Clusters used in RADIC validation

Cluster	Heterogeneous	Homogeneous I	**Homogeneous II**
Nodes	6	12	**16**
Processors	Pentium-4 / Pentium-III / Ahtlon-XP2600+	Athlon-XP2600+/ 1.9GHz/ 512KB L2 cache	**Pentium-4/ 1.8GHz/ 512KB L2 cache**
Memory	256 / 128 / 256MB	256 MB	**512MB**
Disk/node	20 / 10 / 40 GB ATA	40GB ATA	**30GB ATA**
Network	100-baseTX hub	100-baseTX switch	**100-baseTX switch**
Operating System	Linux Fedora Core 3 Kernel 2.6.9	Linux Fedora Core 4 Kernel 2.6.17	**Linux Fedora Core 2 Kernel 2.4.18**
Compiler	gcc v3.4.2	gcc v4.0.2	**gcc v3.4.5**
Checkpoint library	BLCR v0.4.2	BLCR v0.4.2	**BLCR v0.4.2**

Basing on these requirements, we choose three programs for performing the tests: a simple ping-pong program, a matrix-multiplication program using the MW (Master/Worker) paradigm, and a matrix-multiplication program using the SPMD paradigm (Cannon's algorithm).The ping-pong program was the better program to evaluate the message-passing scheme, because it allows a complete control of the communication between the application processes. We used this program to simulate different message patterns, in order to confirm the correct operation of message-passing mechanism.

We choose the matrix-multiplication algorithms because their MW and SPMD algorithms generate distinct message patterns, facilitating the creation of different fault scenarios. As shown in Figure 6-6, the MW algorithm has a 1-to-N message pattern (Figure 6-6a). 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 6-6b). Each application process communicates with their neighbors.

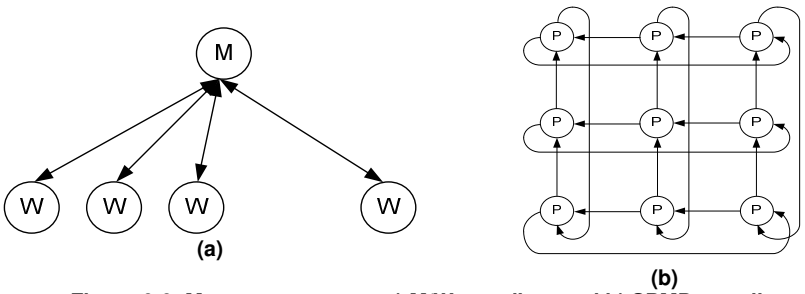


Figure 6-6: Message patterns - a) M/W paradigm and b) SPMD paradigm.

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: static and dynamic. 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.

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.

6.2 Validation of the message-passing mechanism

The first group of tests served to evaluate the behavior of the of the RADICMPI implementation as if it was a normal MPI implementation. For this, we used MPICH-1.2.7 [Argonne National Laboratory, 2007b] as the reference implementation.

The current version of RADICMPI contains all ideas to the RADIC architecture in the form of a reduced MPI implementation and the main goal of RADICMPI was to test the RADIC fault tolerance functionalities in a real environment, instead of to operate as a new fault-tolerant MPI implementation. Although it may seem meaningless to test RADICMPI with its fault tolerance capabilities deactivated, these tests served to several purposes:

a) To confirm the transparency of RADICMPI, using the same application codes in all tests;

b) To validate the message-passing mechanism of RADICMPI;

c) To serve as reference for calculating the time overheads caused by the operation of the fault tolerance mechanism;

d) To evaluate the applications behavior;

Tests with the master/worker matrix multiplication algorithms

Figure 6-7 shows the execution times (in seconds) as a function of the number of nodes in the cluster, required to multiply two 3000x3000 matrices with double-float elements using the Master/Worker matrix multiplication programs with static and dynamic load balance. In all experiments using N nodes, there is a master and N-1

workers. We can see that the programs present a similar behavior. As expected, the execution time decreases as the number of nodes increase.

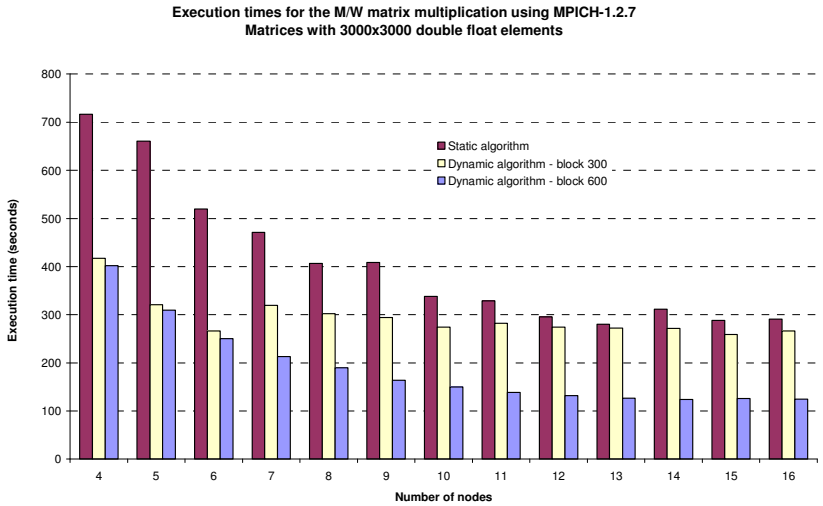


Figure 6-7: Execution time of the M/W matrix multiplication programs using MPICH-1.2.7 for 3000x3000 double element matrices.

Analyzing the curves, we see that the program using dynamic balancing scaled better than the program using static load balance. This occurred because the communications in the static program are concentrated by master at the beginning and at the end of computation, causing a slow down of the whole system.

The difference between the behaviors of the dynamic balance program using different computation-to-communication ratios confirms that the communication with the master greatly influences the scalability of the application.

In Figure 6-8, we show the speedup curves for the programs, using the execution with four nodes as reference. We can see that the program using blocks with 600x600 elements presents a better performance and a regular increase in the speedup. Blocks with 300x300 elements yield a computation-to-communication ratio worse than blocks with 600x600 elements, because there is more communication between the master and the workers. The master has to attend to several workers at a time, enlarging the communication time between the workers and the master and slowing down the whole system. With blocks of 600x600 elements, there is less

communication between the master and the workers and the communications become are better distributed. This reduces the communication time between the workers and the master and improves the system performance.

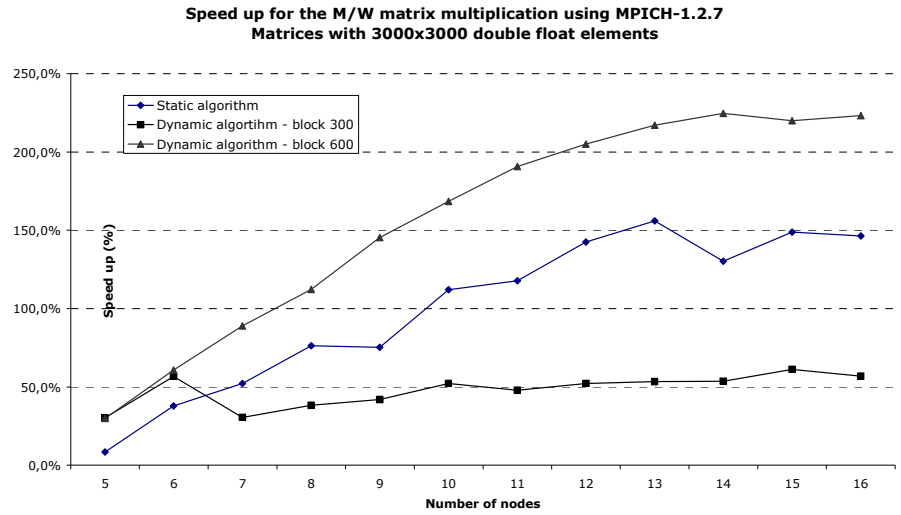


Figure 6-8: Speedup of the M/W matrix multiplication programs using MPICH-1.2.7 for 3000x3000 double element matrices

To test the programs with RADICMPI we simply compiled the same source codes with *radicc* and executed the binaries with *radicrun*. We confirm the correctness of the programs by comparing the results generated using RADICMPI with the results generated using MPICH-1.2.7.

The behavior of the programs with RADICMPI, in terms of scalability, was completely similar to the behavior of the programs using MPICH-1.2.7. Again, and because the same reasons we have explained for MPICH-1.2.7, the program with static load balance presented the larger execution times. Similarly, the program with dynamic balance using blocks with 300x300 elements had a worse performance than the program using blocks with 600x600 elements.

We can see the execution times of the programs using RADICMPI in Figure 6-9. For all M/W algorithms, the execution time (in seconds) reduces as the number of nodes increases, according to the same tendency presented by MPICH-1.2.7.

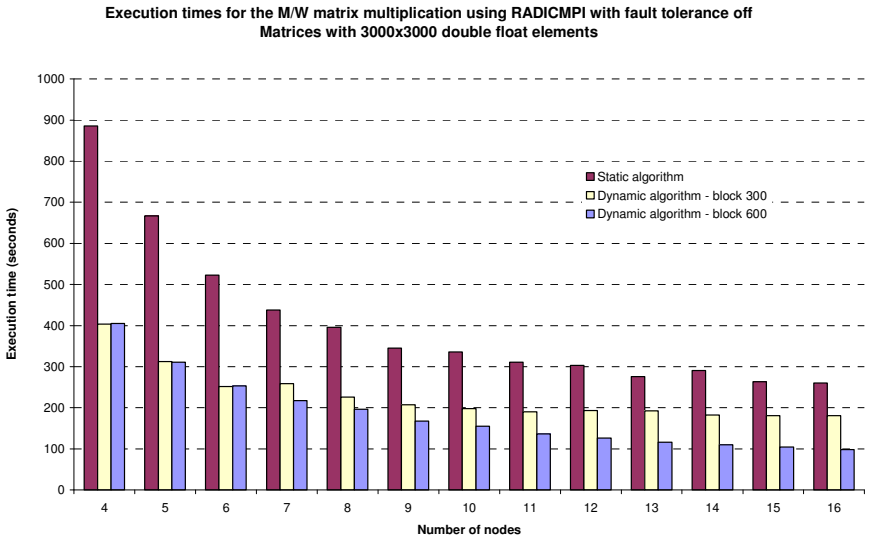


Figure 6-9: Execution time of the M/W matrix multiplication programs using RADICMPI (with fault tolerance off) for 3000x3000 double element matrices.

The similar behavior of RADICMPI with fault tolerance off and MPICH-1.2.7 is also observable in the speedup curves for the matrix multiplication programs. The curves in Figure 6-11 show the speedup for the cluster size since five until sixteen nodes, using the execution time with four nodes as reference. The curves present a similar tendency with the curves for MPICH-1.2.7 (Figure 6-8); however, there are differences between the performance of RADICMPI and MPICH-1.2.7.

Comparing the curves in Figure 6-8 against the curves in Figure 6-10, one can see that RADIMPI with fault tolerance off presents a better scalability than MPICH-1.2.7. The better behavior of RADICMPI in this case, is consequence of the message reception engine in the observers (see the observer operation in paragraph 5.3). In the current RADICMPI implementation, there are three threads dedicated to receive messages. Each observer has a temporary buffer that keeps a received message until the application requires it. Therefore, this multi-thread implementation associated with the temporary buffer of RADICMPI, improves the communication efficiency for the blocking MPI functions, because until three senders may communicate with a single observer at same time.

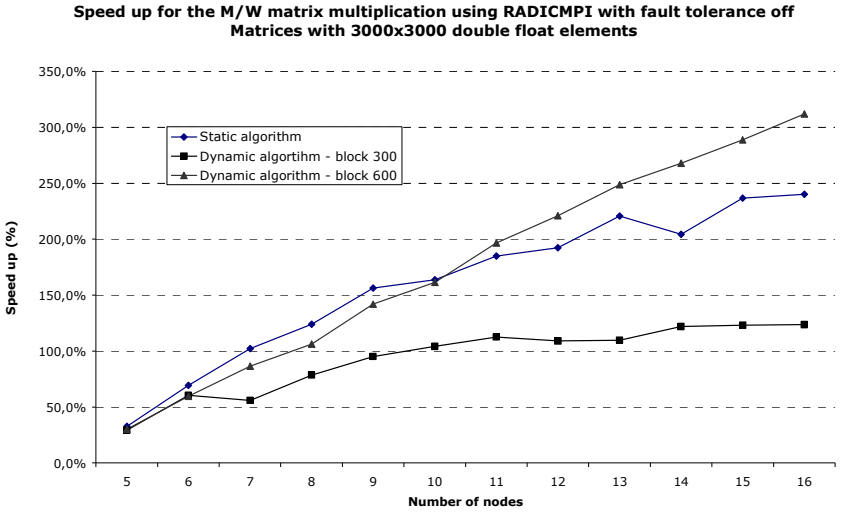


Figure 6-10: Spped-up of the M/W matrix multiplication programs using RADICMPI (with fault tolerance off) for 3000x3000 double element matrices.

Furthermore, because MPICH-1.2.7 is a full MPI implementation oriented to run in different systems. Because of this, MPICH has to execute several internal activities to assure its correct operation, independent of the structure of the parallel computing in which it runs. On the other hand, RADICMPI without fault tolerance is a partial MPI implementation oriented to a specific parallel computer structure. Therefore, the operation of RADICMPI is simpler than the operation of MPICH-1.2.7.

Tests with the SPMD matrix multiplication algorithm

The communication pattern of the Cannon's algorithm (SPMD) defines a square mesh topology, as depicted in Figure 6-6b. Because of this restriction, the number of nodes is a square power of the mesh dimension. For instance, a 3-dimensional mesh requires nine nodes; a 4-dimensional mesh requires sixteen nodes and so on. In the Cannon's algorithm, each node starts with blocks of the operand matrices and finishes with a block of the result matrix.

In order to assure that all blocks would have the same size, the size of the matrices must be an integer multiple of the block size. The number of nodes in the cluster and the requirements of the mesh topology, have allowed us to execute tests

with four, nine and sixteen nodes. Basing on these cluster sizes, we selected matrices with size multiple of these numbers for assuring that the blocks had the same size in each node. In this thesis, we present the results for matrices with 3024x3024 elements.

Figure 6-11 shows the results of the tests conducted with the SPMD matrix multiplication program using MPICH-1.2.7 and RADICMPI with the fault tolerance functions deactivated. The figure shows the execution times (in seconds) as a function of the number of nodes in the cluster. We can see that both implementations have the same behavior in terms of scalability. Again, the differences between the message reception mechanism of RADICMPI and of MPICH-1.2.7 cause that the programs using RADICMPI present a better performance than the programs using MPICH-1.2.7.

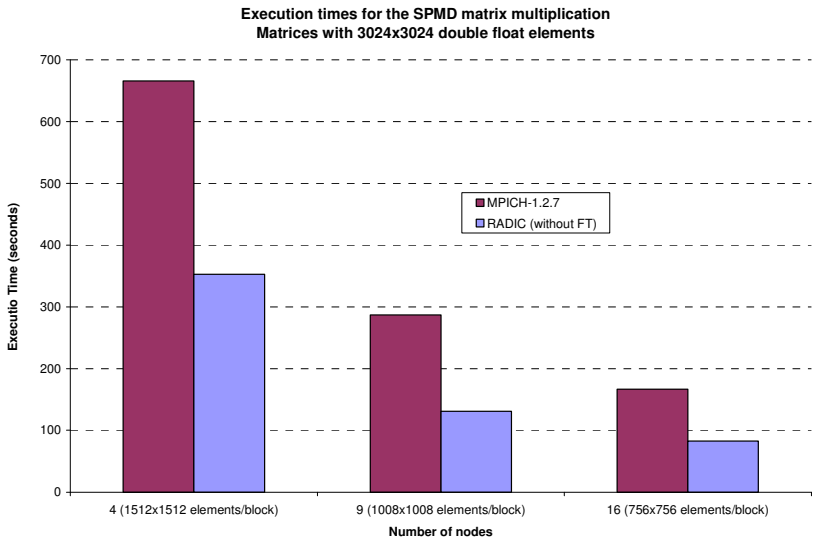


Figure 6-11: Results of the SPMD (Canon algorithm) matrix multiplication program using RADICMPI with fault tolerance deactivated and using MPICH-1.2.7.

We calculated the speedup of the programs for both implementations as a function of the number of nodes in the cluster, using the execution with four nodes as reference. The results presented in Table 6-4 show that speedup for both implementations have the same magnitude, confirming that RADICMPI does not affect the behavior of the application scalability.

Table 6-4: Speedup for the SPMD matrix multiplication programs using MPICH-1.2.7 and RADICMPI with fault tolerance off

Nodes	MPICH-1.2.7	RADICMPI (fault tolerance off)
9 (3x3)	232%	270%
16 (4x4)	400%	425%

6.3 Functional tests without failures

After validating RADICMPI as a common MPI implementation, we started the tests with the fault tolerance mechanism activated. The tests in the absence of failures, served to assure that:

a) the fault-tolerant architecture satisfy to the functional requirements of RADIC;

b) the operation of the fault tolerance mechanism implemented in RADICMPI did not interfere in the application 's correctness;

c) the flexibility of RADICMPI allow the configuration of different protector's structures.

Furthermore, the tests without failure offered the time references in the experiments that we used for studying the impact of the fault tolerance parameters over the system performance. We will comment about such experiments in the next chapter.

To validate that all applications executed correctly using RADICMPI with the controller for fault tolerance activated, we compared the results obtained using RADICMPI against the results obtained with MPICH-1.2.7. In all cases, the application generated the same results, confirming that the operation of the RADICMPI controller for fault tolerance did not compromise the application correctness.

To validate if the fault-tolerant architecture attended to the functional requirements of the RADIC architecture, we checked if the system accomplished each

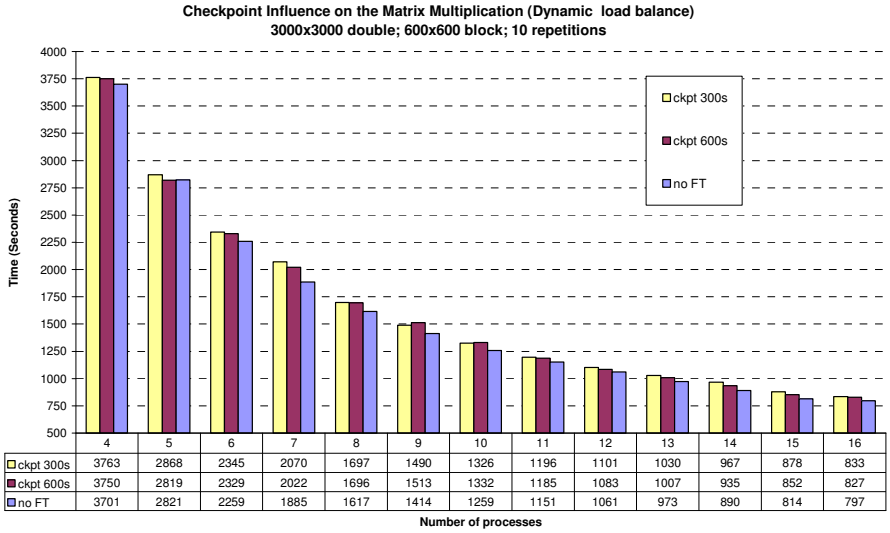
individual feature of RADIC. We resume the results for transparency, decentralization and flexibility as the checklist presented in Table 6-5. In this checklist, we represented the feature and the criteria we used to confirm each feature.

Table 6-5: Checklist for the features RADICMPI

Feature	Validation criteria
Transparency	Same source code used in a common MPI implementation generated the same application results Fault tolerance operation is automatic
Flexibility	The system allowed the use of different checkpoint intervals and different protector structures
Decentralization	We did use neither a central nor a dedicated node to support the fault tolerance activities

In order to validate the scalability criterion, we executed the testing programs and observed two effects: how the operation of the fault tolerance mechanism affected the scalability of the system and how the RADIC controller scaled. In Figure 6-12, we represent the execution times of the M/W program using dynamic load balance and RADICMPI. In order to enlarge the execution times, we modified the algorithms to repeat the number of block multiplication in each node. In these experiments, each node multiplied each matrix block ten times. Therefore, we could better evaluate the effects of the number of checkpoints over the execution time.

The Figure 6-12 contains the results for three scenarios: without fault tolerance and with fault tolerance using two checkpoint intervals. The enlargement in the execution times increases when the checkpoint interval is shorter because more checkpoints occur during the program execution. We may see that



	4	5	6	7	8	9	10	11	12	13	14	15	16
ckpt 300s	3763	2868	2345	2070	1697	1490	1326	1196	1101	1030	967	878	833
ckpt 600s	3750	2819	2329	2022	1696	1513	1332	1185	1083	1007	935	852	827
no FT	3701	2821	2259	1885	1617	1414	1259	1151	1061	973	890	814	797

Figure 6-12: Execution times for the matrix multiplication M/W program using dynamic load balance and RADICMPI with different checkpoint intervals

6.4 Functional tests with failures

The main difficulty of these functional tests was to inject faults in the critical points of the system, because the communication' state of a process strongly interferes in the behavior of the fault tolerance when the process fails.

Failures occur randomly. Therefore, a failure may occur during any system activity. To assure that RADIC can manage any kind of failure, we had to specify all possible failure scenarios and build a fault injection mechanism that could create each failure scenario. The fault injection mechanism followed the general algorithm described in Figure 6-13.

For the test scenarios with faults, our unique premise was that in a scenario with multiple faults, a fault never occurred in an element that was involved in the recovery of a previous or simultaneous fault while the recovery process was in progress. In paragraph 4.1.3, we have argued that such premise is indeed less restrictive than it seems *a priori.*

Table 6-6: Fault scenarios according to the elements involved in the fault

The fault occurs when…	Scenarios
The communication is already established between two elements	An observer is transmitting a message to another process An observer is transmitting a checkpoint or a message log transmission to its protector A protector is receiving a checkpoint or message log from its observer
An observer tries to communicate with another element	An observer starts a message transmission to another process An observers starts a checkpoint or a message log transmission to its protector
No communications are present	A process is just computing An observer is idle

We divided the tests according to the instant when a fault occurs in the system. There are two major types of scenarios: when two elements are communicating and when an observer starts a communication. We summarize these scenarios in table Table 6-6.

In order to explain the tests with failure, we will consider the typical cluster structure depicted in Figure 4-3. This cluster contains nine nodes (N_0-N_8), each one running a RADIC protector process (T_0-T_8) and nine application processes (P_0-P_8), each one attached to an observer (O_0-O_8). The sequence depicted in Figure 4-11 is an example of what happens to the cluster structure if a failure occurs.

For each scenario, we defined how the elements of the RADIC architecture should operate when a fault occurs, and validate the operation of the elements using the event-log mechanism.

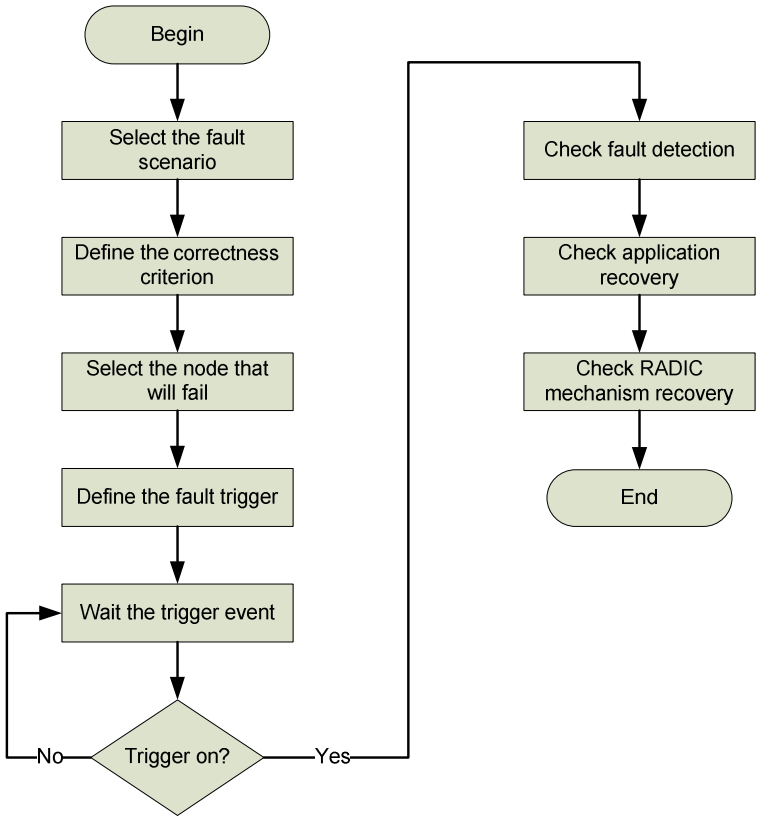


Figure 6-13: Algorithm of the fault injection mechanism

6.4.1 Failure when a process is just computing

This is the simplest case because the application process is not interacting with any other application process in the cluster when the fault occurs. The fault detection and the recovery procedure actions are:

1. The protector of the faulty application process detects the fault and mark the process as recovering;
2. The protector starts the recovering procedures for the processes that were in the faulty node.
3. Every recovered observer establishes a new protector and sends the current message log file and checkpoint file to it. The observer now has two protectors:

the actual one and the new one. These two protectors mark the status of the application as recovering.

4. Every recovered observer opens the message log file and, for each message requested from its application process during recovering time, delivers the messages from the log file until the application request a message that is not in the log.
5. When the application requests the last message from the log, the observer communicates to the protectors that the recover is complete.
6. The recovered observer reopens its communication channel. Henceforth its application process may receive messages from another application processes again.
7. The recovery procedure terminates. The observer terminates its relationship with the old protector and signal to the new protector that the application process is ready.

6.4.2 Failure while a process is communicating

Since there are two observers (source and destination) affected by the fault, we are going to evaluate each one point of view. The destination observer simply discards the incomplete message that it was receiving from the faulty source. The destination observer "knows" that the source will recover and will try to communicate again in the future.

The source observer should run the following algorithm:

1. Look at the *radictable* in order to find the destination protector address;
2. Contact the destination protector.
3. If the communication with the protector of the destination process fails, then starts the algorithm to find the new protector of the faulty process and repeats step 2. If the communication is ok, update the *radictable* to replace the new protector of the destination process.

4. If the protector answers that the peer is checkpointing or recovering, then wait and repeat step 2;
5. If the protector informs that the peer is ready, update the destination address in the *radictable*.
6. Send the message.

At the end of this procedure, the *radictable* of the source observer will have the new address of the destination process together with the address of the new protector of the destination process.

6.4.3 Failure when a process starts a communication

This scenario is similar to what happens with a source observer if a fault occurs when two process are communicating. The only difference is that the source observer will not start the communication, because the destination process is faulty. The source observer will follow the same steps described in paragraph 6.4.2.

6.4.4 Failure when an observer is checkpointing or logging a message

In both cases, the watchdog of the protector must have identified that the node of the observer has failed. Therefore, the protector of the process will start the procedure to recover the observer.

6.4.5 The protector node fails when the observer is idle

This case occurs if no observer is communicating with a protector when it fails, i.e., the observer is neither receiving a message from another process (and hence the observer is not saving a log in the protector) nor performing a checkpoint.

In this case, the local protector informs to the observer that its neighbor is faulty. When this occurs, the observer must immediately perform a checkpoint. The observer will follow a procedure similar to the procedure it follows when the protector fails during a message log or checkpoint.

6.4.6 The protector node fails when the observer is checkpointing

In such case, the observer should execute the following algorithm:

1. Use the *radictable* to calculate the antecessor address of this current protector. From now on, the observer consider this antecessor as its protector;
2. Connect to the protector;
3. If the connection fails because the protector is faulty, then repeat step 1 to find a new protector. If the protector answers, confirms it as the current protector;
4. Transmit the pending checkpoint;

6.4.7 The protector fails when the observer is transmitting a message log

If the protector fails while an observer is logging a message, such observer should follow the next sequence:

1. Use the *radictable* to calculate the antecessor address of this current protector. From now on, the observer consider this antecessor as its protector;
2. Connect to the protector;
3. If the connection fails because the protector is faulty, then repeat step 1 to find a new protector. If the protector answers, confirms it as the current protector;
4. Take a checkpoint, such checkpoint includes the message received, and transmit it to the new protector;

In this chapter, we validated the fundamental functionalities of the RADIC architecture in a real environment. We used a group of different applications that served to experiment the transparency, the flexibility, the decentralization and the scalability of the RADIC architecture. Each application had different communication-to-computation ratios and generated different message patterns. The evaluation of all scenarios rested on the event-log mechanism of RADICMPI and, of course, on the

comparison of the results generated by the applications running in scenarios with failures against the results generated in scenarios without failures.

In the next chapter, we present experiments that served to better understand the operation of the RADIC controller in terms of the impact of its parameters have over the performance of the application.

Chapter 7

Experiments with RADIC

In the last chapter, we validated the RADIC architecture in terms of its functional requirements: decentralization, flexibility, scalability and transparency. We implemented in RADICMPI a fault injection mechanism and an event log mechanism, which helped us to assess the behavior of the system in several scenarios.

Now we will discuss the practical aspects related to the design of RADIC architecture and will analyze how the RADIC parameters may affect the application's behavior. For this, we focused on the impact of the operation of RADIC over the application performance. This impact is consequence of two factors: the reduction of the cluster structure caused by the resources consumed by the fault tolerance scheme, and the interference caused by the operation of the fault-tolerant architecture over the application.

Any fault-tolerant architecture for message-passing systems consumes resources during its operation. For schemes based on rollback-recovery protocols, these resources consumed are, typically, disk storage, network bandwidth, and computational time. For the final user, the cost of a fault-tolerant architecture is often associated with the enlargement of the execution time of his/her application. On the other hand, for a system administrator, the requirements of disk storage or memory per node are also relevant.

The operation of the RADIC controller for fault tolerance consumes the cluster's resources that, originally, would be available for the parallel application. As a result, the application "sees" a parallel machine with fewer resources than the resources actually available. Such reduced machine will execute the application in a time larger than it would execute without fault tolerance. Additionally, the checkpoint and message log procedures directly interfere in the application behavior. The

computation of a process stops while the checkpoint procedure occurs, and the message log procedure increases the message latency.

We studied the interaction between the operation of the RADIC controller and the application using RADICMPI, focusing on how the parameters of the controller could influence the system performance, i.e., how the RADIC parameters influence the overhead caused by the operation of the fault tolerance mechanism.

As we have already discussed, the effect of such overhead is an enlargement in the execution time of the application and a reduction in the cluster capacity. In the RADIC architecture, the main factors that influence the overhead are:

a) The checkpoint's interval and the checkpoint's cost;

b) The application's message pattern and the cost of message log;

c) The interaction between checkpoints and messages with the application processes;

The combination of all these factors make difficult to model a general function that represents the cost of the RADICMPI fault tolerance mechanism. Nevertheless, it is possible to study the influence of the RADIC parameters for a specific application. Therefore, we conducted experiments in order to evaluate the overall impact of the interaction between checkpoints and messages in our testing programs.

These experiments consisted in, for a given application, to study the influence of each RADIC parameter over the application behavior. In RADICMPI, these parameters were: a) the watchdog/heartbeat cycle; b) the protector's structure and c) the checkpoint interval.

7.1 Experiments with the watchdog/heartbeat cycle

The heartbeat mechanism of RADICMPI creates a continuous flow of short messages (the heartbeats) between the nodes. In Figure 4-8, we can see an example of how the protectors communicate using the heartbeats.

Because, in our testing system, these short messages passed through the same network structure used by the application processes to communicate, we conducted experiments to measure the influence of the heartbeat over the message latency and the network throughput.

The tests consisted in use the ping-pong program to create several message patterns between the processes. Then, we execute the program in the system without fault tolerance and measure the message latency and network throughput for each message pattern.

To assess the influence of the heartbeats, we first executed the ping-pong program without fault tolerance and measured the message latencies and network throughput.

Next, we used the option *-justprot* available in the command *radicrun* (see Figure 5-14.) in order to create only the protectors of the RADIC controller for fault tolerance. The protectors started all the threads and the watchdog/heartbeat mechanism, but since there are no observers active, the only network traffic generated by RADIC comes from the heartbeats.

Since the literature about fault tolerance does not make any reference about what should be the best cycle four our architecture, we could base only in practical concerns to guide us in the election of a heartbeat cycle. Considering that the watchdog/heartbeat cycle defines how fast a protector detects a failure in its neighbor, we assumed that heartbeat cycle of one second would be fast enough for systems which intend to execute long-time applications.

With all heartbeats active, we re-executed the ping-pong program, re-created the same message patterns, and measured again the message latencies and the network throughput for each message pattern. Finally, we compared the results obtained in the executions without fault tolerance against the executions with fault tolerance using only watchdog/heartbeat messages.

The comparison between the results has shown that, in the testing cluster used for the experiments, the influence of the heartbeats over the message latency or the

network throughput was imperceptible. This result indicated the heartbeat messages created a negligible influence over the cluster structure.

7.2 Experiments with the protectors structure

In Chapter 6, we conducted experiments to validate that RADICMPI allowed different protector's structures, like in the example depicted in Figure 4-14. These experiments served to verify the flexibility of the RADIC architecture.

We used the same experiments to verify if the protectors' structure would influence the performance of the application or the performance of the controller for fault tolerance. Figure 4-14 represents an example of how it is possible to make an arrangement of two chains in a cluster of nine nodes. We used the same strategy to make four distinct structures in a cluster with sixteen nodes: a) two protectors' chains with eight nodes; b) three protectors' chains (two with five nodes and one with six nodes); c) four protectors' chains with four nodes; d) a chain with all nodes.

In scenarios without failures, we did not perceive any influence of the protectors' structure over the application performance of the RADIC controller performance. Then, using the fault injector, we injected faults individually and simultaneously in the protectors of each chain in order to verify how the fault tolerance mechanism would operate in the different scenarios. In all tests, we respected the limit of faults per chain imposed by Equation 3 and, again, we did not assess any difference between the operations of the systems with different protector structures.

Basing on the test results, we assessed that the protector structure does not cause impact over the performance of the RADIC controller. The only limit that the protector structure causes over the RADIC operation is the maximum number of failures that the system can support.

7.3 Experiments with the checkpoint interval

From all parameters of RADIC, the checkpoint interval was, by far, the parameter that has had more influence over the behavior of the system. This great influence of

the checkpoint over the system's behavior is consequence of the interference of the checkpoint over the operation of the parallel application.

In RADICMPI, such interference comes from three factors: a) a process must stop during a checkpoint; b) a process cannot communicate during a checkpoint and c) a checkpoint increases the network traffic, and interferes on the message latency and on the network throughput.

The first factor is obvious; a checkpoint is a snapshot of the process state. Therefore, the checkpoint procedure must stop a process in order to take a steady state of it. The process remains stopped until the finish of the checkpoint procedure. The longer the checkpoint procedure takes, the longer the enlargement in the runtime of the process.

The second factor occurs because of the first one. The process' state changes when a process communicates. Since a checkpoint is a snapshot of the process's current state, this state may not change while a checkpoint occurs, i.e., a process may neither receive nor transmit messages while a checkpoint is in progress. Similarly, the checkpoint procedure must retard the beginning of a checkpoint if the process is communicating.

Such compromise between checkpoints and messages causes a delay in the beginning of the checkpoint procedure if a checkpoint should starts when a process is communicating. Similarly, checkpoints delay message transmissions because of two reasons: a) they retard the process normal execution; b) a process may not receive messages during a checkpoint, therefore any other process that tries to communicate with a "in-checkpoint" process have to wait until the destination concludes its checkpoint.

Finally, the third factor is consequence of the traffic naturally generated by the checkpoint procedure. When an observer takes a checkpoint of its process, it must send this checkpoint throughout the network until its neighbor protector. Therefore, the observers of the RADIC controller for fault tolerance generate a network traffic, which interferes with the messages of the parallel application.

In the experiments with the checkpoint interval, we fixed the communication-to-computation ratio and measured the enlargement in the runtime of the application for distinct checkpoint intervals.

For the tests, we choose the SPMD matrix multiplication program and the master/worker matrix multiplication program with dynamic load balance. We choose these programs because they offered more flexibility to control the communication-to-computation ratio.

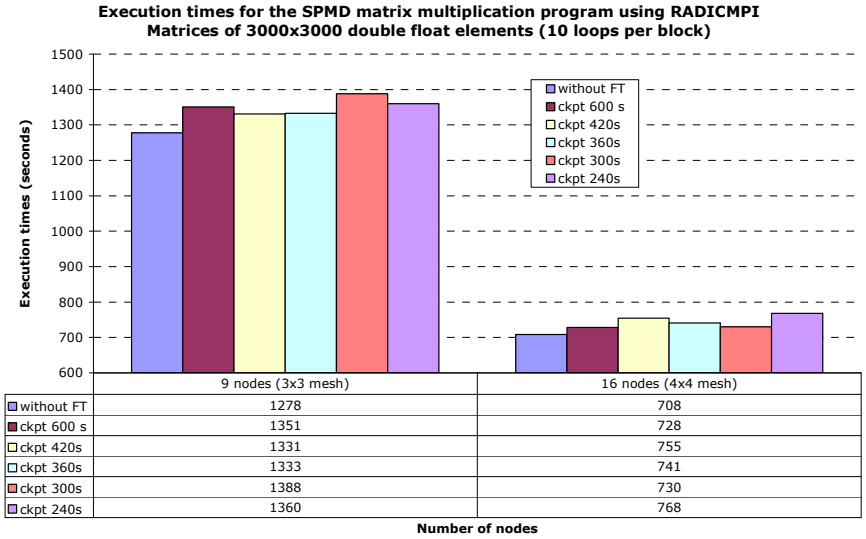
In order to enlarge the execution times, we changed the original algorithms to force that each node multiplied the matrices blocks more than once. This allowed us to take more checkpoints in each experiment.

In Figure 7-1, we present a summary of some of these tests. The figure represents the execution time (in seconds) for the SPMD matrix multiplication program, obtained for matrices with 3000x3000 double float elements. Each node repeated ten times the multiplication of the matrices' blocks.

We used two cluster sizes: nine nodes and sixteen nodes. There are five bars for each cluster in the figure. Each bar corresponds to the execution time using a specific checkpoint interval, measured in seconds. We experiment the system with different checkpoint intervals (240, 300, 360, 420 and 600 seconds) and compared the results with the executions without fault tolerance.

In this experiment with the SPMD program, the communication-to-computation ratio changed according to the cluster size. In the cluster with nine nodes, each node worked with blocks of 1000x1000 elements, and in the cluster of sixteen nodes, each node worked with blocks of 750x750 elements.

The lower computational load assigned to each node in the large cluster yielded a lower cost of the fault tolerance in the cluster with sixteen nodes. This is consequence of two factors: a) the checkpoint sizes in the cluster with sixteen nodes were smaller because each process operated with less data; b) the message logs were also smaller because each process had to communicate less data.



	9 nodes (3x3 mesh)	16 nodes (4x4 mesh)
without FT	1278	708
ckpt 600 s	1351	728
ckpt 420s	1331	755
ckpt 360s	1333	741
ckpt 300s	1388	730
ckpt 240s	1360	768

Figure 7-1: The minimum structure for a protectors' chain.

The combination of these two factors contributed to the different enlargements in the execution times. Using Equation **4**, we calculate the time overheads caused by the operation of the fault tolerance mechanism.

$$Overhead(\%) = \left(\frac{Execution time\ with\ fault\ tolerance}{Execution time\ without\ \ fault\ tolerance} - 1 \right) \times 100\% \qquad \mathbf{4}$$

The overheads presented in Table 7-1 shows how much the fault tolerance mechanism interferes in the execution time of the application process. This table shows the time overhead calculated with equation **4**, as a function of the checkpoint interval, for the experiments represented in Figure 7-1. The checkpoint interval also defined the number of checkpoints taken during the executions, represented in the table together with the overhead information.

We can see in the table that the maximum time overhead was 8,6%. Furthermore, we see that the overheads are not directly proportional to the checkpoint intervals. For example, in the cluster with 16 nodes, we see that the impact of a single checkpoint is very different depending on the moment in which the checkpoint occurs.

Table 7-1: Influence of the checkpoints over the execution time.

Checkpoint interval	9 nodes		16 nodes	
	Time overhead	# checkpoints	Time overhead	# checkpoints
240 s	6,5%	5	8,5%	3
300 s	8,6%	4	2,7%	2
360 s	4,3%	3	4,6%	2
420 s	4,1%	3	6,6%	1
600 s	5,7%	2	2,8%	1

To analyze this behavior, we used the event log information to asses how much time each checkpoint took. The results confirmed that, as we have explained before, the interaction between checkpoints and the messages modified the communication times between the processes. In practice, the interaction between the checkpoints and the application made that some processes had to spend more time inside an `MPI_Recv()` function, creating a message pattern different from the original message pattern of the application.

7.4 Experiments with faults

In these tests, our interest was to assess the impact of failures in terms of execution time. The failure's cost is a combination of distinct factors: time elapsed until detecting the fault, time spent to recover failed processes, amount of computation lost because of the failure and impact of the failure on the cluster's structure.

The RADIC parameters determine the first three factors. The watchdog/heartbeat cycle defines how long it takes to detect a failure. The recover procedure defines how long it takes to recover failed processes. The instant in which the fault occurs inside the checkpoint interval determines how long a recovered process has to roll back, i.e., how much computation the process must re-execute.

The last factor, the impact of the failure on the cluster's structure, strongly depends on the application behavior. A failure causes a reduction in the number of cluster's nodes and creates unbalance in the computational load between the nodes

that has survived after the failure. The impact of such unbalance on the execution time depends on how good the application adapts to the cluster's heterogeneity. Applications that naturally balance the computational load among the processes adapt better to the heterogeneity. Therefore, such applications suffer lower enlargements in the execution time if a failure occurs.

We have conducted experiments in order to assess the time overhead caused by failures. In these experiments, we defined the instant of the failure in terms of percentage of the total application runtime without failures.

We focused our attention in the impact of failures on the total application runtime. We consider that in a system without a fault tolerance mechanism, the application will collapse if a failure occurs in a node. Therefore, the user loses all previous computations and has to re-launch the application again. Furthermore, we also have to consider that a user is not monitoring his/her application during all time it is executing, and hence the fault detection phase and the recovery phase are not instantaneous.

Using such assumptions, we defined that, for a system without fault tolerance, the total execution time in case of a failure is expressible by Equation 5:

$$T_F = T_{lost} + T_{det} + T_{re} + T_{N-1} \quad (5)$$

Where:

T_F – Total execution time in case of a single failure

T_{lost} – Failure time. Indicates the computational time since the beginning until the moment when the failure occurs.

T_{det} – Time required for failure detection

T_{re} – Time required for application recovery

T_{N-1} – Total execution time with N-1 nodes

Our test procedure consisted in measuring the application runtimes in the following scenarios:

a) without fault tolerance using *N* and *N-1* nodes;

b) with fault tolerance, with N nodes and different checkpoint intervals, in the absence of failures;

c) with fault tolerance, with N nodes and different checkpoint intervals, in different fault scenarios.

We calculated the execution time for the application without fault tolerance using Equation 5 and compared the results against the total execution time with fault tolerance. The Table 7-2 presents the results of the experiments in the cluster with 15 nodes, for the matrix multiplication program with 3000x3000 double float elements, using blocks with 1000x1000 float elements.

At the bottom of the Table 7-2, we inform the execution times for the cluster with 15 nodes in the absence of failures. The "calculated w/o FT" column corresponds to the theoretical execution times according to Equation 5 with T_{14} equals to 1207 seconds and considering T_{det} and T_{re} equal to zero.

We unconsidered those times because we could not define how long the user would take to detect the failure and to re-start his/her program. Therefore, we assumed the best possible scenario, i.e., automatic instantaneous fault detection and program re-start. Such assumption approximates the behavior of the system without fault tolerance from the system with fault tolerance.

Table 7-2: Execution times in the presence of failures for the M/W matrix multiplication program (dynamic load balance) using 15 nodes (1000x1000 elements/block)

Fault time	calculated w/o FT (T_{14}=1207 s)	ckpt 600 s	ckpt 300 s	ckpt 200 s
fault at 250 s	1457	n/a	n/a	1082
fault at 350 s	1557	n/a	962	1079
fault at 650 s	1857	1093	1089	1099

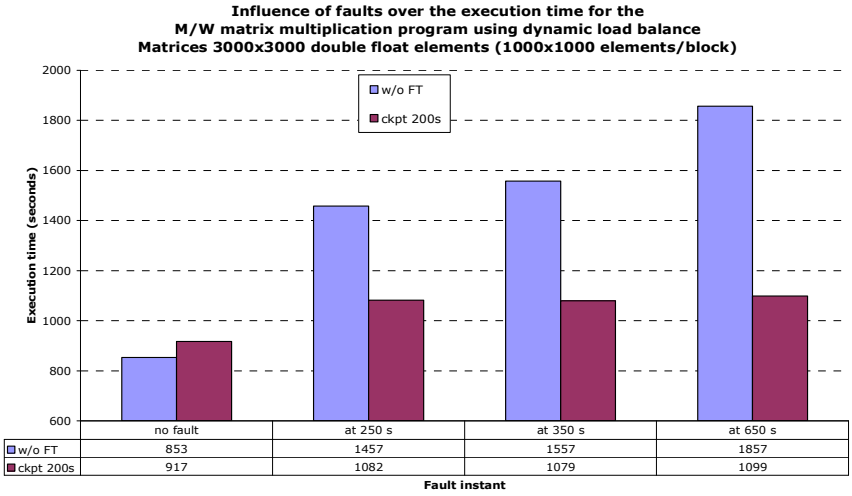
Obs.: n/a = not applicable

Execution time (in seconds) using 15 nodes in the absence of failures			
w/o FT	ckpt 600 s	ckpt 300 s	ckpt 200 s
853	884	888	917

Figure 7-2 depicts the impact of the failures according to the time in which the failure occur. The bars in the figure represent the execution time in seconds, taller bars indicates larger runtimes.

The advantage of using the RADIC architecture is evident when we compare the effects of failures in the system with and without fault tolerance in all cases. The bars in the columns "no fault" represent the execution times in the absence of failures. We see that the maximum execution time with fault tolerance (917 seconds for a checkpoint interval of 200s) was 7.5% greater of the execution time without fault tolerance (853 seconds). Larger checkpoint intervals generated a lower impact over the execution time in the absence of failures.

However, when failures occurred, the advantage of the fault-tolerant architecture becomes clear. The execution times using RADIC were always inferior to the execution times without fault tolerance. We can see that the best benefits of RADIC architecture appeared when the failure occurs near the end of the application. This occurs because, when the system operates without fault tolerance, if the program fails it wastes all computation done and it must restart since the begging.



	no fault	at 250 s	at 350 s	at 650 s
w/o FT	853	1457	1557	1857
ckpt 200s	917	1082	1079	1099

Figure 7-2: Impact of faults over the execution time in the M/W matrix multiplication program using dynamic load balancing (1000x1000 elements/block)

We repeated the same tests now using blocks of 600x600 elements in order to experiment the system with a different computation-to-communication ratio. Table 7-3 represents the execution times (in seconds) assessed in this experiment. The new communication-to-computation ratio has created a different application behavior because now the checkpoints are lower and the number of message logs increased because there are more messages between processes.

Table 7-3: Execution times in the presence of failures for the M/W matrix multiplication program (dynamic load balance) using 15 nodes (600x600 elements/block)

Fault time	calculated w/o FT (T_{14}=888 s)	ckpt 600 s	ckpt 300 s	ckpt 200 s
fault at 250 s	1138	n/a	n/a	970
fault at 350 s	1238	n/a	957	928
fault at 650 s	1538	831	954	924

Obs.: n/a = not applicable

Execution time (in seconds) using 15 nodes in the absence of failures			
w/o FT	ckpt 600 s	Ckpt 300 s	ckpt 200 s
812	855	853	879

Figure 7-3 compares the total execution times for the experiment. The bars in the columns "no fault" represent the execution times in the absence of failures. The maximum execution time with the fault tolerance (879 seconds for a checkpoint interval of 200 seconds) was 8.3% greater of the execution time without fault tolerance (812 seconds).

Comparing the results obtained for the two experiments with the matrix multiplication using dynamic load balance algorithm, we notice that, when the blocks are smaller, the overheads caused by faults are a little greater than when the blocks are larger. The reason for this is the increment in the number of messages during the application execution.

When we increased the number of messages two symptoms may occur. First, the cost of message log increased. Second, the influence of the checkpoints over the messages increased. The first factor is obvious, more messages generate more message logs and this creates more interference in the application behavior. The second factor is consequence of we have explained previously in this chapter.

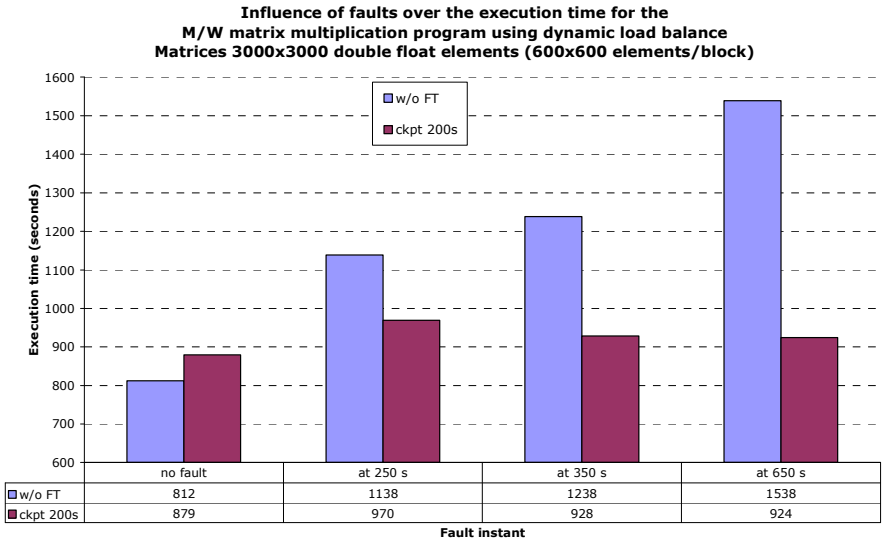


Figure 7-3: Execution times in presence of faults for the master/worker with dynamic load balance using 15 nodes (blocks 600x600)

In this chapter, we evaluated the design parameters of the RADIC fault tolerance architecture. We presented several experiments that allowed us to understand better the practical implications of each parameter over the application performance.

Based on these experiments, we concluded that the watchdog/heartbeat mechanism had an imperceptible influence over the application execution time. We also concluded that the protector's structure did not interfere in the runtime of the application. Nevertheless, we verified that the number of protectors in a chain bounds the number of fault that the system may suffer without crash.

Finally, we assessed the influence of the checkpoint interval together with the application communication-to-computation ratio in different fault scenarios. We concluded that these parameters are the critical factors that determine the impact of the RADIC controller for fault tolerance over the application runtime.

Chapter 8

Conclusions

The current trend for the parallel computers indicates that the size and the complexity of these machines will continue to increase in the near future. In this scenario, users and system administrators will need tools and mechanisms that help them to manage failures transparently, efficiently and with as little influence as possible on the creation and execution of the parallel application.

These large parallel computers require new fault-tolerant architectures that can manage faults transparently and with little interference over the system performance. Adaptability to the system's features (like the system's scalability) and flexibility are also key features that such fault-tolerant architectures must simultaneously attend.

This thesis describes RADIC (Redundant Array of Distributed Independent Fault Tolerance Controllers), the fault tolerance architecture capable to face the current challenges of fault tolerance for the modern parallel computers. RADIC creates a fully distributed fault tolerance controller based on the collaboration of two types of processes: *observers* and *protectors*. Because *protectors* and *observers* share the same resources reserved to the parallel application processes, such controller automatically performs all activities required to implement fault tolerance in a parallel computer, without needing dedicated or fully stable elements.

By putting one protector in each node reserved for the parallel application, RADIC creates a protection chain in which a protector monitors at least one other protector in its neighborhood. Therefore, the system has a fully decentralized fault detector in which a node can detect failures in a neighbor node. The protector's chain is flexible in order to allow that several protectors monitor a node. Therefore, a survivor protector can always detect and manage simultaneous failures that could affect two or more neighbor nodes. The flexibility of the protector's chain makes

possible that the RADIC controller easily adapts to different parallel computer structures.

The observers monitor the processes of the parallel application and communicate with a protector in a neighbor node, in order to save the information required by the rollback-recovery protocol. Since there is one observer attached to each application process, this structure creates a decentralized storage in which there is no central element involved in the operation of the fault-tolerance mechanism.

The decentralized operation of our architecture is one of its strongest features. Because the fault tolerance controller does not have any central element, its operation does not compromise the scalability of the parallel application. Although the scalability suffers the interference of the rollback-recovery protocol, because of the overheads caused by checkpoints and message logs, RADIC minimizes such interference using a fully distributed structure. Our tests showed that increasing the number of nodes in the cluster did not influence the behavior of the fault tolerance mechanism.

The implementation of RADIC in a system requires no modification in the parallel application. The mechanism operates as a layer between the message-passing mechanism and the parallel computer's structure. The parallel application does not have to concern about the operation of the fault tolerance mechanism; however, thanks to the flexibility of the architecture, it is possible to adjust the parameters of the fault-tolerant architecture in order to attend to particular application requirements.

The principles and the functional model of the RADIC architecture appeared for the first time in **ParCo 2005** conference **[Duarte, *et al.*, 2005]**. For the experiments, we developed an implementation based on the MPI standard (RADICMPI).

In the tests, we chose three different parallel applications and compiled their source codes with a legacy MPI implementation (MPICH-1.2.7) and with RADICMPI. The applications generated the same results in all cases (even in the presence of failures) confirming the transparency of the architecture. The test bed was a cluster built with common off-the-shelf IBM-PC machines with no special fault tolerance feature.

After prove that the functionality of the architecture, we redesign RADIC to operate as a layer between the message-passing application and the parallel computer structure. In the **13th EuroPVM/MPI**, we demonstrated the results of the new structure **[Duarte, *et al.*, 2006b]**.

Thanks to the new structure of observers and protectors, we managed to reduce the overheads caused by the operation of the fault tolerance mechanism. In the **2006 IEEE International Conference on Cluster Computing**, we presented the performance results of our new RADICMPI implementation **[Duarte, *et al.*, 2006a]**.

Because the RADIC controller works as a distributed application, sharing the computer's resources with the parallel application processes, we had to design a test procedure in order to assure that the fault tolerance controller worked correctly in all possible fault scenarios. To perform the tests, we implemented a fault injection mechanism in RADICMPI, which created all fault scenarios required to experiment the system. We explained the test procedure and the failure injection mechanism in the **15th Euromicro Conference on Parallel, Distributed and Network-based Processing [Duarte, *et al.*, 2007]**.

The results obtained with different types of parallel applications using RADICMPI shown that that the interference of the fault tolerance operation have not imposed a strong overhead over the execution time in failure free executions. We argue that this is because of the RADIC distributed mechanism, which allows that fault tolerance procedures work concurrently with the parallel application. Nevertheless, it has become clear that the message pattern, and the communication-to-computation ratio, of the parallel application strongly interfere on the overhead caused by the fault tolerance operation.

In the presence of failures, the overhead is strongly dependent of how the parallel application suffers with the unbalancing caused by the reduction in the parallel computer's structure. Failures have low influence over the total execution time of parallel applications that have a dynamic computational load balancing. Conversely, parallel applications in which the load balance is static greatly suffer the influence of failures.

We confirmed this behavior by comparing the matrix-multiplication algorithms in executions with failures. In our tests, the algorithms that used static load balancing (Master/Work with static balancing and Canon) suffered a strong enlargement in the execution time after a failure. Such enlargement occurred because a process recovers in a node that already has another application processes. Because the load balance is rigid, nodes that received recovered processes become slower and the total application execution time enlarged.

On the other hand, failures produced a low impact on the execution time of the master/worker program using dynamic balancing. This algorithm has a natural mechanism for balancing the computational load between the workers, because the master automatically distributes more workload to faster workers. Therefore, when a node becomes slower because it has to execute its original process together with a recovered process, the master will send less work for the processes that are in this node, balancing the workload between the faster nodes.

8.1 Future Works

The RADIC architecture relies on a simple and powerful idea: the complete distribution of the fault tolerance mechanism throughout the nodes of the parallel computer. In this thesis, we explained the architecture and presented a RADIC prototype, namely RADICMPI, which served to test the concepts of the architecture in practice.

There are still many open questions. Some of them relate to theoretical aspects of RADIC, while others refer to the practical issues about RADICMPI. We address such open questions in the next paragraphs.

8.1.1 The future of the RADIC architecture

A major question about the RADIC architecture is how it will operate in large parallel computers. The elements of the fault tolerance controller interact without any global synchronization and are fully distributed throughout the nodes of the parallel computer. The decentralized operation of RADIC *a priori* does not compromise the

scalability of the system. So far, our tests indicated that the number of nodes of the computer does not affect the operation of the fault tolerance controller. However, it is necessary to keep researching about how protectors and observers will operate in larger and complex structures, like hypercubes or crossbar topologies.

We have to develop some strategy to guide the protection mapping for these complex architectures. Because the RADIC controller is a parallel application, the distribution of the protectors must take into consideration the parallel computer structure. The network distance between two neighbor protectors must be as short as possible in order to minimize the effects of the network latency over the heartbeat/watchdog mechanism, and to reduce the traffic of heartbeats throughout the network. For example, processes that have large states or that receive many messages must be placed as near as possible from their protectors in order to reduce the interference of the traffic of checkpoints and message logs over the network.

In order to study the operation of RADIC in such large and complex systems, we have to develop a simulator to help us in the study of the architecture in such systems. The simulator will is an indispensable tool for studying the interaction between RADIC and the system and for studying the influence of the fault tolerance parameters over the application performance.

In order to build the simulator, we are developing an analytical model that describes the RADIC operation in terms of its functional parameters. Such model will relate the parameters of the three major elements of the system: the parallel computer (fault distribution, latency and the bandwidth networks, and disk throughputs) the parallel application (computation-to-communication ratio, message sizes and state sizes,) and RADIC (checkpoint intervals, heartbeat frequency,) in order to describe the impact over the system (time overheads, resources consumed).

One important question is how to adjust the checkpoint in order to reduce the overhead in failure free executions without compromise the fault penalty. The literature about fault tolerance contains several works dedicated to the optimization of the checkpoint interval and we have to verify how to apply this knowledge to RADIC.

Another interesting question is how RADIC would operate using a different rollback-recovery protocol. In practice, any observer-protector or protector-protector relationship may exist in order to implement another rollback-recovery protocol. We created RADIC using the receiver-based pessimistic message log protocol because this is the only protocol in which the recovery process is local. Therefore, this protocol does not compromise the scalability, one of the major features of the RADIC architecture. However, we want to investigate how RADIC will function with protocols like causal protocols or sender-based protocols, in order to establish the advantage and disadvantage of these protocols in our scheme.

We also need to generalize the concept of failures in RADIC by expanding the activities of protectors and observers in order to enlarge the spectrum of failures that the controller may detect. Currentely, the fault tolerance controller bases on communication failures in order to detect faults. It assumes that the only way to establish that an element has failed is when such element stops to communicate. Furthermore, if an element has failed, it is "detached" of the system.

To generalize the fault detection, protectors and observers need to assume new activities as, for example, to create a protocol that manages transient failures. Such protocol could establish that a node is suspicious before of establishing that the node is faulty. Therefore, if the node begins to operate again, the system may reuse it to reestablish the original load balance of the application or as a spare node to recovery the processes that can fail in the future.

8.1.2 The future of RADICMPI

RADICMPI contains a small subset of MPI-1 communication's functions. The development of a full fault-tolerant MPI implementation demands a huge effort that requires a large amount of resources. Therefore, we have started negotiations with the developers of MPICH, namely the Laboratory for Scalable Parallel System Software of the Argonne National Laboratory in Chicago-USA, in order to integrate the RADIC concepts in the MPICH implementation.

Meanwhile, we continue the development of RADICMPI. Currently, we are working in two major lines. First, we are including more communication functions in order to test a larger number of parallel applications. The first set of functions are the nonblocking communication functions, which demands that the log protocol consider a new group of determinants created by the `MPI_Test()` function. At the future, we intend to include the buffered, synchronous and ready modes into the `MPI_Send()` and `MPI_Recv()` functions; and to implement the collective communication functions.

The second line is the utilization of spares in order to mitigate the fault penalty caused by the reduction in the parallel computer's structure after a failure. We know that the cost of a fault depends on how long the RADIC controller takes to detect the fault, how long it takes to recover the process and how much of computation was lost. This cost relates to the parameters of the fault tolerance controller, namely, the heartbeat cycle and the checkpoint interval.

Additionally, the load unbalance caused by the redistribution of the recovered processes is another factor that determines the impact of a failure over the execution runtime. The effects of this factor strongly relates to how the parallel application will react to the new structure of the parallel computer.

Parallel algorithms with dynamic load balance will better adapt to changes in the cluster structure, mitigating the cost of a fault. However, these algorithms typically generate more messages between the processes than their equivalents using static load balance. This behavior increases the impact of the message log and the checkpoint interval over the application execution.

To avoid the impact of the load unbalancing, we have to include spare nodes in the system. Therefore, when a node fails, the process in the faulty node will recover in a spare node. In order to face such challenge, we are currently improving RADICMPI to support spare nodes that can recover faulty processes. We have presented the idea of using spares nodes at **CACIC'2006** congress **[Santos, *et al.*, 2006]**. The first results were promising and we believe that this strategy is crucial to satisfy the requirements of time-critical applications. Therefore, we kept the development of the

idea and obtained new results that are going to appear in the **ParCo2007** congress **[Santos*, et al.*, 2007]**.

Additionally, in the near future we can make two improvements in RADICMPI for mitigating the impact of checkpoints and message logs over the application runtime: to implement a quasi-synchronous checkpoint protocol based on communication failures caused by checkpoints.

In a quasi-synchronous protocol, whenever a sender process gets a communication error with a receiver process, the sender would evaluate if the error occurred because the receiver was in a checkpoint procedure. If the receiver is making a checkpoint, then the sender may decide to make a checkpoint also, instead of just waiting for the receiver is available again. Therefore, the sender uses the time that it would be waiting for the receiver to take its own checkpoint, reducing the impact of the checkpoints over the execution.

Another improvement is to use multicast communications in order to reduce the message log cost. In such scheme, the sender observer should simultaneously send the message to the receiver observer and to the protector of the receiver observer, reducing the time of the log procedure. We believe that such multicast communications will be decisive in order to reduce the cost of fault tolerance for the MPI collective communications.

The RADIC architecture debuted in this thesis and has a large future ahead. As we have shown in the last paragraphs, we are far from complete exploring all theoretical and practical aspects of the architecture. So far, RADIC has demonstrated its great potential as a fault-tolerant architecture for message-passing systems. Nevertheless, we are aware that many aspects need more investigation and that additional works must complement the description we have made in this thesis. In the future, our goal is to establish RADIC as a general fault-tolerant platform for the current and future message-passing implementations. We believe that this work has contributed to achieve this goal.

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, *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.

[Anderson and Lee, 1981] - Anderson, T. and Lee, P. A. *Fault Tolerance: Principles and Practice*. 1st. ed. Englewood Cliffs, USA: Prentice Hall, 1981.

[Argonne National Laboratory, 2007a] - Argonne National Laboratory. *The Message Passing Interface (MPI) standard*. Available at http://www-unix.mcs.anl.gov/mpi/index.htm. Accessed in February, 2007. Chicago, USA. 2007a.

[Argonne National Laboratory, 2007b] - Argonne National Laboratory. *MPICH-A Portable Implementation of MPI*. Available at http://www-unix.mcs.anl.gov/mpi/mpich1/. Accessed in February, 2007. Chicago, USA. 2007b.

[Argonne National Laboratory, 2007c] - Argonne National Laboratory. *MPICH2*. Available at http://www-unix.mcs.anl.gov/mpi/mpich2/. Accessed in March, 2007. Chicago, USA. 2007c.

[Avizienis, *et al.*, 2004] - Avizienis, A., Laprie, J. C., Randell, B. and Landwehr, C. *Basic concepts and taxonomy of dependable and secure computing*. IEEE Transactions on Dependable and Secure Computing, 1(1):11-33 (January 2004)

[Avresky, *et al.*, 1992] - Avresky, D., Arlat, J., Laprie, J. C. and Crouzet, Y. *Fault injection for the formal testing of fault tolerance*. In Proceedings of The 22nd International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 345-354. July 8-10, 1992. IEEE Press.

[Baldoni, *et al.*, 1998] - Baldoni, R., Quaglia, F. and Ciciani, B. *A VP-accordant checkpointing protocol preventing useless checkpoints*. In Proceedings of The 17th IEEE Symposium on Reliable Distributed Systems, pp. 61-67. West Lafayette, USA. october 20-23, 1998. IEEE Press.

[Batchu, *et al.*, 2001] - Batchu, R., Neelamegam, J. P., Zhenqian, C., Beddhu, M., Skjellum, A., Dandass, Y. and Apte, M. *MPI/FT™: Architecture and Taxonomies for Fault-tolerant, Message-passing Middleware for Performance-portable Parallel Computing*. In Proceedings of The 1st. IEEE/ACM International Symposium on

Cluster Computing and Grid, pp. 26-33. Starkville, USA. May 15-18, 2001. IEEE Press.

[Bhargava and Shu-Renn, 1988] - Bhargava, B. and Shu-Renn, L. *Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach*. In Proceedings of The 17th Symposium on Reliable Distributed Systems, pp. 3-12. Columbus, USA. October 10-12, 1988.

[Borg, *et al.*, 1989] - Borg, A., Blau, W., Graetsch, W., Herrmann, F. and Oberle, W. *Fault tolerance under UNIX*. ACM Transactions on Computer Systems, 7(1):1-24 (January 1989)

[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 Proceedings of ACM/IEEE 2002 Supercomputing Conference (SC'2002), pp. 29-29. Baltimore, USA. November, 16-22, 2002. IEEE Press.

[Bouteiller, *et al.*, 2003] - 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)

[Coti, *et al.*, 2006] - Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E. and Cappello, F. *Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI*. In Proceedings of The ACM/IEEE SC 2006 Supercomputing Conference (SC '06), pp. Tampa, USA. November, 11-17, 2006. IEEE Computer Society.

[Cristian and Jahanian, 1991] - Cristian, F. and Jahanian, F. *A timestamp-based checkpointing protocol for long-lived distributed computations*. In Proceedings of The 10th Symposium on Reliable Distributed Systems, pp. 12-20. San Jose, USA. September 30 - October 2, 1991.

[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)

[Chandy, 1972] - Chandy, M. a. R., C. . *Rollback and recovery strategies for computer programs*. IEEE Transactions on Computers, 21(6):546–556 (1972)

[Daly, 2006] - Daly, J. T. *A higher order estimate of the optimum checkpoint interval for restart dumps*. Future Generation Computer Systems, 22(3):303-312 (February 2006)

[Damani, *et al.*, 2003] - Damani, O. P., Wang, Y.-M. and Garg, V. K. *Distributed recovery with K-optimistic logging*. Journal of Parallel and Distributed Computing 63(12):1193-1218 (December 2003)

[Duarte, *et al.*, 2005] - Duarte, A., Rexachs, D. and Luque, E. *A Distributed Scheme for Fault-Tolerance in Large Cluster of Workstations*. In Proceedings of The International Conference Parco2005, pp. 473-480. Malaga, Spain. September 13-16, 2005. NIC Directors.

[Duarte, *et al.*, 2006a] - Duarte, A., Rexachs, D. and Luque, E. *Increasing the cluster availability using RADIC*. In Proceedings of 2006 IEEE International Conference on Cluster Computing, pp. 1-8. Barcelona, Spain. September 25-28, 2006a. IEEE Computer Society.

[Duarte, *et al.*, 2006b] - Duarte, A., Rexachs, D. and Luque, E. *An Intelligent Management of Fault Tolerance in Clusters 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.

[Duda, 1983] - Duda, A. *The effects of checkpointing on program execution time*. Information Processing Letters, 16(5):221-229 (June 10 1983)

[Elnozahy, *et al.*, 1992] - Elnozahy, E. N., Johnson, D. B. and Zwaenepoel, W. *The performance of consistent checkpointing*. In Proceedings of The 11th Symposium on Reliable Distributed Systems, pp. 39-47. Houston, USA. October 5-7, 1992.

[Elnozahy and Zwaenepoel, 1992] - Elnozahy, E. N. and Zwaenepoel, W. *Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit*. IEEE Transactions on Computers, 41(5):526-531 (May 1992)

[Elnozahy and Zwaenepoel, 1994] - Elnozahy, E. N. and Zwaenepoel, W. *On the use and implementation of message logging*. In Proceedings of The 24th International Symposium on Fault-Tolerant Computing (FTCS-24), pp. 298-307. Austin, USA. June 15-17, 1994. IEEE Press.

[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)

[Fagg and Dongarra, 2000] - Fagg, G. E. and Dongarra, J. *FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world*. In Proceedings of The 7th European PVM/MPI Users' Group Meeting, pp. 346-353. Balatonfüred, Hungary. September 10-13, 2000. Springer.

[Fagg, *et al.*, 2005] - Fagg, G. E., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J. and Dongarra, J. *Scalable Fault Tolerant MPI: Extending the Recovery Algorithm*. In Proceedings of The 12th European PVM/MPI Users' Group Meeting, pp. 67-75. Sorrento, Italy. September 18-21, 2005. Springer.

[Gabriel, *et al.*, 2004] - Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L. and Woodall, T. S. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implemention*. In Proceedings of The 11th European PVM/MPI Users' Group Meeting, pp. 97-104. Budapest, Hungary. September, 19-22, 2004. Springer.

[Geist, 2002] - Geist, A. *Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors* Available at http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf. Accessed in March 2, 2007. Oak Ridge, USA. 2002. Oak Ridge National Laboratory.

[Gelenbe, 1979] - Gelenbe, E. *On the optimum checkpoint interval*. Journal of the Association for Computing Machinery, 26(2):259-270 (April 1979)

[Goldberg, 1990] - Goldberg, A. *Transparent recovery of Mach applications*. In Proceedings of Mach Workshop Conference, pp. 169-184. Burlington, USA. October 4-5, 1990. USENIX.

[Graham, *et al.*, 2003] - Graham, R. L., Choi, S.-E., Daniel, D. J., Desai, N. N., Minnich, R. G., Rasmussen, C. E., Risinger, L. D. and Sukalski, M. W. *A network-failure-tolerant message-passing system for terascale clusters*. International Journal of Parallel Programming, 31(4):285-303 (August 2003)

[Gropp and Lusk, 2004] - Gropp, W. and Lusk, E. *Fault Tolerance in Message Passing Interface Programs*. International Journal of High Performance Computing Applications, 18(3):363-372 (August 2004)

[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.

[Helary, *et al.*, 1997a] - Helary, J.-M., Mostefaoui, A. and Raynal, M. *Virtual precedence in asynchronous systems: concepts and applications*. In Proceedings of The 11th Workshop on Distributed Algorithms (WDAG'97), pp. 170-194. Saarbrücken, Germany. September 24-26, 1997a. Springer.

[Helary, *et al.*, 1997b] - Helary, J. M., Mostefaoui, A., Netzer, R. H. B. and Raynal, M. *Preventing useless checkpoints in distributed computations*. In Proceedings of The 16th Symposium on Reliable Distributed Systems, pp. 183-190. Durham, UK. October, 22-24, 1997b. IEEE Press.

[Hoarau and Tixeuil, 2005] - Hoarau, W. and Tixeuil, S. *A language-driven tool for fault injection in distributed systems*. In Proceedings of The 6th IEEE/ACM International Workshop on Grid Computing, pp. 8 pp. Seattle, USA. November 13-14, 2005.

[Hoarau, *et al.*, 2006] - Hoarau, W., Lemarinier, P., Herault, T., Rodriguez, E., Tixeuil, S. and Cappello, F. *FAIL-MPI: How Fault-Tolerant Is Fault-Tolerant MPI?* In Proceedings of The 2006 IEEE International Conference on Cluster Computing, pp. 1-10. Barcelona, Spain. September 25-28, 2006.

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

[Johnson and Zwaenepoel, 1987] - Johnson, D. B. and Zwaenepoel, W. *Sender-Based Message Logging*. In Proceedings of The 17th Fault-Tolerant Computing Symposium, pp. 14-19. Pittsburgh, USA. July, 1987. IEEE Computer Society.

[Juang and Venkatesan, 1991] - Juang, T. T. Y. and Venkatesan, S. *Crash recovery with little overhead*. In Proceedings of The 11th International Conference on Distributed Computing Systems, pp. 454-461. Arlington, USA. May 20-24, 1991. IEEE Computer Society.

[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)

[Koo and Toueg, 1987] - Koo, R. and Toueg, S. *Checkpointing and Rollback-Recovery for Distributed Systems*. IEEE Transactions on Software Engineering, SE-13(1):23-31 (January 1987)

[Kopetz, 1997] - Kopetz, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. The Kluwer International Series in Engineering and Computer Science 1st. ed. Dordrecht, The Netherlands: Kluwer Academic Publisher Group, 1997.

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

[Lamport, *et al.*, 1982] - Lamport, L., Shostak, R. and Pease, M. *The Byzantine Generals Problem*. ACM Transactions on Programming Languages and Systems (TOPLAS), 4(3):382-401 (July 1982)

[Litzkow, *et al.*, 1988] - Litzkow, M. J., Livny, M. and Mutka, M. W. *Condor-a hunter of idle workstations*. In Proceedings of The 8th International Conference on Distributed Computing Systems, pp. 104-111. San Jose, USA. June 13-17, 1988. IEEE Computer Society Press.

[Mandal and Mukhopadhyaya, 2003] - Mandal, P. S. and Mukhopadhyaya, K. *Estimating Checkpointing, Rollback and Recovery Overheads*. In Proceedings of The 5th International Workshop on Distributed Computing - IWDC 2003, pp. Kolkata, India. December 27-30, 2003. Springer.

[Marcus and Stern, 2003] - Marcus, E. and Stern, H. *Blueprints for High Availability*. 2nd ed. Indianapolis, USA: Wiley Publishing, Inc., 2003.

[Micah, *et al.*, 1999] - Micah, B., Jack, J. D., Graham, E. F., Geist, G. A., Paul, G., James, K., Mauro, M., Keith, M., Terry, M., Philip, P., Stephen, L. S. and Vaidy, S. *Harness: a next generation distributed virtual machine*. Future Generation Computer Systems - Special issue on metacomputing, 15:571-582. Elsevier Science Publishers B. V., 1999.

[MPICH, 2007] - MPICH, A. N. L. *MPICH-A Portable Implementation of MPI*. Available at http://www-unix.mcs.anl.gov/mpi/mpich1/. Accessed in February, 2007. Chicago, USA. 2007. Argonne National Laboratory.

[Netzer and Jian, 1995] - Netzer, R. H. B. and Jian, X. *Necessary and sufficient conditions for consistent global snapshots*. IEEE Transactions on Parallel and Distributed Systems, 6(2):165-169 (February 1995)

[Oak Ridge National Laboratory, 2007] - Oak Ridge National Laboratory. *PVM: Parallel Virtual Machine*. Available at http://www.csm.ornl.gov/pvm. Accessed in April, 2007. Oak Ridge, USA. 2007. Computer Science and Mathematics.

[Open-MPI Project, 2007] - Open-MPI Project. *Open MPI: Open Source High Performance Computing*. Available at http://www.open-mpi.org/. Accessed in March, 2007. Bloomington, USA. 2007.

[Patterson, *et al.*, 1988] - Patterson, D. A., Gibson, G. A. and Katz, R. H. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*. In Proceedings of 1988 ACM SIGMOD International Conference on Management of Data, pp. 109-116. Chicago, USA. June 1-3, 1988. ACM Press.

[Plank, *et al.*, 1995] - Plank, J. S., Beck, M., Kingsley, G. and Li, K. *Libckpt: Transparent Checkpointing under UNIX*. In Proceedings of USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems, pp. New Orleans, USA. January 16-20, 1995. USENIX Association.

[Plank and Thomason, 2001] - Plank, J. S. and Thomason, M. G. *Processor Allocation and Checkpoint Interval Selection in Cluster Computing Systems*. Journal of Parallel and Distributed Computing,, 61(11):1570-1590 (November 2001)

[Randell, 1999] - Randell, B. *Fault tolerance in decentralized systems*. In Proceedings of The 14th International Symposium on Autonomous Decentralized Systems (ISADS'99), pp. 174-179. Tokyo, Japan. March 21-23, 1999. IEEE Computer Society.

[Rao*, et al.*, 1999] - Rao, S., Alvisi, L. and Vin, H. M. *Egida: an extensible toolkit for low-overhead fault-tolerance*. In Proceedings of The 29th Annual International Symposium on Fault-Tolerant Computing, pp. 48-55. Madison, USA. June 15-18, 1999.

[Russell, 1980] - Russell, D. L. *State Restoration in Systems of Communicating Processes*. IEEE Transactions on Software Engineering, SE-6(2):183-194 (March 1980)

[Sankaran*, et al.*, 2005] - Sankaran, S., Squyres, J. M., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J. and Hargrove, P. *The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing*. International Journal of High Performance Computing Applications, 19(4):479-493 2005)

[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.

[Sistla and Welch, 1989] - Sistla, A. P. and Welch, J. L. *Efficient distributed recovery using message logging*. In Proceedings of The 8th Annual ACM Symposium on Principles of Distributed Computing (PODC'89), pp. 223-238. Edmonton, Canada. August 14-16, 1989. ACM Press.

[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)

[Tamir and Sequin, 1984] - Tamir, Y. and Sequin, C. H. *Error Recovery in Multicomputers Using Global Checkpoints*. In Proceedings of The 13th International Conference on Parallel Processing, pp. 32-41. Bellaire, USA. August, 1984.

[Tixeuil, *et al.*, 2006] - Tixeuil, S., Hoarau, W. and Silva, L. *An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids*. The 2nd CoreGRID Workshop on GRID and Peer to Peer Systems Architecture. Paris, France. January 16-17, 2006. Institute on System Architecture, CoreGRID - Network of Excellence.

[Tong, *et al.*, 1992] - Tong, Z., Kain, R. Y. and Tsai, W. T. *Rollback recovery in distributed systems using loosely synchronized clocks*. IEEE Transactions on Parallel and Distributed Systems, 3(2):246-251 (March 1992)

[Top500.Org, 2006] - Top500.Org. *TOP500 Supercomputers Sites*. Available at http://www.top500.org/. Accessed in February, 2007. Celle, Germany. 2006. Prometeus GmBH.

[Wang, 1997] - Wang, Y.-M. *Consistent global checkpoints that contain a given set of local checkpoints*. IEEE Transactions on Computers, 46(4):456-468 (April 1997)

[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.

[Ziv and Bruck, 1996] - Ziv, A. and Bruck, J. *An on-line algorithm for checkpoint placement*. In Proceedings of The 7th International Symposium on Software Reliability Engineering, pp. 274-283. White Plains, USA. October 30 - November 2, 1996. IEEE Press.