



Universitat Autònoma
de Barcelona

Departament d'Arquitectura de
Computadors i Sistemes Operatius
Màster en Computació d'Altes Prestacions

Applying RADIC in Open MPI

The methodology used to implement RADIC over a
Message Passing Library

Master thesis submitted by
Leonardo Fialho directed by
Dolores Rexachs in fulfillment of
requirements for Master Degree
at Universitat Autònoma de
Barcelona

2008

Trabajo de investigación

Máster en Computación de Altas Prestaciones

Curso 2007-08

Título Applying RADIC in Open MPI

Autor: Leonardo Fialho de Queiroz

Director: Dolores Isabel Rexachs del Rosario

Departamento Arquitectura de Computadores y Sistemas Operativos

Escuela Técnica Superior de Ingeniería (ETSE)

Universidad Autónoma de Barcelona

Firmado

Autor

Leonardo Fialho

Director

Dolores Rexachs

For myself.

Acknowledgements

One year ago I left my country in order to achieve a Ph.D. In the middle of this journey I am finishing the master degree, and more than this degree I have found a new life and friends.

Of course, there are lots of people whom I need to thank. Starting from the principle, thank to my perfect family which has supported me in all my decisions. Terezinha, Eunápio, Erick and Diana, I have no words to describe how perfect you are.

It is amazing to imagine that a guy, whom does not like to study, is crossing this long and painful journey. The responsible for it is Osvaldo Requião. For me Osvaldo will be always an example of teacher and person. He has planted the research seed in my head while I was doing my bachelor degree.

A special thank to Genaro Costa whom have asked me if I want to study for Ph.D. Thanks to Eduardo Argollo who have explained and helped me to follow all the steps necessary to arrive in the Universitat Autònoma de Barcelona.

Here in Barcelona I had all support necessary to start my new life and this work. Thus, thanks to Guna Alexander who gave me more than concepts and knowledge, he gave me his friendship. And thank you Angelo Duarte, even distant you gave me your help.

However, this work only exists because I have two “kindly” and special tutors: Dolores Rexachs and Emilio Luque. I think that, after one year, the

seed planted by Osvaldo is start blowing. Lola and Emilio, you gave me, and are still giving, all “nutrients” necessary to develop my work here.

Thanks too for all other people who are part of the “CAOS” department. Thanks to all professors and, in special, two guys: Daniel Ruiz and Javier Navarro.

Finally, thanks to the Catalan girl who makes my life, here in Barcelona, be better. She gave to me more than comprehension during this painful job: Sònia Ferré, thank you to be who you are.

Leonardo Fialho

Bellaterra, July 2008

Abstract

Fault tolerance has become a major issue for computer and software engineers because the occurrence of faults increases the cost of using a parallel computer. RADIC is the fault tolerance architecture for message passing systems which is transparent, decentralized, flexible and scalable.

This master thesis presents the methodology used to implement the RADIC architecture over Open MPI, a well-know large-used message passing library. This implementation kept the RADIC architecture characteristics.

In order to validate the implementation we have executed a synthetic ping program, besides, to evaluate the implementation performance we have used the NAS Parallel Benchmarks.

The results prove that the RADIC architecture performance depends on the communication pattern of the parallel application which is running. Furthermore, our implementation proves that the RADIC architecture could be implemented over an existent message passing library.

Resum

La tolerància a fallades s'ha convertit en un requeriment important pels enginyers informàtics i els desenvolupadors de programari, degut a que l'aparició de fallades augmenta el cost d'explotació d'un ordinador paral·lel. RADIC és una arquitectura de tolerància a fallades per sistemes de pas de missatges transparent, descentralitzada, flexible i escalable.

Aquest treball d'investigació de final de màster presenta la metodologia utilitzada per implementar l'arquitectura RADIC per a Open MPI, una llibreria ben coneguda i molt utilitzada de pas de missatges. Aquesta implementació manté les característiques originals de RADIC.

Per validar la implementació executem una aplicació sintètica de ping. A més, per avaluar les prestacions de la implementació utilitzem els benchmarks paral·lels NAS.

Els resultats han provat que les prestacions de l'arquitectura RADIC depenen de l'aplicació que s'estigui executant. A més, la nostra implementació prova que l'arquitectura RADIC pot ser implentada en una llibreria de pas de missatges ja existent.

Resumen

La tolerancia a fallos se ha convertido en un requisito importante para los ingenieros informáticos y los desarrolladores de software, debido a que la aparición de fallos aumenta el coste de explotación de un ordenador paralelo. RADIC es una arquitectura tolerante a fallos para sistemas de paso de mensajes transparente, descentralizada, flexible y escalable.

Este trabajo de investigación de máster presenta la metodología utilizada para implementar la arquitectura RADIC en Open MPI, una librería bien conocida y muy utilizada de paso de mensajes. Esta implementación mantiene las características originales de RADIC.

Para validar la implementación ejecutamos una aplicación sintética de ping, por otro lado, para evaluar las prestaciones de la implementación usamos los “benchmarks” paralelos NAS.

Los resultados muestran como las prestaciones de la arquitectura RADIC dependen de la aplicación que se esté ejecutando y su patrón de comunicación. Además, nuestra implementación prueba que la arquitectura RADIC puede ser implementada en una librería de paso de mensajes ya existente.

Resumo

Tolerância a falhas tornou-se uma questão importante para os engenheiros de computadores e analistas de sistemas devido ao fato de que a ocorrência de falhas aumenta o custo de utilização de computadores paralelos. RADIC é uma arquitetura transparente, descentralizada, flexível e escalável de tolerância a falhas para sistemas de passo de mensagens.

Este documento apresenta a metodologia utilizada para aplicar RADIC em Open MPI, uma biblioteca de passo de mensagens conhecida e largamente utilizada. O processo de codificação manteve as características originais da arquitetura RADIC.

A fim de validar nossa codificação, executamos um programa sintético de ping, e para avaliar o desempenho utilizamos os benchmarks paralelos NAS.

Os resultados provam que o desempenho da arquitetura RADIC depende do padrão de comunicação da aplicação paralela utilizada. Ademais, prova que a arquitetura RADIC pode ser implementada sobre uma biblioteca de passo de mensagens existente.

Table of Contents

CHAPTER 1 INTRODUCTION	27
1.1 GOALS	28
1.2 ORGANIZATION OF THIS DISSERTATION	29
CHAPTER 2 THE RADIC ARCHITECTURE.....	31
2.1 RADIC ARCHITECTURE MODEL	32
2.2 RADIC FUNCTIONAL ELEMENTS	32
2.2.1 Protector	33
2.2.2 Observer.....	34
2.2.3 RADIC Controller for Fault Tolerance.....	34
2.3 RADIC OPERATION	36
2.3.1 Message Passing Mechanism.....	36
2.3.2 State Saving Task	37
2.3.3 Failure Detection Task.....	40
2.3.4 Recovery Task.....	43
2.3.5 Fault Masking Task.....	45
2.4 RADIC FLEXIBILITY	48
2.4.1 Concurrent Failures.....	49
2.4.2 Structural Flexibility.....	50
CHAPTER 3 MESSAGE PASSING LIBRARY IMPLEMENTATIONS.....	53
3.1 RADICMPI.....	53
3.1.1 Observer and Message Passing Interface	55
3.1.2 Protector	61
3.2 OPEN MPI	63
3.2.1 Software Architecture	64
3.2.2 Modular Component Architecture Frameworks	67
3.2.3 Open Run-Time Environment Details	71
3.2.4 Communication Layers	73
3.2.5 Fault Tolerance Operation	75
3.3 FAULT TOLERANCE MESSAGE PASSING IMPLEMENTATIONS	79
3.3.1 MPICH Based Implementations.....	80
3.3.2 Open MPI Based Implementations.....	81
3.3.3 Others Implementations	83
3.3.4 Comparing Fault Tolerant Message Passing Implementations	84
CHAPTER 4 ANALYSIS AND DESIGN OF RADIC OVER OPEN MPI.....	87
4.1 ANALYZING RADIC.....	88

4.1.1 <i>Observer Functionalities</i>	89
4.1.2 <i>Protector Functionalities</i>	90
4.2 CHOOSING A MESSAGE PASSING LIBRARY	92
4.2.1 <i>Software Architecture and Code Structure</i>	93
4.2.2 <i>Communication Layers</i>	94
4.3 ANALYZING OPEN MPI	95
4.3.1 <i>Open MPI Operation</i>	95
4.3.2 <i>Communication Layers</i>	98
4.4 MERGING TWO ARCHITECTURES.....	100
4.4.1 <i>Methodology</i>	101
4.4.2 <i>Observer inside Communication Message Layers</i>	103
4.4.3 <i>Protector inside Open Run-Time Environment Daemon</i>	105
CHAPTER 5 IMPLEMENTATION DETAILS	109
5.1 OBSERVER FUNCTIONALITIES	109
5.1.1 <i>observer Component on PML Framework</i>	110
5.1.2 <i>single Component on SnapC Framework</i>	114
5.1.3 <i>uncoord Component on CRCP Framework</i>	115
5.2 PROTECTOR FUNCTIONALITIES.....	116
5.2.1 <i>Protector Daemon</i>	116
CHAPTER 6 VALIDATING AND EXPERIMENTING THE	
IMPLEMENTATION	123
6.1 NAS PARALLEL BENCHMARK	123
6.2 EXPERIMENTS PLAN AND ENVIRONMENT	127
6.3 VALIDATION EXPERIMENTS	129
6.4 EVALUATION EXPERIMENTS.....	141
6.4.1 <i>Evaluation According Communication Pattern</i>	141
6.4.2 <i>Evaluation According Process Size</i>	152
CHAPTER 7 CONCLUSIONS.....	159
7.1 FUTURE WORKS	161
APPENDIX I.....	163
APPENDIX II	185
APPENDIX III.....	189
REFERENCES	191

List of Figures

FIGURE 2-1: RADIC LEVELS IN A PARALLEL SYSTEM.....	32
FIGURE 2-2: AN EXAMPLE OF PROTECTOR IN A CLUSTER WITH N NODES. RED ARROWS INDICATE THE RELATIONSHIP EXISTENT BETWEEN THEM.....	33
FIGURE 2-3: A CLUSTER USING THE RADIC ARCHITECTURE, A_N ARE APPLICATION PROCESSES, O_N ARE OBSERVERS AND P_N ARE PROTECTORS. GREEN ARROWS REPRESENT THE RELATIONSHIP BETWEEN OBSERVERS AND PROTECTOR AND RED ARROWS THE RELATIONSHIP BETWEEN PROTECTORS.	35
FIGURE 2-4: MESSAGE PASSING MECHANISM IN RADIC.	37
FIGURE 2-5: RELATION BETWEEN OBSERVER AND ITS PROTECTOR.	38
FIGURE 2-6: MESSAGE DELIVERY AND MESSAGE LOG MECHANISMS.	39
FIGURE 2-7: THREE PROTECTORS AND THEIR RELATIONSHIP TO DETECT FAILURES. W IS A WATCHDOG PART OF A PROTECTOR AND H IS A HEARTBEAT SEND PART.	40
FIGURE 2-8: PROTECTOR ALGORITHMS FOR HEARTBEAT AND WATCHDOG TASKS.	41
FIGURE 2-9: RECOVERING TASKS IN A 6-NODE CLUSTER. (A) FAILURE FREE CLUSTER. (B) FAILURE IN NODE N_3 . (C) PROTECTORS P_2 AND P_4 DETECT THE FAILURE AND REESTABLISH THE PROTECTION. (D) P_2 RECOVERS A_3/O_3	44
FIGURE 2-10: FAULT DETECTION ALGORITHMS FOR SENDER AND RECEIVER OBSERVERS.	46
FIGURE 2-11: A 6-NODE CLUSTER USING TWO PROTECTOR CHAINS.	49
FIGURE 2-12: MINIMUM STRUCTURE FOR A PROTECTOR CHAIN. AFTER ONE FAULT THIS CLUSTER WILL NOT BE FAULT TOLERANT ANYMORE.	51
FIGURE 3-1: RADICMPI SOFTWARE LAYERS.....	54
FIGURE 3-2: TYPICAL APPLICATION COMPILED WITH RADICMPI: THREAD AND THEIR RELATIONSHIPS.	56
FIGURE 3-3: OBSERVER ENGINE WITH INTERNAL AND EXTERNAL EVENTS.	57
FIGURE 3-4: MESSAGE LOG PROCEDURE.....	58
FIGURE 3-5: CHECKPOINT PROCEDURE.	59
FIGURE 3-6: MODULAR COMPONENT ARCHITECTURE (MCA) TOP-LEVEL VIEW.	64
FIGURE 3-7: THE THREE MAIN COMPONENT CLASSES IN OPEN MPI.....	65
FIGURE 3-8: OPEN MPI FRAMEWORK CLASS DEPENDENCY RELATION.....	65
FIGURE 3-9: UNIVERSE: OPEN MPI PARALLEL ENVIRONMENT.	72
FIGURE 3-10: POINT-TO-POINT FRAMEWORK ARCHITECTURE IN OPEN MPI.....	73
FIGURE 3-11: OPEN MPI PARTICIPATING IN A DISTRIBUTED CHECKPOINT. 3D BOXES REPRESENT NODES CONTAINING WHITE APPLICATION PROCESSES. ROUNDED BOXES REPRESENT PROCESSES.	76
FIGURE 3-12: OPEN MPI HANDLING A CHECKPOINT REQUEST.....	77

FIGURE 4-1: RADIC LAYERS.....	88
FIGURE 4-2: MPIRUN LAUNCHES ORTED PROCESSES ON THE COMPUTATION NODES.	96
FIGURE 4-3: (A) MPIRUN AS PART OF THE COMPUTATION NODES OR NOT (B).	97
FIGURE 4-4: COMMUNICATION BETWEEN ORTE DAEMONS AND BETWEEN ORTE DAEMON AND APPLICATION PROCESS. BLACK ARROWS ARE RML COMMUNICATION AND RED ARROWS MPI COMMUNICATIONS.	98
FIGURE 4-5: CALL TRACING IN OPEN MPI COMMUNICATIONS LAYERS.	99
FIGURE 4-6: SIMILARITY EXISTENT BETWEEN A RADIC AND AN OPEN MPI NODE. ..	102
FIGURE 5-1: TYPICAL FUNCTION CALL STACK FOR RADIC OVER OPEN MPI.	110
FIGURE 5-2: LOGGING OPERATION PERFORMED BY THE OBSERVER PML WRAPPER. GREEN ARROWS INDICATE MPI COMMUNICATION. RED ARROWS INDICATE RML COMMUNICATION.....	112
FIGURE 6-1: INTERNAL PROCESS IDENTIFICATION EXAMPLE.	129
FIGURE 6-2: DEBUG LOG OF A PING PROGRAM SHOWING THE MESSAGE LOGGING MECHANISM.	130
FIGURE 6-3: RADIC.LOG PRINTED BY LOGREADER.	131
FIGURE 6-4: REPORT GENERATED BY THE NAS BT APPLICATION.	132
FIGURE 6-5: SUMMARIZED OUTPUT GENERATED BY THE LOGREADER.	132
FIGURE 6-6: DEBUG LOG REPRESENTING A CHECKPOINTING PROCEDURE.....	134
FIGURE 6-7: FAILURES WHICH START THE PROCEDURE FOR APPLICATION STATUS VERIFICATION ON NODE N.	135
FIGURE 6-8: EXECUTION LOG OF A NORMAL HEARTBEAT WATCHDOG OPERATION.	136
FIGURE 6-9: DEBUG LOG REPRESENTING THE STATUS VERIFICATION PROCEDURE.....	137
FIGURE 6-10: DEBUG LOG OF A PROTECTOR DAEMON REPRESENTING A RECOVERY REQUEST FROM AN APPLICATION PROCESS.....	140
FIGURE 6-11: DEBUG LOG OF A PROTECTOR DAEMON RECOVERING AN APPLICATION PROCESS.....	140
FIGURE 6-12: EXECUTION OF NAS BT BENCHMARK APPLICATION CLASS A, B AND C IN 4, 9, 16 AND 25 NODES.	143
FIGURE 6-13: EXECUTION OF NAS LU BENCHMARK APPLICATION CLASS A, B AND C IN 4, 8, 16 AND 32 NODES.	145
FIGURE 6-14: EXECUTION OF NAS SP BENCHMARK APPLICATION CLASS A, B AND C IN 4, 9, 16 AND 25 NODES.	146
FIGURE 6-15: EXECUTION OF NAS IS BENCHMARK APPLICATION CLASS A, B AND C IN 4, 8, 16 AND 32 NODES.	148
FIGURE 6-16: EXECUTION OF NAS FT BENCHMARK KERNEL CLASS A AND B IN 4, 8, 16 AND 32 NODES.	149
FIGURE 6-17: EXECUTION OF NAS CG BENCHMARK KERNEL CLASS A, B AND C IN 4, 8, 16 AND 32 NODES.....	150
FIGURE 6-18: EXECUTION OF NAS CG BENCHMARK KERNEL CLASS A, B AND C IN 4, 8, 16 AND 32 NODES.....	151

FIGURE 6-19: TIME SPENT TO TAKE A CHECKPOINT AND RECOVER A PROCESS ACCORDING PROCESS SIZE.	153
FIGURE 6-20: TIME SPENT TO SEND A CHECKPOINT ACCORDING PROCESS SIZE.....	154
FIGURE 6-21: LINEAR TENDENCY OF TRANSFER TIME FOR SMALL CHECKPOINT FILES.	155
FIGURE 6-22: COMPARISON BETWEEN THE CHECKPOINT TRANSFER WITH (BLUE LINE) AND WITHOUT (RED LINE) MPI COMMUNICATION.....	157
FIGURE A-1: A MESSAGE PASSING SYSTEM WITH FOUR PROCESSES EXCHANGING MESSAGES.....	164
FIGURE A-2: EXAMPLES OF INCONSISTENT GLOBAL SYSTEM STATE, (A).CAUSED BY A LOST MESSAGE AND (B) BY AN ORPHAN MESSAGE.	169
FIGURE A-3: DOMINO EFFECT.	171
FIGURE A-4: DIFFERENT CHECKPOINTING APPROACHES.....	173
FIGURE A-5: ABOUT P_0 , DETERMINISTIC EVENTS IN BLUE AND NON-DETERMINISTIC EVENTS IN RED.	179

List of Tables

TABLE 2-1: RADIC KEY FEATURES.....	31
TABLE 2-2: A <i>RADICTABLE</i> EXAMPLE IN A CLUSTER WITH N NODES.....	36
TABLE 2-3: RECOVERY ACTIVITIES PERFORMED BY THE EACH ENTITY IMPLICATED IN A PROCESS FAILURE.	43
TABLE 2-4: PART OF THE UPDATED <i>RADICTABLE</i> OF A PROCESS THAT HAS TRIED TO COMMUNICATE WITH A_3 AFTER IT HAS RECOVERED AS SHOWN IN FIGURE 2-9....	47
TABLE 2-5: <i>RADICTABLE</i> OF AN OBSERVER LOCATED IN A CLUSTER PROTECTED BY TWO PROTECTOR CHAIN, LIKE IN FIGURE 2-11.	50
TABLE 3-1: HOW RADICMPI SATISFIES RADIC ARCHITECTURE FEATURES.	55
TABLE 3-2: RADICMPI COMMUNICATION ROUTINES.	56
TABLE 3-3: FAULT TOLERANT MPI IMPLEMENTATION COMPARISON.	85
TABLE 4-1: OBSERVER FUNCTIONALITIES.	103
TABLE 4-2: OBSERVER FUNCTIONALITIES AND OPEN MPI COMMUNICATION FRAMEWORKS.....	104
TABLE 4-3: OBSERVER FUNCTIONALITIES AND COMPONENT IMPLEMENTATIONS.	105
TABLE 4-4: PROTECTOR FUNCTIONALITIES.....	105
TABLE 4-5: PROTECTOR FUNCTIONALITIES AND OPEN MPI FRAMEWORKS.	106
TABLE 4-6: PROTECTOR FUNCTIONALITIES AND PROTECTOR DAEMON.	107
TABLE 5-1: NEW ORTE COMMANDS FOR PROTECTOR DAEMON.	117
TABLE 6-1: NAS PARALLEL BENCHMARK PROBLEM SIZE (BAILEY, ET AL., 1995) ...	124
TABLE 6-2: PROCESS SIZE FOR NPB BENCHMARKS CLASS A.	125
TABLE 6-3: PROCESS SIZE FOR NPB BENCHMARKS CLASS B.	125
TABLE 6-4: PROCESS SIZE FOR NPB BENCHMARKS CLASS C.	126
TABLE 6-5: NPB COMMUNICATION PATTERN WHILE USING SIXTEEN PROCESSORS....	127
TABLE 6-6: FAILURES WHICH STARTS THE PROCEDURE FOR APPLICATION STATUS VERIFICATION.	135
TABLE 6-7: EXPERIMENTS EXECUTED TO EVALUATE THE OVERHEAD INTRODUCED BY RADIC IN A FAULT FREE EXECUTION ACCORDING THE APPLICATION COMMUNICATION PATTERN.....	142
TABLE 6-8: OVERHEAD INTRODUCED BY RADIC FAULT TOLERANCE IN NAS BT APPLICATION CLASS A WHILE USING 4, 9, 16 AND 25 PROCESSORS.	144
TABLE 6-9: OVERHEAD INTRODUCED BY RADIC FAULT TOLERANCE IN NAS BT APPLICATION CLASS B WHILE USING 4, 9, 16 AND 25 PROCESSORS.	144
TABLE 6-10: OVERHEAD INTRODUCED BY RADIC FAULT TOLERANCE IN NAS BT APPLICATION CLASS C WHILE USING 4, 9, 16 AND 25 PROCESSORS.	144

TABLE 6-11: OVERHEAD INTRODUCED BY RADIC FAULT TOLERANCE IN NAS SP APPLICATION CLASS A WHILE USING 4, 9, 16 AND 25 PROCESSORS.	146
TABLE 6-12: OVERHEAD INTRODUCED BY RADIC FAULT TOLERANCE IN NAS CG KERNEL CLASS C WHILE USING 4, 8, 16 AND 32 PROCESSORS.....	152
TABLE 6-13: APPLICATION AND KERNELS SELECTED TO ANALYZE THE CHECKPOINTING TIME AND THE MEAN TIME TO RECOVERY – MTTR.....	152
TABLE 6-14: EXECUTION TIME WHILE RUNNING THE SELECTED APPLICATION USING 4 PROCESSORS TO EVALUATE COMPETITION BETWEEN THE CHECKPOINT TRANSFER AND MPI COMMUNICATIONS.	156
TABLE 6-15: COMPARISON BETWEEN APPLICATIONS EXECUTION TIME WHILE SENDING CHECKPOINT FILES OR NOT.	157

Chapter 1

Introduction

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

As a rule of thumb, all computers engineers know that when a computer system becomes more complex, the system susceptibility to faults increases. Such rule is fully applicable to parallel computers. Currently, there is no signal that the trend of increasing the number of nodes in parallel computers will stop. Such trend forces computers and software engineers to increase the amount of effort dedicated to deal with faults.

Dealing with fault tolerance has become a major task 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 fault tolerance mechanism operation interferes on the parallel application operation. Such interference often appears as a performance loss for system users.

RADIC – Redundant Array of Distributed Independent Fault Tolerance Controllers (Duarte, 2007), is a transparent architecture which provides fault tolerance to message passing system. The message passing systems is commonly used to develop parallel applications. RADIC acts as a layer that

isolates the parallel application from possible failures. This architecture does not demand extra resources to provide fault tolerance.

RADIC has two entities working together in order to perform fault tolerance tasks: *observer* and *protectors*. Each node has one protector and each application process has one observer. These entities perform *receiver based message pessimistic logging* and *uncoordinated checkpointing* in order to recover the parallel application from a failure and *heartbeat/watchdog* mechanism to detect faults.

To validate the RADIC architecture, Duarte (Duarte, et al., 7-9 Feb. 2007) has created a prototype named RADICMPI which implements a subset of the MPI – Message Passing Interface (Forum, 1993), standard. Meanwhile, Duarte announces that his fault tolerance architecture can be implemented over an existent message passing library if this message passing library provides all resources necessary to implements RADIC fault tolerance tasks (Duarte, 2007).

In this thesis, we present our journey to prove this affirmation. This work is addressed to any user, developer or researcher focused in to use or to implement fault tolerance in message passing libraries.

1.1 Goals

In order to prove the affirmation made by Duarte, we will select an MPI implementation. The message passing library characteristics, to implement RADIC, must support all RADIC requirements.

Our major premise is to keep the RADIC decentralization, transparency, flexibility and scalability characteristics. Thus, we must make an analysis of the RADIC architecture identifying the fault tolerance

functionalities in order to implement these functionalities in a well-know message passing library implementation.

The RADIC transparency must be maintained in this new implementation allowing us to manage the entire process without any user intervention, or application code changes. In order to keep the fully distributed operational mode from RADIC, all nodes should to work independently, exchanging information as needed just with a few neighbor nodes. Our work must keep the flexibility of RADIC from the point of view of allowing different structures and relationships between its entities and RADIC configuration parameters.

We like to perform several experiments with our RADIC implementation over Open MPI in order to validate its functionality and to evaluate its appliance in different scenarios. We will analyze the execution to validate its operation. For this validation we will use a synthetic program due to its simplicity.

The evaluation of our solution will be made comparing the effects of the message logging protocol and the checkpointing operation. We will use different environments while executing a well-know benchmark.

1.2 Organization of this Dissertation

This dissertation contains seven chapters. In the next chapter, we present the RADIC concepts and describe RADIC Architecture operation and entities.

Chapter 3 talks about message passing libraries, which include the RADICMPI prototype created by Duarte (Duarte, et al., 2006) to test the

RADIC architecture, and the Open MPI library. In this chapter we discuss about current research in fault tolerance for message passing systems.

In Chapter 4, we present a new point of view of the RADIC architecture used to implement it in a message passing library. The message passing library chosen is described in details, as well as the methodology used to merge the two architectures.

Chapter 5 discusses about RADIC architecture implementation details over Open MPI. In Chapter 6 we validate the implementation and evaluate it according the application communication pattern and state size. Finally, in Chapter 7 we state our conclusions and future works.

Chapter 2

The RADIC Architecture

This chapter discusses about characteristics and behavior of the architecture chosen as basis of our work. In his work, Duarte (Duarte, 2007) introduces a new fault tolerance architecture called RADIC, an acronym for Redundant Array of Independent Fault Tolerance Controllers.

The RADIC architecture provides fault tolerance for message passing systems. Furthermore, our intention is not discuss about message passing fault tolerance concepts. These concepts are depicted in Appendix I. The RADIC architecture has been developed to be transparent, decentralized, flexible and scalable. Table 2-1 depicts the RADIC key features.

Table 2-1: RADIC key features.

Feature	How it is achieved
Transparency	<ul style="list-style-type: none">— No change in the application code— No administrator intervention is required to manage the failure
Decentralization	<ul style="list-style-type: none">— No central or fully dedicated resource is required. All nodes may be simultaneously used for computation and protection
Scalability	<ul style="list-style-type: none">— The RADIC operation is not affected by the number of nodes in the parallel computer
Flexibility	<ul style="list-style-type: none">— 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

2.1 RADIC Architecture Model

RADIC establishes an architecture model that defines the interaction of the fault-tolerant architecture and the parallel computer structure. Figure 2-1 depicts how the RADIC architecture interacts with the parallel computer and the parallel application structure. RADIC implements two layers between the *message passing standard* and the computer structure. The lower layer implements the fault tolerance mechanism and the higher layer implements the fault masking and message delivering mechanism.

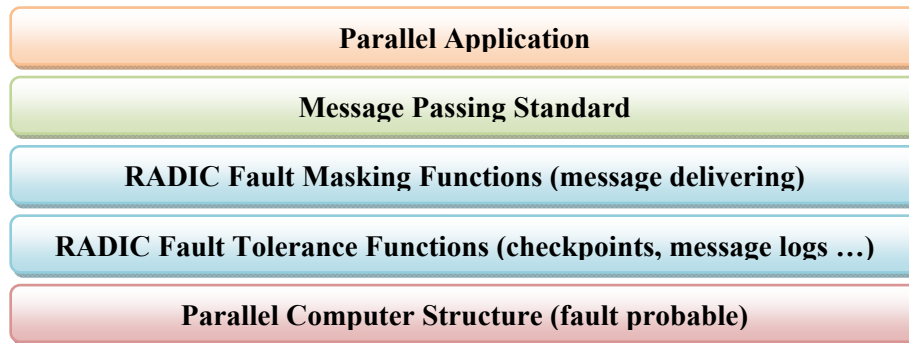


Figure 2-1: RADIC levels in a parallel system.

The RADIC core architecture is a fully distributed controller fault tolerance which automatically handles faults occurred in the cluster structure. Such controller uses the same computational resources used by the parallel application. Furthermore, it is capable to survive to failures.

2.2 RADIC Functional Elements

The RADIC architecture structure uses two entities in collaboration to create a distributed controller for fault tolerance. These entities are *protectors* and *observers*. Every parallel computer node has a protector process running and there is an observer attached to every parallel application process.

2.2.1 Protector

As explained, there is a protector process in each parallel computer node. Each protector communicates with two neighbor protectors. Therefore, all protectors establish a protection system throughout the parallel computer nodes. In Figure 2-2, we depict a simple cluster built using N nodes and a possible relation of the respective P protectors of each node.

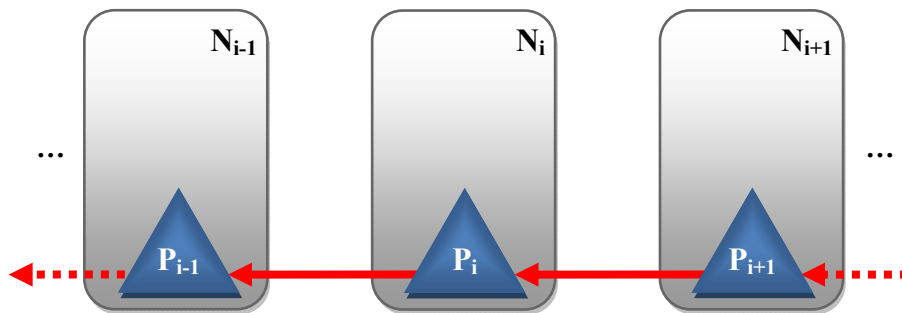


Figure 2-2: An example of protector in a cluster with N nodes. Red arrows indicate the relationship existent between them.

The relationship between neighbor protectors exists thanks to the fault detection procedure. There is a *heartbeat/watchdog* mechanism running between two neighbor protectors: everyone has the watchdog and receives heartbeats from the other. The arrows in Figure 2-2 indicate the heartbeat signals source and destination. Each protector, simultaneously, acts as a sender for a neighbor and as a receiver for the other neighbor.

Each protector executes the following tasks:

- a) It stores checkpoints and message-logs from the application processes those are using it as a protector;
- b) It monitors neighbor protectors 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;
- d) It implements the recovery mechanism.

2.2.2 Observer

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. Observers implement the message passing mechanism for the parallel application. Furthermore, each observer executes the following fault tolerance related tasks:

- a) It takes checkpoints and perform the message logging of its application process and send them to a protector running in another node;
- 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 and establishes a new protector;
- d) It maintains a node/*MPI rank* mapping table, called *radictable*, indicating the location of all application processes and their respective protectors.

2.2.3 RADIC Controller for Fault Tolerance

The collaboration between protectors and observers allows the execution of the RADIC controller tasks. Figure 2-3 depicts the same cluster of Figure 2-2 with all RADIC elements, as well as their relationships. The

arrows in Figure 2-3 represent the communications between fault-tolerance elements. Communications between application processes does not appear in the figure because they are related to the application behavior.

Each observer has an arrow which connects it to a protector whom it sends checkpoints and message logs of its application process. Each protector has an arrow which connects it to a neighbor protector. A protector only communicates with their immediate neighbors.

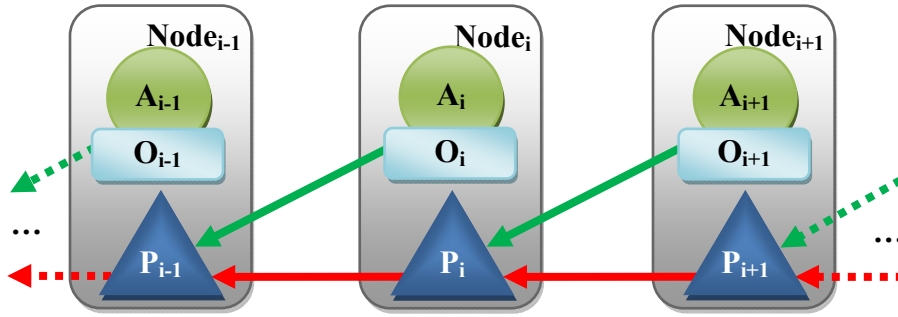


Figure 2-3: A cluster using the RADIC architecture, A_n are application processes, O_n are observers and P_n are protectors. Green arrows represent the relationship between observers and protector and red arrows the relationship between protectors.

The RADIC controller uses a *receiver based pessimistic log* rollback-recovery protocol to handle faults in order to satisfy the scalability requirement. This protocol is the only one in which the recover mechanism does not demand synchronization between in-recovering process and processes which has been not affected by the fault.

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

2.3 RADIC Operation

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

2.3.1 Message Passing Mechanism

In the RADIC message passing mechanism, an application process sends a message through its observer. The observer takes care of delivering the message through the communication channel. Similarly, all messages which come to an application process are received by the observer. The observer delivers the messages to the application process. Figure 2-4 clarifies this process.

To discover a destination process address, the observer uses its routing table, the *radictable*. The radictable represents a mapping between the process identification, or rank, and the node identification, or address. Table 2-2 represents a typical *radictable*.

Table 2-2: A *radictable* example in a cluster with N nodes

Process ID	Address	Protector	Logical Clock for sent messages	Logical Clock for received messages
0	Node 0	Node N	2	3
1	Node 1	Node 0	0	0
2	Node 2	Node 1	0	0
3	Node 3	Node 2	1	1
...

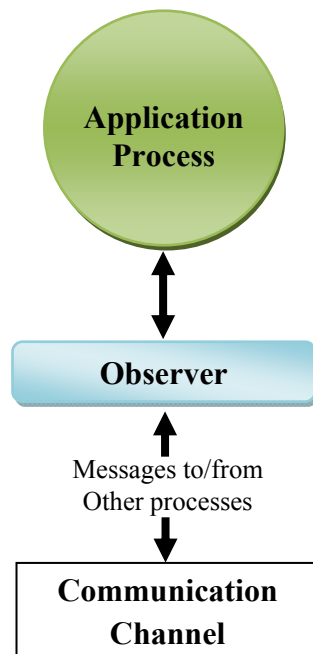


Figure 2-4: Message passing mechanism in RADIC.

2.3.2 State Saving Task

To perform this task, protectors and observers collaborate in order to save snapshots of the parallel application state. The system must supply storage space for checkpoints and message logs required by the rollback-recovery protocol.

Checkpoints

Each observer takes checkpoints of its application process and sends them to his protector located in another node. Figure 2-5 depicts a simplified scheme to clarify the relationship between an observer and its protector.

A checkpoint is an atomic procedure. The process becomes unavailable to communicate while a checkpoint procedure is in progress. The fault detection mechanism must identify a communication failure caused by a real failure from a communication failure caused by a checkpoint procedure. We explain this differentiation in item 2.3.3.

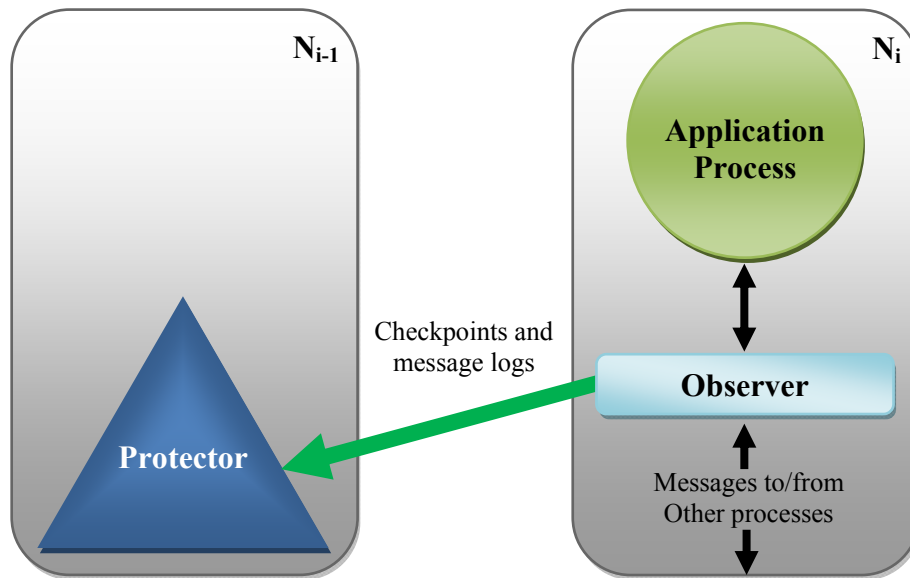


Figure 2-5: Relation between observer and its protector.

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. Thus, if a process fails, all information required to recover it is in a survivor node.

Thanks to the uncoordinated checkpoint mechanism and the message logging protocol used by RADIC, each observer could define an individual checkpoint policy.

A checkpoint represents all computational work done by a process until that moment. Thus, observer sends such computational work to the protector.

As the process continues its work, the state saved in the protector becomes obsolete. In order to update this state, the observer logs all messages which it has received in its protector. Thus, the protector always has all information required to recover a process. Furthermore, such state is always older than the current process state.

Message Logs

Due to the pessimistic log based rollback-recovery protocol, each observer must log all messages received. The use of message logs with checkpoint optimizes the fault tolerance mechanism avoiding the domino effect and reducing the amount of checkpoints which must be maintained.

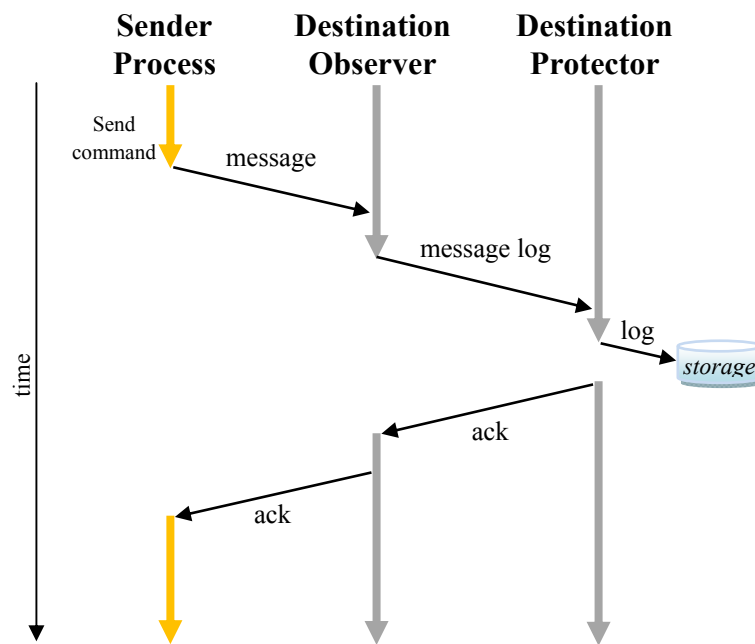


Figure 2-6: Message delivery and message log mechanisms.

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. Figure 2-6 depicts the message delivery and message log mechanisms. The log mechanism enlarges the message latency because the sender has to wait until the protector concludes the message log procedure in order to consider the message was delivered.

2.3.3 Failure Detection Task

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

How Protectors Detect Failures

The failure detection procedure contains two tasks: a passive monitoring task and an active monitoring task. Because of this, each protector has two parts: it is, simultaneously, a heartbeat sender and receiver.

The observer part which receives the heartbeat has a watchdog element and the other part is the heartbeat sender. Figure 2-7 depicts three protectors and the heartbeat/watchdog mechanism between them.

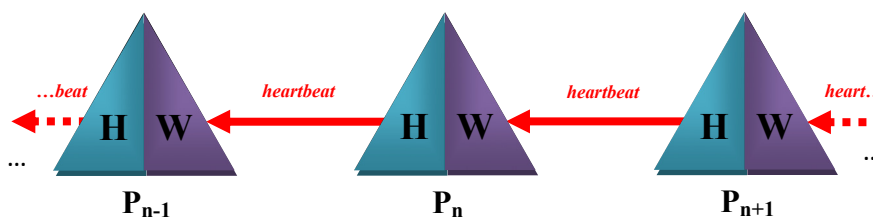


Figure 2-7: Three protectors and their relationship to detect failures.
***W* is a watchdog part of a protector and *H* is a heartbeat send part.**

The heartbeat/watchdog cycle determines how fast a protector will detect a failure in its neighbor. Short cycles reduce the response time, but also increase the interference over the communication channel.

A node failure generates events in the neighbor nodes. If a node confirms a failure it immediately starts a search for a new neighbor. Figure 2-8 represents the operational flow of each protector element. After a failure, the protector backs to send heartbeats and the watchdog side, in turns, begins to wait for heartbeats.

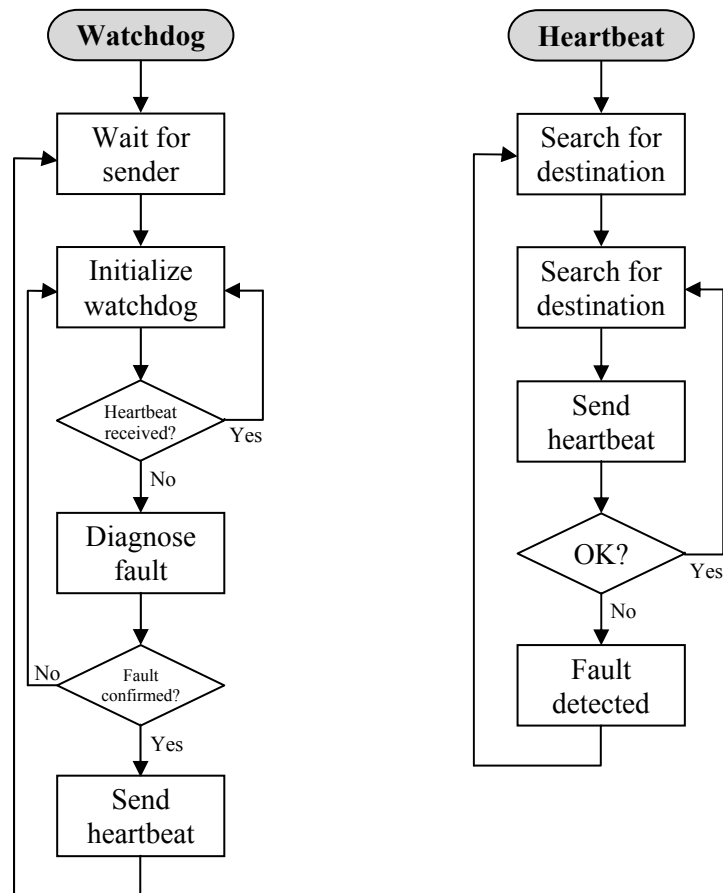


Figure 2-8: Protector algorithms for heartbeat and watchdog tasks.

How the Observers Detect Failures

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 which is running in the same node of the observer). Whenever a protector detects a fail in other node, such protector sends a warning message to all observers in its node. When an observer receives such message, it immediately establishes a new protector and takes a checkpoint.

How the Observer Confirm a Failure

There are two situations which create a communication failure between application processes. Furthermore, it could 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 followings paragraphs, we explain how the observers get rids of the first problem. We will explain how observer gets rid of second situation in item 2.3.5.

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

In order to avoid this fake failure detection, the sender observer contacts the destination protector and asks about the destination process status. Using its *radictable*, any sender observer may locate the destination protector. Thus, the sender observers can discover if the communication failure is consequence of a checkpointing procedure.

2.3.4 Recovery Task

In normal operation, protectors are monitoring nodes, and observers take care about checkpoints and message logs of the distributed application processes. When protectors and observers detect a failure, both actuate to reestablish the consistent state of the parallel application and the structure of the RADIC controller.

Reestablishing the RADIC Structure after Failures

Protectors and observers implicated in the failure will take actions in order to reestablish the integrity of the RADIC controller structure. Table 2-3 depicts the activities of each element.

Table 2-3: Recovery activities performed by the each entity implicated in a process failure.

Protector	Observer
Heartbeat sender: — Fetches a new heartbeat destination — Reestablishes the heartbeat mechanism — Commands the local observers to checkpoint	Survivor: — Establish a new protector — Take a checkpoint
Watchdog: — Waits for a new heartbeat sender — Reestablishes the watchdog mechanism — Recovers the failed processes	Recovered: — Establish a new protector — Copy current checkpoint and message log to the new protector — Replays message from the message-log

When the recovery task is finished, the RADIC controller structure is reestablished and the system is ready to manage new failures. Figure 2-9 presents the configuration of a cluster from a normal situation until the recovery task has finished.

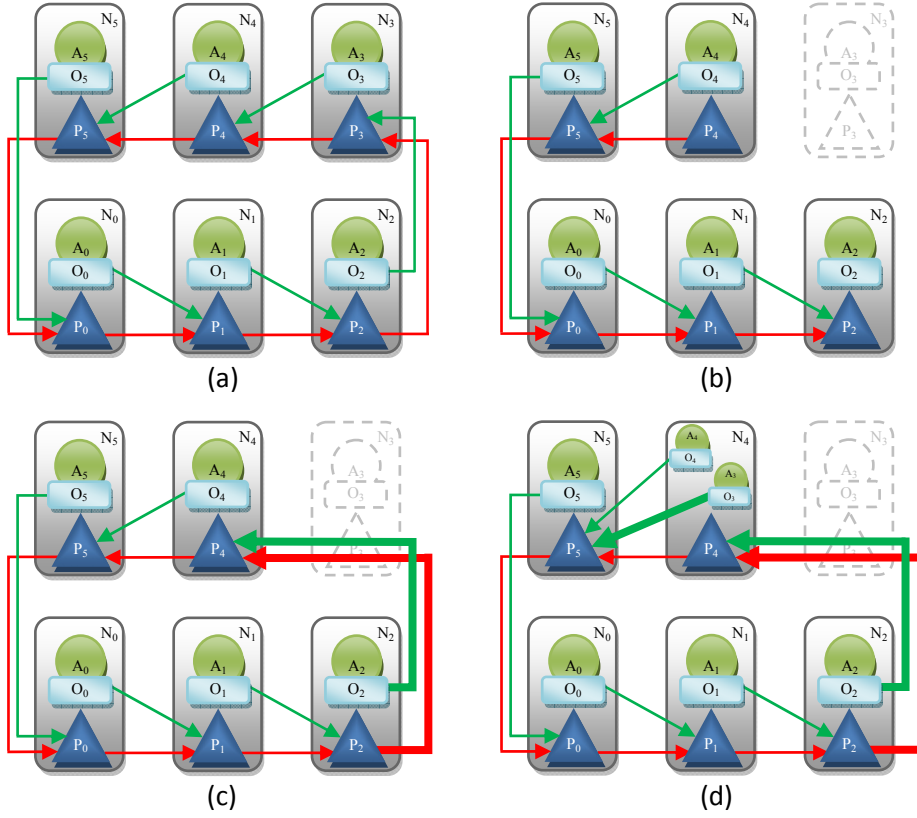


Figure 2-9: Recovering tasks in a 6-node cluster. (a) Failure free cluster. (b) Failure in node N_3 . (c) Protectors P_2 and P_4 detect the failure and reestablish the protection. (d) P_2 recovers A_3/O_3 .

Recovering Failed Application Processes

The protector which stores checkpoint and message logs of the failed process recovers it in the same node in which the protector is running. Immediately after the recovery, each observer connects to a new protector. The recovered observer gets the information about its new protector from the local node protector.

2.3.5 Fault Masking Task

The fault masking is an observer attribution. In case of failures, the observer assures that the processes continue to correctly communicate through 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 identification of all parallel application processes associated with their respective protectors, and uses this information to locate the recovered processes.

Locating Recovered Process

When a process fails, its protector detects the fail and starts the recovering procedure. Therefore, the faulty processes now restart their execution in the protector node, resuming since their last checkpoint.

In the explanation of the failure detection task, item 2.3.3, 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 task. Now, we describe the second situation and the solution for it.

After a process failure, all processes which want to communicate with it will detect a communication failure. Thus, these processes will start the fault detection procedure. Figure 2-10 describes the algorithms used by an observer in the fault detection procedure. An observer uses this algorithm only when communication fails while 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 expects that the faulty sender will

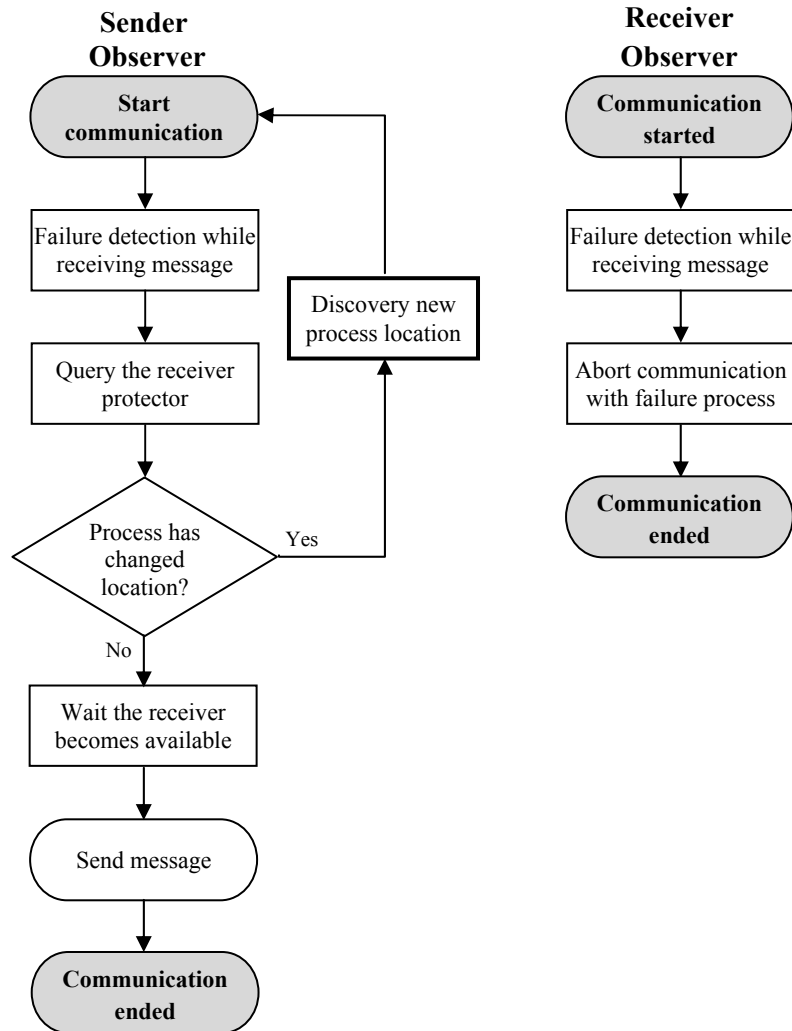


Figure 2-10: Fault detection algorithms for sender and receiver observers.

restart the communication after it has recovered. The algorithm used by the sender observer uses the protector of the receiver process to discover the receiver status.

The example in Figure 2-9d clarifies the location of the recovered process A_3 after a failure in node N_3 . The new protector of A_3 is now P_1 , because A_3 currently is running in the same node of its original protector P_2 . If

some observer tries to communicate with the faulty process A_3 , such observer will obtain a communication error and will ask to the protector P_2 about the status of A_3 . In this case, P_2 informs that it is not responsible for A_3 , because P_1 is now the current protector of A_3 .

In order to find who the current protector of A_3 is, the sender observer uses its *radictable* to follow the protector chain. The sender observer knows that if P_2 is no more protecting A_3 , then the probable protector of A_3 shall be previous entry in the *radictable*. Thus, the sender observer reads its *radictable* and calculates who the A_3 protector is. In our example is P_1 . Now that the sender observer knows who the probable protector of the receiver process A_3 is, it contacts such protector and asks about the status of A_3 . If the protector confirms the location of A_3 , the sender observer updates its *radictable* and retries the communication.

In our example, the updated *radictable* of a process who tries to communicate with the recovered process A_3 has the information presented in Table 2-4. In this table, line three – represent with bold font, shows the update location of process A_3 with its new protector.

Table 2-4: Part of the updated *radictable* of a process that has tried to communicate with A_3 after it has recovered as shown in Figure 2-9.

Process ID	Address	Protector
0	Node 0	Node N
1	Node 1	Node 0
2	Node 2	Node 1
3	Node 2	Node 1
4	Node 4	Node 3
...

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. 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.

2.4 RADIC Flexibility

The RADIC controller allows the setup of two time parameters: checkpoint interval and watchdog/heartbeat cycle. To choose the optimal checkpoint interval is a difficult task, it depends on the application behavior and communication pattern. The watchdog/heartbeat cycle defines the failure detection mechanism sensitivity.

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

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

2.4.1 Concurrent Failures

In RADIC, recover procedure is complete after the recovered process establishes a new protector and has done its first checkpoint. RADIC relies that the protector which is recovering a failed process will not fail before the recovery completion. Nevertheless, RADIC architecture allows the construction of N -protection scheme in order to manage such situation.

In such scheme, each observer would transmit process checkpoints and 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.

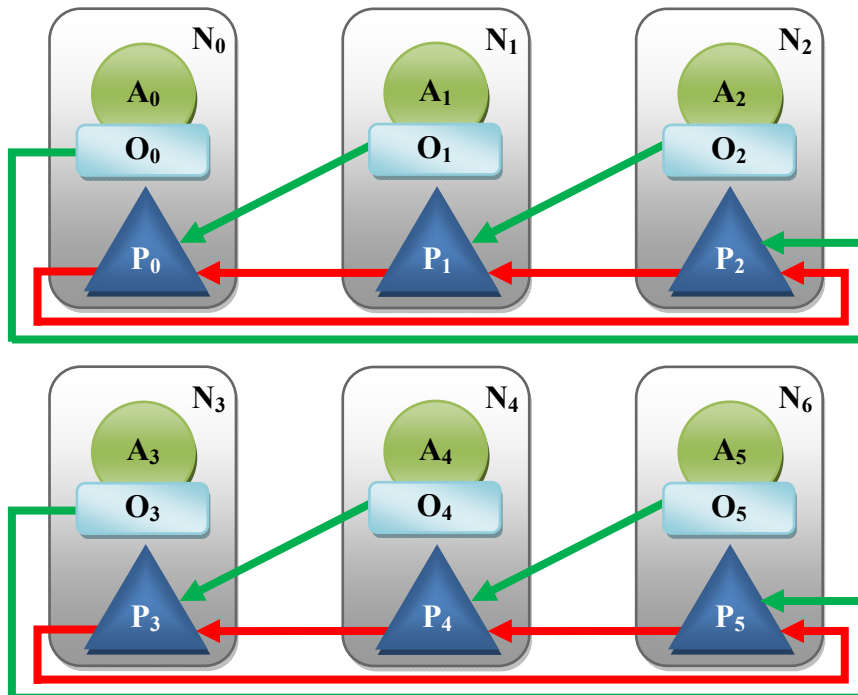


Figure 2-11: A 6-node cluster using two protector chains.

2.4.2 Structural Flexibility

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

One example of RADIC structural flexibility is clustering protector chain. In this case, the system would have several independent protector chains. Therefore, each individual chain would function like an individual RADIC controller and fault tolerance information traffic would be restricted to elements involved in each chain. Figure 2-11 depicts an example using two protector chains.

In order to implement this feature is necessary to add one column to *radictable* which indicates the protector chain. An observer uses the information in such column to search the protector of a faulty node inside each protector chain. The bold column in Table 2-5 exemplifies chain information in *radictable*.

Table 2-5: *Radictable* of an observer located in a cluster protected by two protector chain, like in Figure 2-11.

Process ID	Address	Protector	Chain	Logical Clock for sent messages	Logical Clock for received messages
0	Node 0	Node 2	0	2	3
1	Node 1	Node 0	0	0	0
2	Node 2	Node 1	0	0	0
3	Node 3	Node 5	1	1	1
4	Node 4	Node 4	1	2	1
5	Node 5	Node 3	1	1	3

RADIC architecture requires that, in order to manage at least one fault in the system, a minimum of four nodes. This constraint exists because each

RADIC controller fault tolerance protector requires two neighbors. Therefore, at least three nodes must compose a protector chain, also after one fault. We depicted such minimal structure in Figure 2-12:

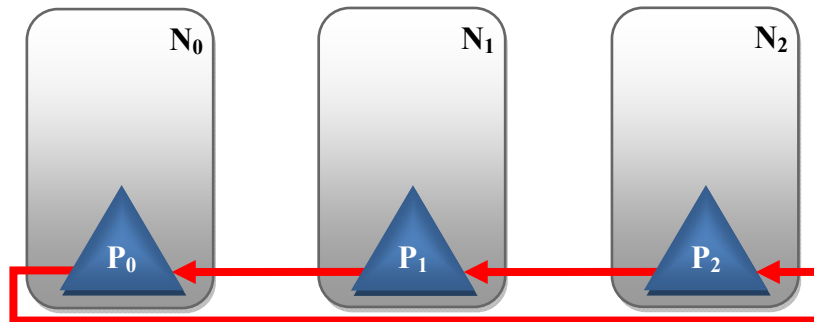


Figure 2-12: Minimum structure for a protector chain. After one fault this cluster will not be fault tolerant anymore.

Chapter 3

Message Passing Library

Implementations

This chapter discusses characteristics and behavior of some message passing library implementations. We start with a prototype created by Duarte (Duarte, 2007) and used by him to test and validate RADIC architecture (Duarte, et al., 7-9 Feb. 2007). After this, an Open Source wide-used implementation called Open MPI is explained in details. Finally some comparisons between different fault tolerance implementations are presented. Due to Message Passing Interface – MPI, success it becomes a *de facto* standard for distributed parallel application development using message passing paradigm. So this chapter, basically, discuss about different MPI implementations.

3.1 RADICMPI

RADICMPI is a MPI implementation created by Duarte (Duarte, et al., 7-9 Feb. 2007) to test and evaluate concepts of RADIC architecture. This implementation is based on Open Source softwares and is available for Linux running on Intel IA-32 architectures. To take checkpoints, RADICMPI uses

the popular Berkeley Labs Checkpoint/Restart library – BLCR (Hargrove, et al., 2006).

In order to attend RADIC features (transparency, flexibility, scalability and decentralization) RADICMPI was developed in layers, Figure 3-1 depicts several software levels of a parallel application using RADICMPI.

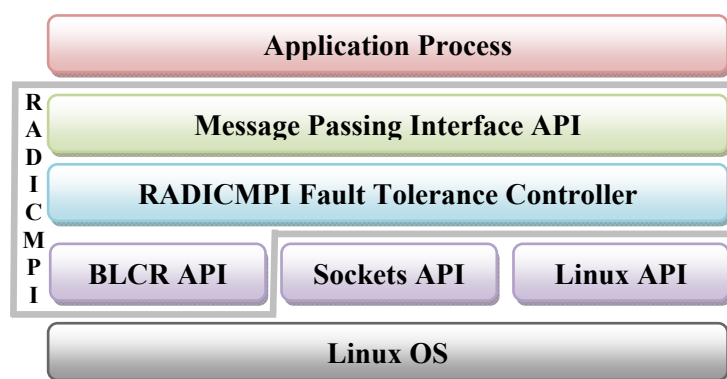


Figure 3-1: RADICMPI software layers.

In order to implement transparency, RADICMPI gives to a programmer a software environment in which a programmer does not need to explicit any directive to achieve fault tolerance. Compile a MPI application using RADICMPI library is enough to get fault tolerance during the execution of this application.

In the cluster administrator side, as depicted in Chapter 2, RADICMPI does not require any central or dedicated element, like a stable storage or a node. All RADIC functional elements, protector and observer, which implement RADIC controller, execute in same nodes where applications processes are. This behavior complies with decentralization requirement.

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

The possibility to modify checkpoint, watchdog and heartbeat interval and protection chain structure gives to RADICMPI flexibility. Table 3-1 summarizes how RADICMPI achieved RADIC architecture characteristics.

Table 3-1: How RADICMPI satisfies RADIC architecture features.

Feature	How it is implemented by RADICMPI
Transparency	<ul style="list-style-type: none"> — A software library implements fault tolerance and message passing mechanisms — Programmer only has to compile application code using RADICMPI library
Decentralization	<ul style="list-style-type: none"> — Protector and observer execute in the same nodes used by application processes — No central or dedicated element is needed by RADICMPI
Scalability	<ul style="list-style-type: none"> — Cluster size does not affect RADICMPI operation, scalability is application dependent
Flexibility	<ul style="list-style-type: none"> — Fault tolerance parameters are defined by the user during application launch — It is possible to implement different checkpoint policies and protection schemas

3.1.1 Observer and Message Passing Interface

In RADICMPI the observer implements the message passing library. It divides RADIC architecture functionalities in threads: program main thread, observer thread and checkpoint thread. Figure 3-2 depicts the relationship existent between these threads.

The main thread executes user program and send to observer thread communication requests based on a subset of the MPI-1 standard. Observer thread implements fault tolerance and message passing mechanisms. This

thread sends checkpoints requests to checkpoint thread. Checkpoint thread deals with checkpoint requests using BLCR library.

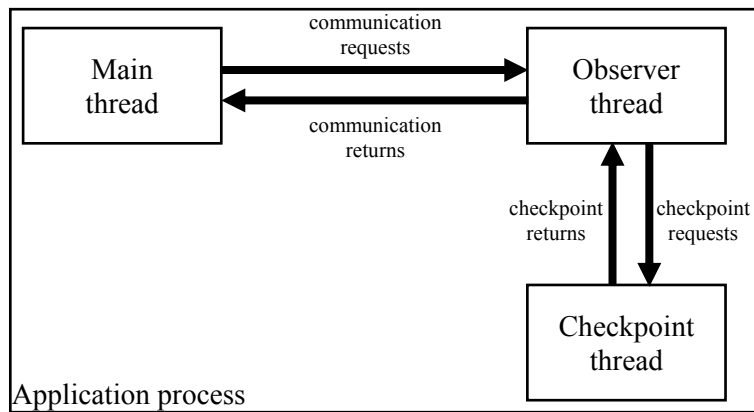


Figure 3-2: Typical application compiled with RADICMPI: thread and their relationships.

Programmer communication necessities are provided by the observer thread that strict follows MPI-1 specifications and syntax. Other tasks are completely transparent for the programmer. Table 3-2 shows MPI-1 standard subset routines implemented by RADICMPI.

Table 3-2: RADICMPI communication routines.

RADICMPI subset of MPI-1 standard		
MPI_Init	MPI_Send	MPI_Isend
MPI_Finalize	MPI_Receive	MPI_Irecv
MPI_Comm_rank	MPI_Sendrecv	MPI_Test
MPI_Comm_size		MPI_Testany
MPI_Wtime		MPI_Testall
MPI_Type_size		MPI_Wait
MPI_Get_processor_name		MPI_Waitany
		MPI_Waitall
		MPI_Allgather

Several events drive the observer functioning, because it implements part of the RADIC distributed controller and the message passing library.

These events can be internal, generated by a thread or external, received from the protector, which implements the other part of the RADIC distributed controller or from another application process.

Figure 3-3 depicts internal and external events which drive observer operation. External events correspond to MPI commands sent by application process located on remote nodes and recovery procedure request sent by a protector. Internal events are consequence of state saving task, fault detection mechanism and MPI commands generated by the local application.

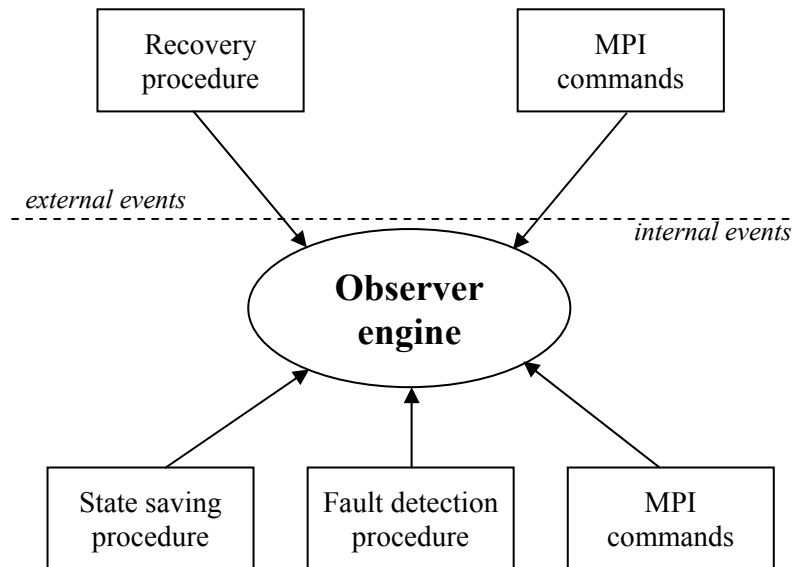


Figure 3-3: Observer engine with internal and external events.

State Saving Procedure

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

When observer receives an external message it generate an internal event related with state saving task described in item 2.3.2. This event is the

message logging event. The observer write a receiver message log into his protector located in other node.

The communication between observer and its protector is simpler than to communication between two observers. According RADIC architecture, an observer always has a connection established with its protector. So when an observer receives a message, before route this message to application process, it copy this message to its protector through the connection existent between them. RADICMPI implement this routine in a different way. The connection used to log messages is opened and closed for each log operation. It occurs because the BLCR cannot checkpoint a process while it has opened connections. Figure 3-4 presents this procedure.

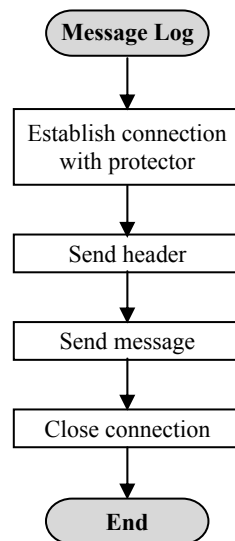


Figure 3-4: Message log procedure

Checkpoint events are started by time counter in the observer and its interval is defined by the user during application launch. This timer is defined at startup and redefined after a recover procedure or checkpoint completion.

The checkpoint procedure behavior depends on BLCR library. BLCR does not perform an application checkpoint if such application has open sockets, except if this connection is a parameter to the BLCR checkpoint routine, and will be used to transmit checkpoint data. To workaround this BLCR characteristic, the observer uses a lock mechanism to block checkpoint procedure if a communication is running. Similarly, observer never initiates nor accepts communications while it is taking a checkpoint.

Due to the uncoordinated checkpoint protocols used by RADIC architecture, a fake error condition could occur if a process tries to communicate with a destination that is taking a checkpoint. The strategy to deal with this fake error condition is implemented in the fault detection procedure.

Before take a checkpoint, observer connect to its protector in order to inform this action, so BLCR API is called to start a checkpoint and transmit checkpoint data *via* the opened connection. After transmission finishes, the observer closes the connection with its protector who knows that the

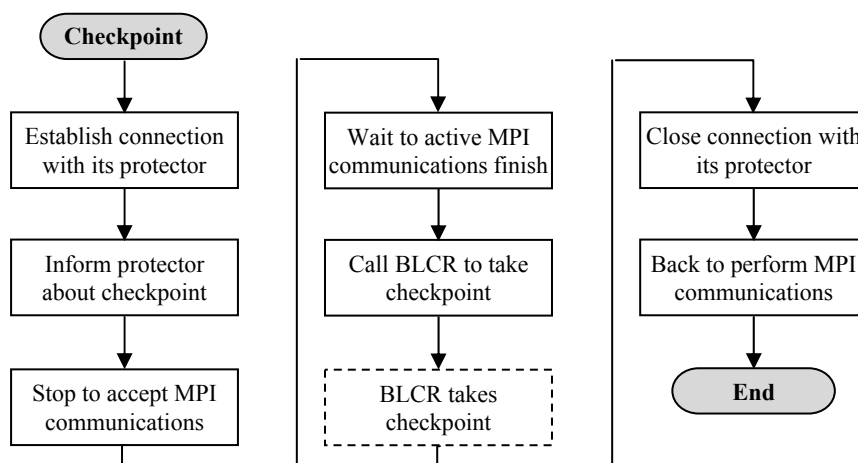


Figure 3-5: Checkpoint procedure.

checkpoint has finished and back to receive and transmit messages. Figure 3-5 presents a complete view of this procedure.

Recovery Procedure

Recovery procedure is the way to revival a process from a previous checkpoint, according to RADIC architecture operation. Recover a process is not an observer attribute, protectors do it, but after a recover, observer must process the message log after recovery. So, observer copy message log contents to an incoming message buffer. Therefore, when application process requests those messages again, they are already in reception buffer. If the recovered application tries to retransmit messages the destination observer discards those messages. These operations are based in message counter information present in *radictable*.

Fault Detection Procedure

As depicted in Figure 3-3 fault detection procedures generate internal events to observer engine. The fault detection procedure mechanism bases on communication failures and communication timeouts in order to detect a fault. There are two errors that can be detected: errors between two observes and errors between observer and protector.

Due the RADIC architecture fault detection task always starts the fault masking task, in RADICMPI those tasks goes together.

The sender observer can detect a error before it initiates a communication. This situation occurs when the destination observer does not accept a connection. There are two reasons to it occurs: the destination observer is checkpointing or the destination node has failed, so the destination process is running in another node.

In both cases observer first contact the destination process protector to argue about its state. If protector answers that the process is checkpointing sender observer keep answering until get checkpointing finish, then it retry the communication.

If an observer detects a failure after a communication has been started exists two possibilities: observer is receiving a message or sending it. If it is receiving a message, the observer simply aborts communication because it knows that the sender will retry again. If it is sending a message, the observer assumes that the destination has failed and uses *radictable* information to discover the new application address, and then retry the communication.

Communication failure between an observer and its protector may occur in two moments: during state saving task and in fault masking task.

If the failure occurs during state saving task, the observer immediately argues the local protector in order to look for a new protector. Then it takes a checkpoint in order to reestablish the protection chain and finally concludes the communication.

During a fault masking task, observer needs to contact the destination protector. If a failure occurs at this moment, observer uses the *radictable* information to calculate the new protector location and retry the communication.

3.1.2 Protector

Protectors are the other RADIC controller part. They operate like a distributed stable storage and as a distributed fault tolerance detector. In RADICMPI implementation, protector has three main threads that deal with observers: observer management, heartbeat and watchdog thread. Different

than observers, protectors executes as separated programs, they do not share memory space with application processes.

Observer Management Thread

This thread exists exclusively to receive checkpoints and message logs from observers. Protector maintains a list of observer that uses it as stable storage and updates this list according observer activities. To response queries about applications status, protector relies in such list.

To save disk space, when protector receives a new checkpoint file from an observer it discard the older and the related message log too.

Heartbeat Thread

When a heartbeat thread starts it immediately fetches another protector in order to sending heartbeats. For each heartbeat sent this thread expects an acknowledge message. If this response does not come or if some communication error occurs, heartbeat thread starts the first fault confirmation procedure that consists in send an additional heartbeat. If two consecutives heartbeats do not get an acknowledge response, protector starts the second fail confirmation procedure described in item 2.3.3.

When the heartbeat finally confirms a failure it warns all local observers that their protector has failed. Then it uses the radictable information to calculate the new heartbeat destination.

Watchdog Thread

Watchdog thread waits for heartbeats to establish the protection chain. Heartbeats received by this thread reset the watchdog timer. For each heartbeat received, protector returns an acknowledge message to sender.

If watchdog timer expires, protector starts the first fail confirmation procedure that consists in wait an additional watchdog cycle to assure that the heartbeat sender is unavailable. If two consecutives heartbeats do not arrive protector starts the second fail confirmation procedure described in item 2.3.3.

When the watchdog thread finally confirms a failure it takes care of recovering faulty process from the faulty node. To do it protector calls BLCR API giving to it the process checkpoint file.

3.2 Open MPI

The Open MPI project was initiated with the express intent of providing a framework to address important issues in emerging networks and architectures, and other. Building upon prior research, and influenced by experience gained from the code bases of the LAM/MPI (Squyres, et al., 2003), LA-MPI (Graham, et al., 2003), FT-MPI (Fagg, et al., 2003), and the PACX-MPI (Keller, et al., 2003) project, Open MPI is an all-new, production-quality MPI-2 implementation. Its component architecture provides both a stable platform for cutting-edge third-party research as well as enabling the run-time composition of independent software add-ons.

While all participating organizations have significant experience in implementing MPI, Open MPI represents more than a simple merger of the LAM/MPI, LA-MPI, FT-MPI, and PACX-MPI code bases. While influenced by previous implementation experiences, Open MPI uses a new software design to implement of the Message Passing Interface. Focusing on production-quality performance, the software implements the MPI-1.2 (Forum, 1993) and MPI-2 (Forum, 1997) specifications and supports concurrent, multi-threaded applications

3.2.1 Software Architecture

Open MPI's primary software design motif is a component architecture called the Modular Component Architecture (MCA). The use of components forces the design of well contained library routines and makes extending the implementation convenient. While component programming is widely used, it is only recently gaining acceptance in the high performance computing community (Bernholdt, et al., 2006) (Squyres, et al., 2003). As shown in Figure 3-6, Open MPI is comprised of three main functional areas: *MCA*, *component frameworks* and *components*, which will be detailed below.

The Open MPI software has three classes of components: *Open MPI* components (OMPI), *Open Run Time Environment* (ORTE) components, and

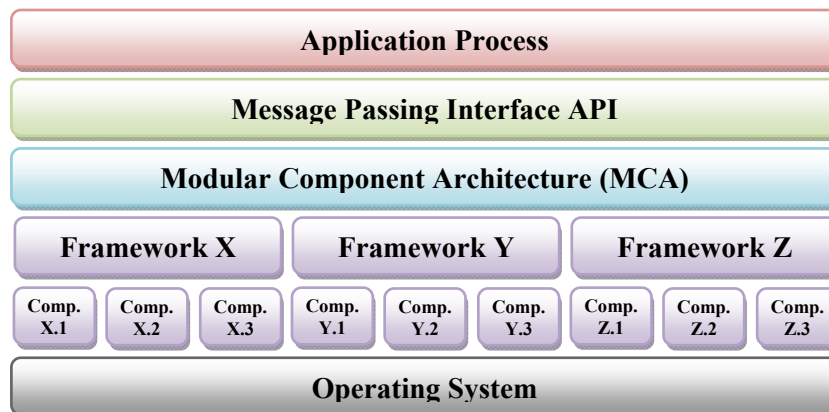


Figure 3-6: Modular Component Architecture (MCA) top-level view.

Open Portable Access Layer (OPAL) components. These classes are combined to provide a full featured MPI implementation, as illustrated in Figure 3-7.

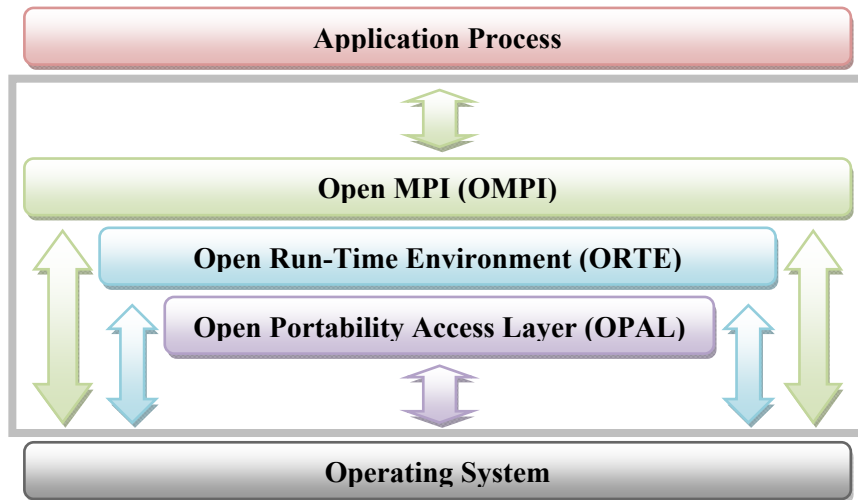


Figure 3-7: The three main component classes in Open MPI

Below the user application is OMPI layer that presents the application with the expected MPI specified interface. Below that is ORTE layer that provides a uniform parallel run-time interface regardless of system capabilities. Next is OPAL that abstracts the peculiarities of a specific system away to provide maximum portability. Below OPAL is the operating system running on machine.

Component frameworks classes are not organized in layers, i. e. an OMPI framework can direct call operating systems functions. Furthermore, in Open MPI, it exist a dependency relation between frameworks classes as shown in Figure 3-8, i. e. an OMPI component requires resources provided by an ORTE component, which requires resources provided by an OPAL component.



Figure 3-8: Open MPI framework class dependency relation.

Each component class has a set of component frameworks that are organized according its functionality, i.e. a component that launch process is located in ORTE class.

Modular Component Architecture (MCA)

MCA is the backbone component architecture that provides management services for all other layers.

The MCA manages the component frameworks and provides services to them, such as the ability to accept run-time parameters from higher-level abstractions and pass them down through the component framework to individual components. The MCA also finds components at build-time and invokes their corresponding hooks for configuration, building, and installation.

Component Frameworks

Each major functional area in Open MPI has a corresponding back-end component framework, which manages modules. Each component framework is dedicated to a single task, such as providing parallel job control or performing MPI collective operations. Upon demand, a framework will discover, load, use, and unload components. Each framework has different policies and usage scenarios; some will only use one component at a time while others will use all available components simultaneously.

Components

Components are self-contained software units that export well-defined interfaces and can be deployed and composed with other components. Components adhere to the interface prescribed by the framework that they belong to, and provide requested services to higher-level tiers. When an

application or a runtime environment is launched, MCA calls necessary frameworks to realize specific functions. Consequently, each framework calls all available components. Each component loaded in named *module*.

A module can be unique or it can exist in multiple instances, depending of framework specification. For example, if a node has two network cards, a component that uses this card load two modules. On the other hand, a component used to launch processes along the cluster is unique for each runtime environment.

3.2.2 Modular Component Architecture Frameworks

In Open MPI there are several MCA frameworks available. As explained in item 3.2.1 frameworks are classified according its functionality. In this master thesis we are going to analyze only the frameworks that are relevant to fault tolerance.

OPAL is composed by ten component frameworks. These frameworks give to Open MPI portability. One of these frameworks is relevant to this work: *Checkpoint/Restart Service*.

ORTE is composed by thirteen component frameworks. These frameworks permit the creation of the parallel environment. Four of these frameworks are relevant to this work: *File Management*, *Snapshot Coordinator*, *Runtime Message Layer*, and *Out-of-Band*.

OMPI is composed by fifteen component frameworks. These frameworks interact to user application providing MPI API and related communication channels. Two of these frameworks are relevant to this work: *Point-to-Point Message Layer* and *Checkpoint/Restart Coordination Protocol*.

Relevant frameworks descriptions are as follows:

Checkpoint/Restart Service

Open MPI provides a single process checkpoint/restart service framework entitled the OPAL CRS (Checkpoint/Restart Service). Since this framework functionality is limited to a single machine by the single process checkpoint/restart service it is implemented at the OPAL layer. The OPAL CRS framework provides a consistent API for Open MPI to use internally regardless of underlying checkpoint/restart system available on a specific machine. Each such system implements a component in the OPAL CRS framework that interfaces the framework API to the checkpoint/restart system API.

The framework API provides the two basic operations of checkpoint and restart. In addition the OPAL CRS framework requires components to implement the ability to enable and disable checkpointing in the system to assist in protecting non-checkpointable sections of code.

In Open MPI checkpointing is enabled upon completion of `MPI_Init` and disabled upon entry into `MPI_Finalize`. This restriction allows checkpointing only while MPI is enabled since the checkpoint/restart framework is a part of the MPI infrastructure and is therefore initialized and finalized within the library. The framework interface is described in more detail in (Hursey, et al., Jul 2006).

There currently exist two components of the OPAL CRS framework. The first is a BLCR implementation, the same used by RADICMPI. The second is a SELF component supporting application level checkpointing by providing the application callbacks upon checkpoint restart and continue operations (Hursey, et al., 26-30 March 2007). Future OPAL CRS framework

API refinements will allow for checkpoint/restart system hints such as memory inclusion and exclusion operations.

File Management

Open MPI provides a file management framework entitled ORTE FILEM. This implementation requires knowledge of all of the machines in the job, but does not require knowledge of MPI semantics therefore it is implemented as a part of the ORTE layer. The framework interface provides Open MPI the ability to pass a list of peers and local and remote file names. If the remote file location is unknown by the requesting process then the remote process is queried for its location.

The first component available for the ORTE FILEM framework uses RSH/SSH remote execution and copy commands. Additional components of this framework may include standard UNIX commands and high performance out-of-band communication channels.

Snapshot Coordinator

Open MPI provides the ORTE SNAPC framework to compartmentalize checkpointing coordination techniques into components with a common API. This compartmentalization allows for a side-by-side comparison of these techniques in a constant environment.

The initial ORTE SNAPC component implements a centralized coordination approach. It involves three sub-coordinators: a *global coordinator*, a set of *local coordinators* and a set of *application coordinators*. Each sub-coordinator is positioned differently in the runtime environment. The interaction between all fault tolerance frameworks will be explained in item 3.2.5.

Runtime Message Layer

Open MPI provides reliable administrative communication services across the ORTE universe framework entitled the ORTE RML (Runtime Messaging Layer). The RML does not typically carry data between processes – this function is left to the MPI messaging layer itself as its high-bandwidth and low-latency requirements are somewhat different than those associated with the RML. In contrast, the RML primarily transports data on process state-of-health, inter-process contact information, and serves as the conduit for general purpose communications (Castain, et al., 2005).

Out-of-Band

Open MPI provides a communication channel for RML necessities entitled ORTE OOB (Out-Of-Band Communications). OOB carry all traffic from RML communications and is the first communication service available during universe creation.

Point-to-Point Message Layer

Open MPI provides a point-to-point communication framework entitled OMPI PML (Point-to-point Management Layer). This framework addresses for point-to-point communication between application processes. Since this framework acts directly with MPI application it is placed at the OMPI layer. A complete description about the Open MPI communication layers is provided in item 3.2.4. The PML, the upper layer of the point-to-point communications design, is responsible for accepting messages from the MPI layer, fragmenting and scheduling messages across the available BTL (Byte Transfer Layer) modules, and managing request progression. It is also the layer at which messages are reassembled on the receive side. MPI semantics are implemented at this layer, with the MPI layer providing little functionality

beyond optional argument checking. As such, the interface functions exported by the PML to the MPI layer map closely to the MPI-provide the point-to-point API (Woodall, et al., 2004).

Checkpoint/Restart Coordination Protocol

Open MPI provides a checkpoint/restart coordination protocol framework entitled the OMPI CRCP (Checkpoint/Restart Coordination Protocol). Since this framework may require knowledge of MPI semantics it is placed at the OMPI layer. The OMPI CRCP framework provides a consistent API for Open MPI to use internally when such a protocol is required. Each component implements a single protocol. The components are provided access to the internal point-to-point management layer framework (PML) by way of a wrapper PML component. The wrapper PML component allows the OMPI CRCP components the opportunity to take action before and after each message is processed by the actual PML component.

The first component of the OMPI CRCP framework is a LAM/MPI-like coordinated checkpoint/restart protocol presented in (Sankaran, et al., 2005). The protocol uses a bookmark exchange to coordinate processes to form a consistent global snapshot of the parallel job. This component refines the protocol slightly by operating on entire messages instead of bytes. Outstanding messages are posted by the receiving peer and used during failure free operation.

3.2.3 Open Run-Time Environment Details

The Open Run-Time Environment (ORTE) has been designed to be capable of supporting heterogeneous operations, efficiently scale from one to large numbers of processors, and provide effective strategies for dealing with

fault scenarios. The system transparently provides support for inter-process communication, resource discovery and allocation, and process launch across a variety of platforms. In addition, users can launch their applications remotely from their desktop, disconnect from them, and reconnect at a later time to monitor progress.

In item 3.2.2 we presented ORTE frameworks. These frameworks combined create a parallel environment necessary to run parallel application. In Open MPI this parallel environment is named *universe*. Open MPI permit a combination of heterogeneous resources to create a universe, as shown in Figure 3-9.

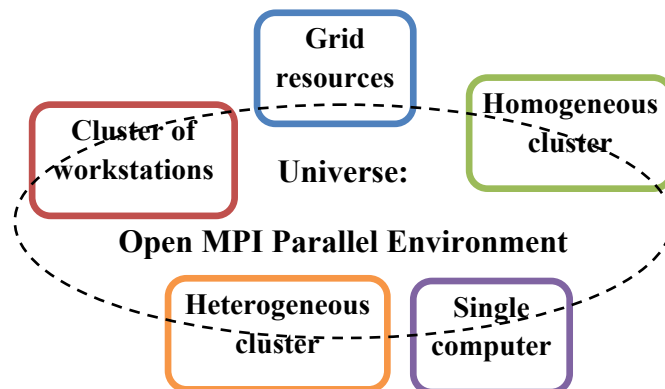


Figure 3-9: Universe: Open MPI parallel environment.

The Open Run-Time Environment has been guided by several key assumptions. As follows we present five of these assumptions:

- a) Users can remotely launch applications and disconnect from the system, and later can reconnect to the running application to monitor progress and adjust parameters “on-the-fly”;
- b) Input/output to/from remote processes may be forwarded user;

- c) Must provides easily interface to monitoring and debugging tools;
- d) For distributed petascale computing systems, ORTE must be capable of supporting applications spanning the range from one to many thousands of processes;
- e) Should support addition of further features and provide a platform for research into alternative approaches for key subsystems;

3.2.4 Communication Layers

The Open MPI communication layers design and implementation is based on multiple MCA frameworks. These frameworks provide functional isolation with clearly defined interfaces (Graham, et al., 2005). Figure 3-10 illustrates the Point-to-Point framework architecture.

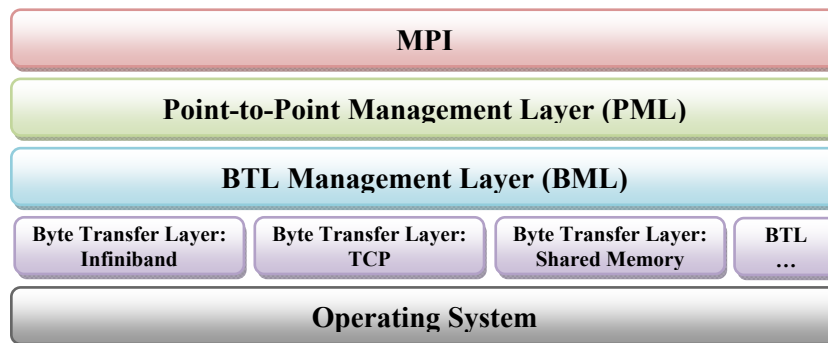


Figure 3-10: Point-to-Point framework architecture in Open MPI.

As shown in Figure 3-10 the architecture consists of four layers. Working from the bottom up these layers are the Byte Transfer Layer (BTL), BTL Management Layer (BML), Point-to-Point Messaging Layer (PML) and the MPI layer. Each of these layers is implemented as an MCA framework.

The BTLs expose a set of communication primitives appropriate for both send/receive and RDMA interfaces. The BTL is not aware of any MPI semantics. It simply moves a sequence of bytes (potentially non-contiguous) across the underlying transport. This simplicity will enable early adoption of novel network devices and encourages vendor support.

The BML acts as a thin multiplexing layer allowing the BTLs to be shared among multiple upper layers. Discovery of peer resources is coordinated by the BML and cached for multiple consumers of the BTLs. After resource discovery, the BML layer is bypassed by upper layers for performance.

The PML implements all logic for point-to-point MPI semantics including standard, buffered, ready, and synchronous communication modes. MPI message transfers are scheduled by the PML based on a specific policy. This policy incorporates BTL specific attributes to schedule MPI messages. Short and long message protocols are implemented within the PML. All control messages (ACK/NACK/MATCH) are also managed at the PML. The benefit of this structure is a separation of transport protocol from the underlying interconnects. This significantly reduces both code complexity and code redundancy enhancing maintainability.

Although, there are currently three PML components available in the Open MPI: *OB1*, *CM* and *DR*. *OB1* is the Open MPI latest generation PML, reflecting the most recent communications research. There is currently only one BML component *R2*. Finally, there is several BTL modules available, providing support for the following networks: TCP, shared memory, Portals, Myrinet/MX, Myrinet/GM, Mellanox VAPI, and OpenIB VAPI.

Moreover, a PML component can be overlapped by a wrapper. Actually Open MPI has a CRCPW component used during fault tolerance operation to allow `ft_event` handling. This wrapper will be explained in item 3.2.5. Wrappers can be concatenated allowing the use of more than one PML wrapper at the same time.

These components are used as follows:

- a) During startup, a PML component is selected and initialized. The PML component selected defaults to OB1 but may be overridden by a run-time parameter. All available wrappers are initialized too.
- b) Next the BML component R2 is selected, since there is only one available. R2 then opens and initializes all available BTL modules;
- c) During BTL module initialization, R2 directs peer resource discovery on a per-BTL component basis. This allows the peers to negotiate which set of interfaces they will use to communicate with each other for MPI communications.

This infrastructure allows for heterogeneous networking interconnects within a cluster.

3.2.5 Fault Tolerance Operation

This section explores the implementation of checkpoint/restart operation in Open MPI. Open MPI modular architecture allows for each checkpoint/restart task to be logically separated into frameworks. Those frameworks where described in item 3.2.2.

In Open MPI, a checkpointing operation starts with a user request. When the user request a checkpoint the SNAPC coordinated component calls the global snapshot coordination.

The global coordinator is a part of the mpirun process. It is responsible for interacting with the command line tools (Figure 3-11A), generating the global snapshot reference, aggregation of the remote files into a global snapshot (Figure 3-11F), and monitoring the progress of the entire checkpoint request (Figure 3-11B and Figure 3-11E). Global coordinator starts a local coordination.

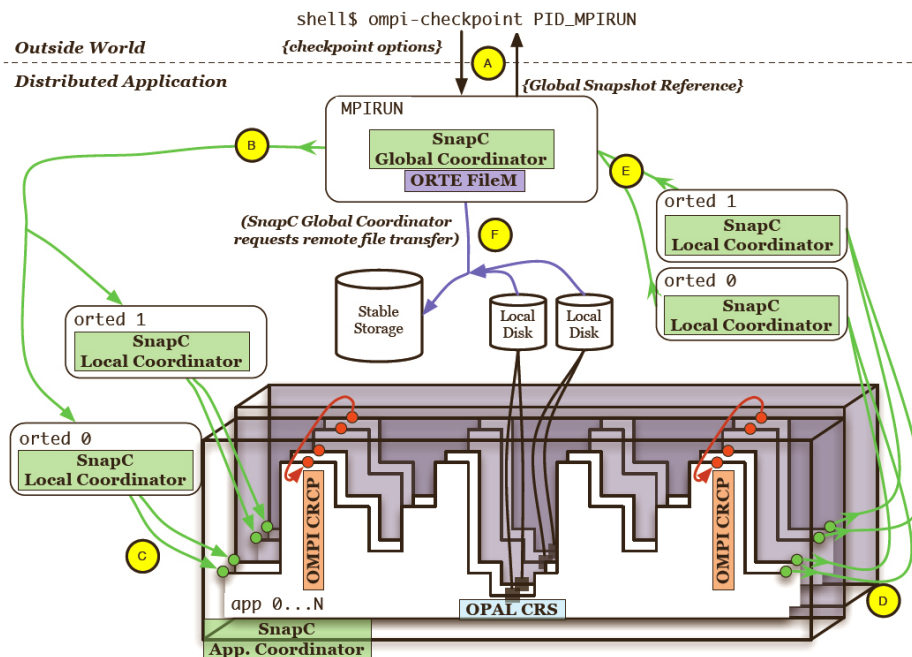


Figure 3-11: Open MPI participating in a distributed checkpoint. 3D boxes represent nodes containing white application processes. Rounded boxes represent processes.

The local coordinator is a part of the ORTE per node daemons (orted). It works with the global coordinator to initiate the checkpoint of a single process

The application coordinator is a part of each application process in the distributed system. This coordinator is responsible for starting the single process checkpoint. Such a responsibility involves interpreting any parameters that have been passed down from the user, i. e. checkpoint and terminate, and calling the OPAL entry point function which begins the interlayer coordination mechanism shown in Figure 3-12.

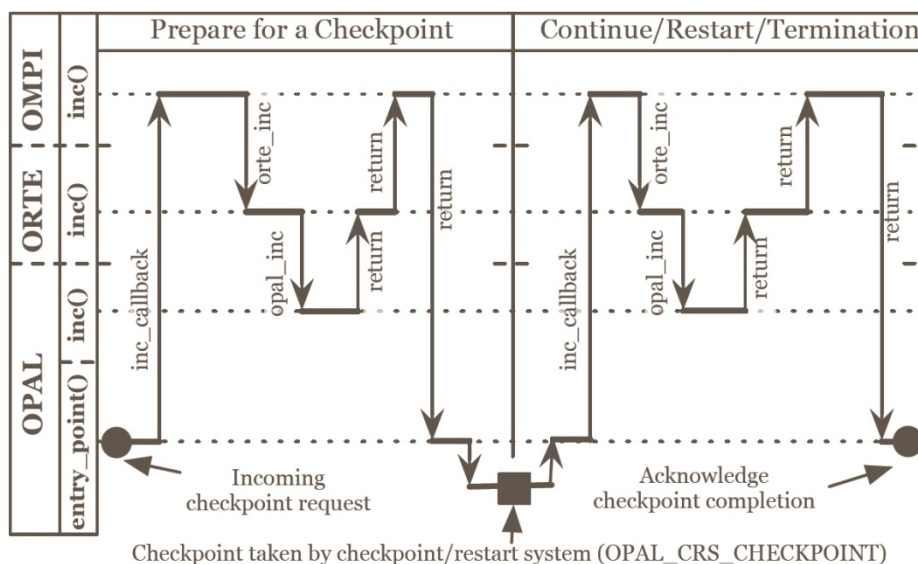


Figure 3-12: Open MPI handling a checkpoint request.

In Open MPI, in order to start application coordination each process in the parallel job has a thread running in it waiting for the checkpoint request. This thread is called the checkpoint notification thread. The thread receives a checkpoint request notification from the system and proceeds into the OPAL entry point function to begin the notification process, as seen in Figure 3-12.

This function then calls the top most registered interlayer notification callback (INC) function.

There are three INC functions in Open MPI, one for each component framework class described in item 3.2.1. If the application registered an INC then it has the opportunity to use the full suite of MPI functionality before allowing the library to prepare for a checkpoint. OMPI is the top most registered INC function. OMPI calls the CRCP component framework and forwards the INC to ORTE.

The CRCP component framework uses a bookmark exchange to coordinate processes to form a consistent global snapshot of the parallel job. This component refines the protocol slightly by operating on entire messages instead of bytes. Outstanding messages are posted by the receiving peer and used during failure free operation.

Once the CRCP component has completed its coordination of the processes then the PML `ft_event` function is called. The PML `ft_event` function involves shutting down interconnect libraries that cannot be checkpointed and reconnecting peers when restarting in new process topologies.

Once the application coordinator has completed the process checkpoint it notifies the local coordinator (Figure 3-11D) that in turn notifies the global coordinator (Figure 3-11E). The global coordinator then requests the transfer of the local snapshots while the processes resume normal operation (Figure 3-11F). Once these local snapshots have been aggregated and saved to stable storage the global snapshot reference is returned to the user Figure 3-11A).

Furthermore, this checkpointing operation is not automatically. It is made manually by an operator and the recovery also.

3.3 Fault Tolerance Message Passing Implementations

In the last ten years several message passing implementations included fault tolerance capabilities. Some of these implementations do offer the requirements expected by actually parallel computers environment. Nevertheless, at the end of this chapter a comparison between that implementations shows that none of them attends simultaneously to the four benefits that RADIC architecture have established: transparency, flexibility scalability and decentralization.

Normally, fault tolerance implementations are based on an existent message passing library. Nowadays there are two major message passing libraries in use: MPICH and Open MPI. Open MPI is an especial case because it was created by the merge of four distributions: FT-MPI, LA-MPI, LAM/MPI and PACX-MPI. Some of these distributions already have fault tolerance capabilities.

For better explanation, the fault tolerance implementations will be showed in three groups: MPICH based implementations; Open MPI based implementations; and finally implementations that are not based on MPICH neither on Open MPI. Open MPI based fault tolerance implementation includes FT-MPI, LA-MPI and LAM/MPI, because there implementations, nowadays, forms Open MPI.

Clearly, some of these fault tolerance solutions are not active projects. However, it has done an important contribution while it implements fault tolerance techniques such as checkpointing and message logging. On the other hand, other solutions reflect researches that are running now.

3.3.1 MPICH Based Implementations

Here are introduced MPICH based implementations.

MPICH-Vx

MPICH-V (Bosilca, et al., 16-22 Nov. 2002) 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 tunneling and repository service. The architecture assumes neither central control nor global snapshots. 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 as follows.

MPICH-V1 (Bosilca, et al., 16-22 Nov. 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 reach a good performance.

MPICH-V2 (Bouteiller, et al., 2003) is designed for homogeneous networks 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 a correct replace of messages during recovers. Computing nodes now keeps message logs.

MPICH-VCL is designed for extra low latency dependent applications. It uses coordinated checkpoint scheme based on the Chandy-Lamport algorithm (Chandy, et al., 1985) in order to decrease the overhead during fault free execution. However, it requires restarting all nodes, even non-crashed ones, in case of fault. Consequently, it is less fault resilient than message logging protocols, and is only suited for medium scale clusters.

MPICH-PCL

MPICH-PCL is a very recent fault tolerant MPI implementation that uses MPICH-2 as message passing platform (Coti, et al., Nov. 2006). MPICH-PCL follows the same architecture modem implemented in MPICH-Vx versions. Therefore, it requires dedicated elements for checkpointing and message management.

3.3.2 Open MPI Based Implementations

Here are presents Open MPI based implementations as well fault tolerance implementations based on independent message passing libraries that actually is part of Open MPI: FT-MPI, LA-MPI, LAM/MPI and PACX-MPI. Furthermore, Open MPI has its own fault tolerance implementation that was described in item 3.2.5.

FT-MPI

FT-MPI (Fagg, et al., 2000) has been developed in the frame of HARNESS (Fagg, et al., 2001) meta-computing framework. The FT-MPI goal is offers to end user a communication library providing an MPI API, which

benefits from the fault-tolerance present in HARNESS system. FT-MPI implements whole MPI-1.2 specification, some parts of MPI-2 standard and extends some semantics of MPI for giving application the possibility to recover from failed processes.

FT-MPI contains the notion of two classes of participating processes within recovery: *Leaders* and *Peons*. Leader tasks are responsible for synchronization, initiating recovery phase, building and dissemination new state atomically. Peons just follow leader orders. In case of a peon dies during recovery, leaders will restart recovery algorithm. In other hand, if a leader dies, peons will enter in an election controlled by the name service using an atomic test and set primitive. A new leader will restart recovery algorithm. This process will continue either until algorithm succeeds or everyone has died (Fagg, et al., 2005).

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. Data delivered integrity is verified at the user level using a checksum or CRC. Corrupt data is retransmitted.

LA-MPI offers a lightweight checksum/retransmission protocols instead of classic TCP/IP protocol. Such protocol allows the use of redundant data paths in networks leading to a high network bandwidth since different messages and/or message fragments can be sent in parallel along different paths. Therefore, LA-MPI guarantees end-to-end network fault tolerance for an MPI application. Furthermore, the application fault tolerance is not available.

LAM/MPI

LAM/MPI (Sankaran, et al., 2005) is a MPI implementation that allows applications checkpoints for later restart. LAM requires a third part single-process checkpoint/restart toolkit for checkpointing and restarting a process. Currently LAM use BLCR library in order to implement coordinated checkpoint protocol.

The approach adopted in LAM/MPI ensures that all MPI communication channels are not in use during checkpointing time its permits a correctly resuming after recovery process.

3.3.3 Others Implementations

Fault tolerance implementations presented below was not based on MPICH neither on Open MPI. Some of these are base on other message passing library and others provide their own.

CoCheck

CoCheck tuMPI (Stellner, 15-19 Apr 1996) addresses fault tolerance application level and was one of the first efforts to incorporate fault tolerance in MPI. CoCheck uses Condor library (Litzkow, et al., 1988) 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.

Starfish

Unlike CoCheck that relies on Condor, Starfish (Agbaria, et al., 1999) uses its own distributed system to provide built in checkpointing. The main difference between Starfish and CoCheck is that the first uses strict atomic

group communication protocols, built upon Ensemble system, in order to handle communication and manage state changes.

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 a specified protocol implementation by gluing together the appropriate objects from an available building blocks library.

MPI/FT

MPI/FT (Batchu, et al., 2004) uses a centralized coordinator that manages fault tolerance mechanism in a transparent way. The coordinator runs in a n-MR (n-Modular Redundancy) mode and monitors application progress, acting as a virtual channel routing every message transferred between process, recovering faulty processes and replaying messages from log to it until the faulty process reach a consistent state. Because of centralized coordination scheme, MPI/FT scalability is limited.

3.3.4 Comparing Fault Tolerant Message Passing Implementations

Table 3-3 presents a summary of implementations analyzed in this chapter. A plus signal indicates that the implementation fully attends and, on the other hand, a minus signal indicates that it not attends, while a plus and a minus signal indicates that it partially attends a specific requirement.

Table 3-3: Fault tolerant MPI implementation comparison.

Implementation	Scalable	Decentralized	Transparent	Flexible
CoChek 1996	-	-	+/-	-
Starfish 1999	-	-	+/-	-
Egida 1999	-	-	+/-	+
FT-MPI 2000	+	-	+/-	-
MPI/FT 2004	-	-	+/-	-
MPICH-V1 2002	-	-	+	+
MPICH-V2 2003	+	-	+	+
LA-MPI 2003	-	-	+	-
LAM/MPI 2005	-	-	+	-
MPICH-PCL 2006	+	-	+	+
Open MPI 2006	-	-	-	-

As one can see, although all solutions are transparent to user, thus none of them simultaneously present the four features. Such lack of a fault tolerant solution that could simultaneously present all four elements has guided the development of RADIC architecture.

Chapter 4

Analysis and Design of RADIC over Open MPI

In order to implement RADIC over message passing library we decided to follow two steps: analyze RADIC and analyze the message passing implementations available. As explained in Chapter 3 MPI is a *de facto* message passing library, and MPICH and Open MPI are the most used MPI implementations.

The first step is obvious, furthermore, in this chapter we analyze RADIC in a different perspective that was used in Chapter 2. Thus, the second step – analyze message passing implementations, consists in choose parameters to evaluate what is the better choice to implement RADIC in contrast to its characteristics and necessities. In item 4.2 we present what parameters were evaluated and why we have decided to use Open MPI.

Finally, the last item of this chapter, that is the main objective of this work, we verify that is possible to implement RADIC in an existent message passing library. It is not obvious how to implement any fault tolerance architecture in a message passing library, RADIC architecture is not an exception.

4.1 Analyzing RADIC

Chapter 2 describes the RADIC architecture and item 3.1 describes the first RADIC implementation, named RADICMPI. Meanwhile, these descriptions are useful to understand the architecture, but to start a RADIC implementation over an existent MPI library other parameters must be analyzed.

Figure 4-1, previous presented in item 2.1 with little modifications, suggest that RADIC functions is a layer between the MPI API and the parallel computer. It means that RADIC implementations do not need to be aware with the message passing API; however, it is directly implicated with message delivering.

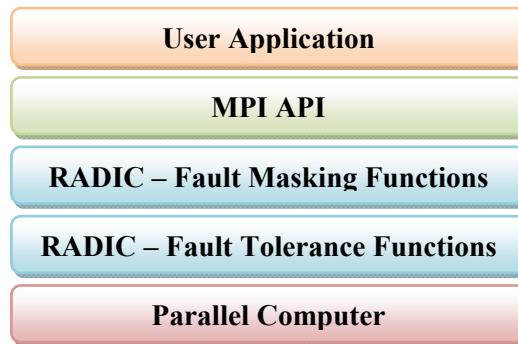


Figure 4-1: RADIC layers.

Thus, the fault masking and fault tolerance functions can be implemented as a transport layer, or part of this. Of course, it is a short perspective, because RADIC architecture must play recovery functions too.

In order to develop this work, we decided to analyze the RADIC architecture as a set of functionalities. Thus, we can analyze message passing libraries looking for the possibility to implement these functionalities.

In a software engineer viewpoint, RADIC architecture depicts two entities, observer and protector, and a relation between them. The relation existent between these two entities have been named *RADIC controller for fault tolerance*. Furthermore, there are only two entities and a relation between them that guide the behavior of these entities.

In order to apply software engineer techniques, we have analyzed RADIC observer and protector looking for these macro functionalities, do not being aware with previous implementations.

Functionalities can be understood as a set of task in order to achieve a requirement.

4.1.1 Observer Functionalities

Observer functionalities can be divided according to the following fault tolerance characteristics: protection, detection, recovery and reconfiguration.

Protection

There are three functionalities for achieve protection that is performed by the observer entity: *look for a protector*, *take and send to protector checkpoints*, *take and send to protector message logs*, and *mask possible communication failures*.

As explained in item 2.2.2 when an application is launched with the observer attached to it, in order to protect this application, observer establishes a connection with its protector. Furthermore, observer takes a checkpoint that defines the first recovery line.

During application execution, all message received by an observer must be logged on its protector. As well as a communication failure must the

treated to start the recovery procedure and never return an error to the user application.

In order to mask communication failures, avoiding that occurs a `MPI_COMM_ERROR`, the observer holds the message request, call the message destination protector to start a recovery procedure and then, retry the message delivery.

Detection

Detecting failures is a protector task. Nevertheless, during message delivery the observer can detect failures when its connection fails during it is sending a message.

Recovery

Recovering processes is a protector task. Furthermore, after protector launches a faulty process, with an observer attached to it, from checkpoint file, the observer must process the message log.

To process the message log consists in replay to application process received messages. On the other hand, a process that receives messages from a recovered process must discard duplicated messages.

Reconfiguration

In order to reconfigure the system after a failure, observers must update their *radictable* while a message delivery attempt fails.

4.1.2 Protector Functionalities

Protection functionalities cannot be compared with any layer in a message passing library because normally do not exist other entity more than

the application process and the message passing library in a parallel execution.

As well as observer functionalities, protector functionalities can be divided according the same characteristics.

Protection

There are two functionalities for achieve protection that is performed by the protector entity: *store checkpoint files* and *store message log*.

These functionalities are considered *passive* because it is depends on observer functionality.

Detection

In order to detect failures there are two functionalities: *heartbeat send* and *watchdog*. They are two faces of the same coin.

When a protector is launched, it looks for their neighbors in order to start to send heartbeats. A parameter can specify the number of consecutives heartbeat failures necessary to start a recovery procedure.

On the other hand, protector waits for heartbeats while it is executing the watchdog task. A parameter can specify the watchdog timeout that starts a recovery procedure.

Recovery

Recovery procedure is a passive functionality. There are three ways to start a recovery procedure: *watchdog timeout*, *consecutives heartbeat failures* and an *observer notification*. In all the three cases the procedure is the same: revival an application process from a checkpoint file.

Reconfiguration

Reconfigure the protection is a passive functionality started by a failure detection confirmation. It is similar to the procedure done while protector launches.

4.2 Choosing a Message Passing Library

When one is looking for a message library to implement RADIC must consider some parameters to evaluate existent libraries. These parameters will guide the analysis necessary to merge two architectures. Furthermore, besides to merge architectures, several hours will be employed in codification. Really, several weeks are necessary to implement RADIC in an existent message passing library.

Thus, in order to adapt RADIC to any message passing implementation is necessary evaluate if the implementation that one is analyzing has a structured code that permits contributions and additions.

Nevertheless, some parameters do not change if someone chooses one implementation or other. These parameters are related with the communication layers. RADIC operation depends on communication layers. If some message passing implementation has their communications implemented as a “black box”, probably implementing RADIC on this message passing will be impossible.

As explained in the beginning of this chapter we have analyzed two implementations: MPICH and Open MPI. After a complete evaluation of both MPI implementations we decided to use Open MPI to implement the RADIC architecture. As follow we present our impressions about each one.

4.2.1 Software Architecture and Code Structure

The Open MPI component architecture provides both a stable platform for third-party research as well as enabling the run-time composition of independent software add-ons. The wide variety of framework types allows third party developers to use Open MPI as a research platform, or even a comparison mechanism for different algorithms and techniques.

The component architecture in Open MPI offers several advantages for end users and library developers. First, it enables the usage of multiple components within a single MPI process. For example, a process can use several network device drivers, or BTLs, simultaneously. Second, it provides a convenient possibility to use third party software, supporting both source code and binary distributions. Third, it provides a fine-grained, run-time, user-controlled component selection mechanism (Gabriel, et al., 2004).

The Open MPI code is organized in directories named according its function. The logical organization of Open MPI is reflected in its code structure.

The Open MPI fault tolerance implementation is a key characteristic that we consider in order to choose Open MPI as bases to start RADIC architecture implementation. We take advantages from the possibility to reuse Open MPI fault tolerance frameworks and develop RADIC specific components for each framework.

On the other hand, MPICH have been developed for efficiency, meanwhile, originally, it is not designed to allow contributions. An exception is made for the communication layer that has been implemented according the *Abstract Device Interface* design pattern.

4.2.2 Communication Layers

As shown in Chapter 3, Open MPI communication architecture has four layers. From the bottom up these layers are the Byte Transfer Layer (BTL), BTL Management Layer (BML), Point-to-Point Messaging Layer (PML) and the MPI layer. Each of these layers is implemented as an MCA framework.

The BTLs expose a set of communication primitives appropriate for both send/receive and RDMA interfaces. The BTL is not aware of any MPI semantics. It simply moves a sequence of bytes.

The BML acts as a thin multiplexing layer allowing the BTLs to be shared among multiple upper layers. Discovery of peer resources is coordinated by the BML and cached for multiple consumers of the BTLs.

The PML implements all logic for point-to-point MPI semantics including standard, buffered, ready, and synchronous communication modes. Moreover, a PML component can be overlapped by a wrapper.

This logical layers specification allows to developers the possibility to add and modify functions on any layer. Developers do not need to be aware about other layers except those that they need to modify.

On the other hand, the MPICH implementation of the MPI standard is built on a lower level communications layer called the abstract device interface. The purpose of this interface is to make it easy to port the MPICH implementation by separating into a separate module that handles just the communication between two processes. In addition, this implementation supports multiple devices in a single MPI application, i. e., TCP/IP and shared memory or TCP/IP and proprietary interconnect.

4.3 Analyzing Open MPI

In Chapter 3 we depict the Open MPI architecture. Furthermore, a low level analysis is necessary to implement the RADIC architecture. The major Open MPI goal to implement RADIC is its software architecture. The advantage to do not be aware with MPI standard modifications, new network patterns or devices network, was decisive to do not continue the RADICMPI development and choose a MPI implementation in order to get a complete MPI-2 (Forum, 1997) compliance RADIC implementation.

As follow we analyses in depth the Open MPI operation and the communications layers operation.

4.3.1 Open MPI Operation

The better to understand the Open MPI operation is understand what happens when a parallel application is launched. It involves the universe creation (parallel environment in Open MPI), the operation itself, and the finalization.

In the item 3.2we depict the Open MPI architecture with the parallel environment named universe that is created by the ORTE. Each machine that is part of the ORTE universe has an application running. This application provides the environment necessary to create and operate the universe. There are two different ORTE applications: `mpirun` and `orted`.

The `orted`, like this name suggest, is the ORTE daemon. This daemon runs in all nodes where an Open MPI parallel application is running. The `orted` was not designed to be launched manually.

The `mpirun` is widely known as the program used to launch a parallel application. However, `mpirun` is executed only in the machine that launches the application. Thus, to create the parallel environment, the `mpirun` launches an `orted` process in each node that will execute the parallel application, as shown in Figure 4-2.

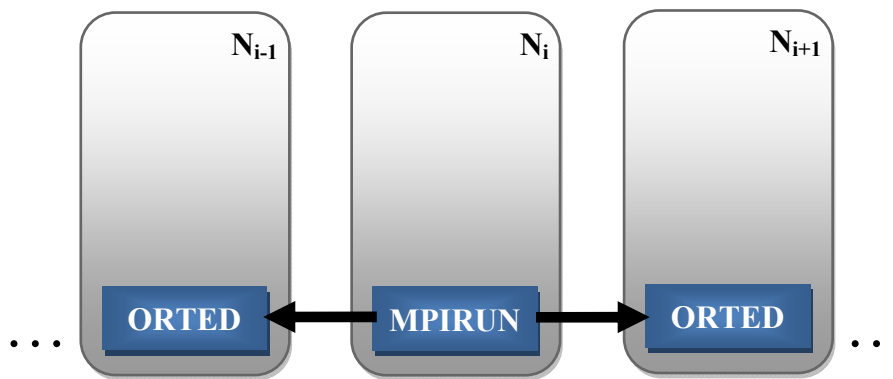


Figure 4-2: `mpirun` launches `orted` processes on the computation nodes.

The information necessary to identify the ORTE daemons is provided by the `mpirun` during the launching operation. The daemons communicate between them to create the universe. Once the universe exists, the daemon and the `mpirun` launch the parallel application.

A node can host more than one application process. However, each node has only one ORTE process instance. `orted` and `mpirun` processes can launch the application process locally, it depends on the configuration chosen by the user.

If the node where `mpirun` has been executed is part of the computation nodes, `mpirun` will launch application processes. Meanwhile, if `mpirun` is not part of the computation nodes, only `orted` process will launch the parallel application. Figure 4-3 depicts these two situations: a) where `mpirun` is part of the computation nodes, and b) where it is not.

The number of application processes per node depends on the user selection. Open MPI consider that each processor in the parallel machine as a slot. The user can allocate application processes by node or by slot. The first distribute processes sequentially across the parallel machine nodes, and the second allocate all slots in a node before start to use other nodes.

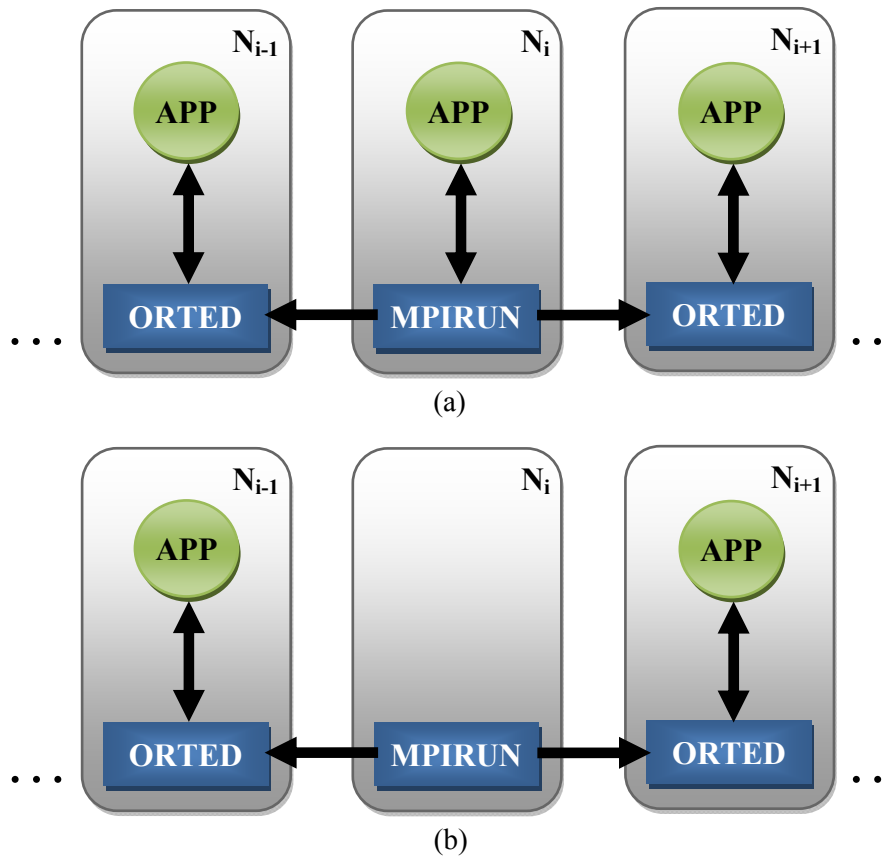


Figure 4-3: (a) mpirun as part of the computation nodes or not (b).

The communication between ORTE processes is made through an out-of-band communication channel implemented by the *Resource Management Layer* (RML) framework. Furthermore, MPI communication do not runs through out-of-band channel. MPI communications are made using a high

performance communication implemented by the BTL framework components, as shown in Figure 4-4.

The Open MPI architecture permits to use more than one network device to communicate between MPI applications or ORTE processes. Furthermore, only a TCP component is available for out-of-band communications.

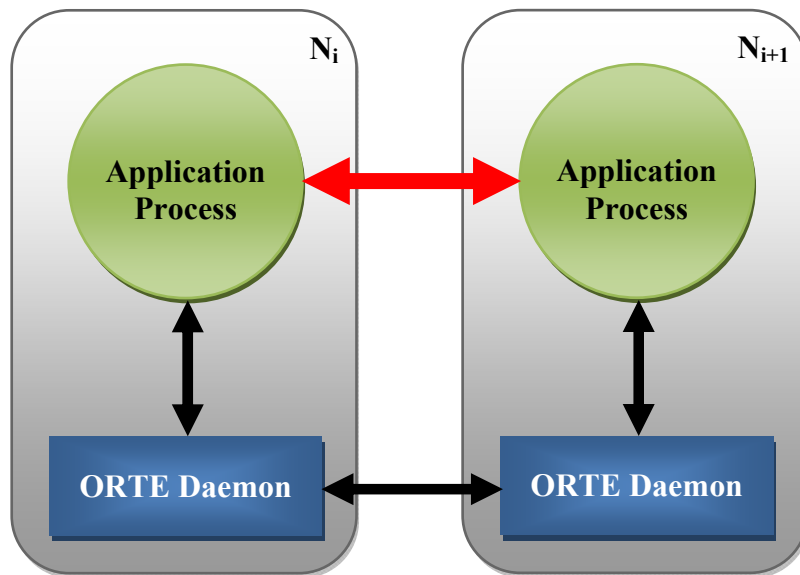


Figure 4-4: Communication between ORTE daemons and between ORTE daemon and application process. Black arrows are RML communication and red arrows MPI communications.

4.3.2 Communication Layers

In order to understand what happens to Open MPI when the user application calls a communication directive we must do a complete call tracing inside the Open MPI communication layers.

In Figure 4-5 we depict the Open MPI communication layer call tracing. The PML wrappers calls go concatenated, that permits the use of more than one wrapper at the same time. This Open MPI characteristic allows the development of small wrapper in order to implement specific functionalities.

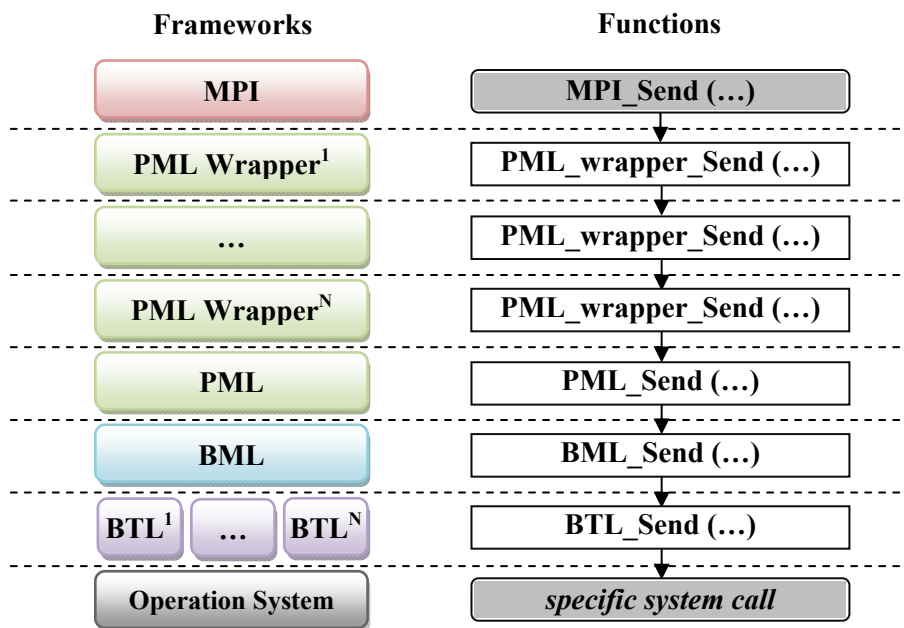


Figure 4-5: Call tracing in Open MPI communications layers.

The BML framework implements a list that contains all BTL available the parallel machine. This list, in addition to BTL, has a priority associated to each BTL. This priority list permits to the BML framework decides what BTL will be used to deliver a user communication request, in other words: the message.

The BTL framework has one component for each network device type. Thus, there are a TCP component, an Infiniband component, etc. These components implements all directives specific to each network device. The

BML layer uses cartography and route framework, in addition to the priority list, to send messages. Once a BML discovery a route to deliver messages to other nodes, this information is cached, and PML uses this cache to bypass BML and avoid one function call in the communication stack.

PML framework, PML wrappers and MPI framework does not have any knowledge about network specific information like address, bandwidth or latency. This abstraction permits that a developer writes a PML component that will work with any network device supported by Open MPI.

The PML framework implements the policies used to manage messages. It means that the PML framework defines the block and synchronisms behavior of each communication directive. Thus, there are a small set of communication primitives that are combined to create other directives, i. e. a PML_Ssend is a PML_Send combined with a PML_Wait.

Finally, the MPI framework is just a layer added to avoid modification on PML framework when the MPI standard changes. Thus, implement new MPI directives that can be created in the future is not a hard task.

4.4 Merging Two Architectures

While merging the RADIC architecture with Open MPI we were concerned to maintain the characteristics of each one. The Open MPI structure has helped us to join both architectures. On the other hand some characteristics turn the merging a hard task.

The analysis made over those architectures was guided on five principles:

- a) Do not change the RADIC architecture concepts or structure. We are implementing RADIC over Open MPI, not doing yet another fault tolerance implementation;
- b) Maintain the Open MPI modularity. The MCA architecture is the best effort in order to permit to researches develops their work using Open MPI. We cannot draw out this advantage while adding RADIC architecture;
- c) The fault tolerance provided by RADIC must work with future Open MPI versions;
- d) Though we are using a specific network to develop and test the implementation, RADIC architecture must be available to all networks devices and types supported by Open MPI;
- e) The RADIC architecture implemented over Open MPI must take advantage of new Open MPI functionalities.

Supported on these principles, we start to construct the methodology to merge these two architectures. Our work takes its base on the RADIC architecture not on the RADICMPI implementation. Thus, some difference exists between RADICMPI and the RADIC architecture implemented for Open MPI.

4.4.1 Methodology

In order to achieve the five principles depicted in item 4.4 we have developed the RADIC architecture using software engineer techniques. Thus, a new component has been developed for each of RADIC functionality or a set of them.

RADIC is the architecture for fault tolerance while Open MPI have been developed using the MCA software architecture. In order to implement the RADIC concept we have used the same software architecture then Open MPI.

There is similarity while comparing the structure of the RADIC node with the Open MPI node some similarity. This similarity is presented in Figure 4-6. This similarity permits a direct comparison between both architectures.

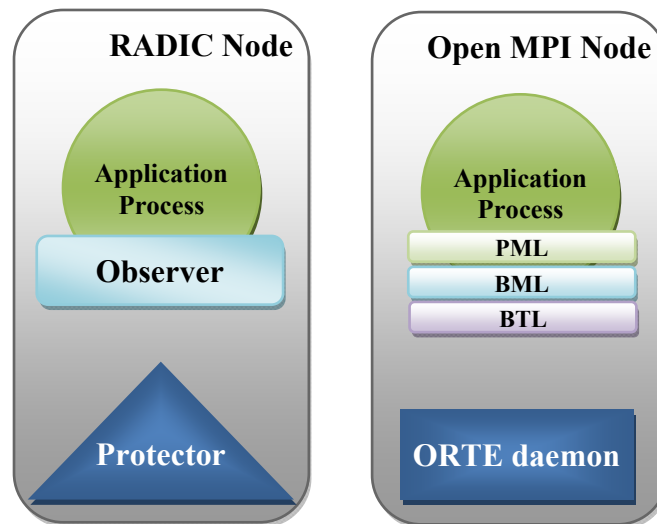


Figure 4-6: Similarity existent between a RADIC and an Open MPI node.

As we can see in Figure 4-6, doing a parallel from a RADIC node to an Open MPI node, the ORTE daemon is comparable to the protector. And observer functionalities, that goes attached to the application process, is comparable to the communications layers present in Open MPI.

Nevertheless, the ORTE daemon is composed by a few component frameworks, with specific component implementation, as well as the Open MPI communication libraries.

Thus, we have started to look for a framework that can shelter each of RADIC functionality present on the observer entity and also on the protector entity. At this point, we are not being aware about component implementation itself, only on the framework functionalities.

4.4.2 Observer inside Communication Message Layers

As depicted in item 4.4.1, the observer functionalities can be implemented on the Open MPI communications layers. Table 4-1 depicts observer functionalities.

Table 4-1: Observer functionalities.

Category	Functionality
Protection	<ul style="list-style-type: none"> — Look for a protector — Take and send checkpoints — Take and send message logs — Communication fault masking
Detection	<ul style="list-style-type: none"> — Detect communication failures
Recovery	<ul style="list-style-type: none"> — Process log after recover
Reconfiguration	<ul style="list-style-type: none"> — Update task/node table

All these RADIC functionalities can be performed by the communication library. However, in Open MPI, the communication library is composed by four frameworks: MPI, PML, BML and BTL. Besides PML are possible PML wrappers.

Thus, we have compared the observer functionalities necessities to the API of the Open MPI communication library framework. This comparison

permits a better accommodation of these necessities on the Open MPI communication layer.

For a better comprehension, we have added one column to Table 4-1 in order to reflect this comparison, as shown in Table 4-2.

Table 4-2: Observer functionalities and Open MPI communication frameworks.

Category	Functionality	Open MPI Framework
Protection	— Look for a protector	PML
	— Take and send checkpoints	PML, SNAPC, CRCP, CRS, RML
	— Take and send message logs	PML, RML
	— Communication fault masking	PML
Detection	— Detect communication failures	PML
Recovery	— Process log after recover	PML
Reconfiguration	— Update task/node table	PML, BML

Nevertheless, some functionalities uses other frameworks to perform all tasks, i.e. to take checkpoints and send then, the new PML component implementation uses the BLCR component from Checkpoint/Restart Service (CRS), as well as to send then across the network uses the RSH component implemented on the File Management framework (FileM).

As can be viewed in Table 4-2, the PML framework can carry more than one observer functionality. Furthermore, there is no component implementation that performs message logging. As well as has no component implementation to perform all other functionalities necessary to RADIC operation. Thus, we have implemented the components necessary.

Table 4-3 depicts the frameworks components that we use to implement observer functionalities. We have developed the *observer* component as a PML wrapper that performs all tasks relative to the RADIC observer.

However, the observer component is a wrapper to the original PML component (OB1) used to manage messages in Open MPI.

Table 4-3: Observer functionalities and component implementations.

Category	Functionality	Framework	Component
Protection	— Look for a protector	PML	observer
	— Take and send checkpoints	PML, SNAPC, CRCP, CRS, RML	observer, single, uncoord, BLCR, OOB
	— Take and send message logs	PML, RML	observer, OOB
	— Communication fault masking	PML	observer
Detection	— Detect communication failures	PML	observer
Recovery	— Process log after recover	PML	observer
Reconfiguration	— Update task/node table	PML, BML	observer, R2

In item 5.1 we describe the implementations details of each component developed.

4.4.3 Protector inside Open Run-Time Environment

Daemon

As depicted in item 4.4.1, the protector functionalities can be implemented on the ORTE daemon. Table 4-1 depicts observer functionalities.

Table 4-4: Protector functionalities.

Category	Functionality
Protection	— Store checkpoints
	— Store message logs
Detection	— Send heartbeats
	— Watchdog
Recovery	— Recover a process
Reconfiguration	— Reestablish monitoring

All these RADIC functionalities can be performed by the ORTE daemon. However, ORTE daemon is composed by an application and many frameworks that depend on resources present on parallel machines such as process launching mechanisms available, network topology, input/output devices and etc.

Thus, we have compared the protector functionalities necessities to the API of the ORTE daemon and the ORTE Open MPI frameworks available. For a better comprehension, we have added one column to Table 4-4 in order to reflect this comparison, as shown in Table 4-5.

Table 4-5: Protector functionalities and Open MPI frameworks.

Category	Functionality	ORTE Element/Framework
Protection	— Store checkpoints	ORTE daemon, FileM
	— Store message logs	ORTE daemon, File M
Detection	— Send heartbeat	ORTE daemon, RML
	— Watchdog	ORTE daemon, RML
Recovery	— Recover a process	ORTE daemon, CRS
Reconfiguration	— Reestablish monitoring	ORTE daemon, RML

Nevertheless, some functionalities uses other frameworks to perform all tasks, i.e. to recover a process the ORTE daemon uses the BLCR component from Checkpoint/Restart Service (CRS).

As can be viewed in Table 4-5, the ORTE daemon can carry more than one protector functionality. Actually, this daemon does not perform any fault tolerance task. As well as has no component implementation to perform all other functionalities necessary to RADIC operation. Thus, we have implemented a new ORTE daemon named *protector daemon*.

Table 4-6 depicts the frameworks components that interact with the protector daemon. In item 5.2 we describe the implementations details of the protector daemon.

Table 4-6: Protector functionalities and Protector daemon.

Category	Functionality	ORTE Element and Framework/Component
Protection	— Store checkpoints	Protector daemon , FileM/RSH
	— Store message logs	Protector daemon , File M/RSH
Detection	— Send heartbeat	Protector daemon , RML/OOB
	— Watchdog	Protector daemon , RML/OOB
Recovery	— Recover a process	Protector daemon , CRS/BLCR
Reconfiguration	— Reestablish monitoring	Protector daemon , RML/OOB

Chapter 5

Implementation Details

This chapter discusses about the RADIC architecture implementation over Open MPI. As depicted in Chapter 4, here we present the implementation details. It is possible to identify the differences between the RADICMPI and the Open MPI implementations of the RADIC architecture.

Our effort is not concerned in present the differences between RADICMPI and the RADIC implementation for Open MPI. Furthermore, to use a structured framework in order to implement RADIC present few advantages. Besides the convenience to do not be aware about MPI details, Open MPI frameworks provide a set o functions and procedures that have helped us to achieve this RADIC implementation.

This chapter is divided in two parts. The first discusses about the three components implemented that combined performs the RADIC observer entity functionalities. The second part depicts the protector daemon. The protector daemon, while using RADIC fault tolerance, replaces the ORTE daemon.

5.1 Observer Functionalities

All observer functionalities are performed by three MCA components. These components are: *observer* – a Peer-to-peer Management Layer framework component, *single* – a Snapshot Coordinator framework

component, and finally *uncoord* – a Checkpoint/Restart Coordination Protocol framework component.

The communication between observer components and the protector daemon goes through the service channel named *Resource Management Layer* (RML). This service channel is implemented by the out-of-band component that is part of the RML framework. The RML provides to us an abstraction level. To use this communication channel have permitted to us the convenience to do not be aware about the communication details.

5.1.1 *observer* Component on PML Framework

The observer component has been implemented as a PML wrapper. It means that during start-up the MCA architecture will select the observer component and a real PML component. The common PML component used is the OB1, a high performance message layer management. Figure 5-1 depicts the functions call stack of observer PML wrapper combined with, *ob1* PML, *r2* BML and *tcp* BTL.

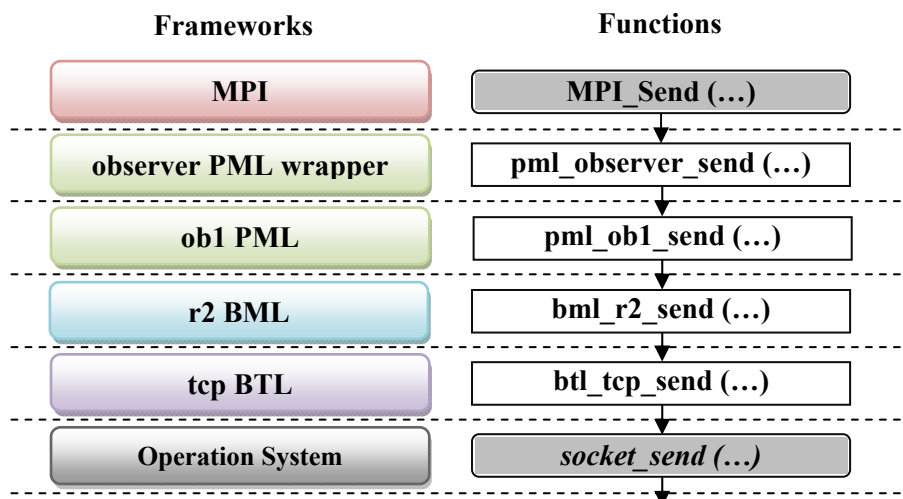


Figure 5-1: Typical function call stack for RADIC over Open MPI.

In Open MPI there is necessary to implement only the peer-to-peer communication functions because the collective functions always call a peer-to-peer function to deliver messages. As depicted in Figure 5-1, the user delivery request cross all communication layers. The message parameters and data are forwarded from top to bottom. Each layer remove, add or modify parameters according its necessity.

As discussed in item 3.2.5, the *crpcw* PML wrapper have a thread waiting for a signal to close BTL connections. This thread exists because BLCR do not take checkpoint from processes with open sockets or files. The *crpcw* wrapper has been incorporated into observer component. However, to implement RADIC we decided to do not create threads.

For better understand, as follow we discuss about each observer functionality.

Look for a Protector

The first task performed by the observer component is establishing a connection with its protector to be registered as an active observer. Actually, the pair observer/protector is defined as a numerical sequence, similar to RADICMPI method, i.e. *node0* is protected by *node1*, *node1* is protected by *node2*, etc.

Nevertheless, it is possible to implement other algorithms in order to choose a better protector, using the *cartography* framework which includes network knowledge or the *resource allocation subsystem* framework which can contribute with information about the resource availability on the nodes.

After a successfully component startup and registration, the protector sends a signal the observer requesting the first checkpoint. These routines take

place when the `MPI_Init` directive is called by the user application. Finally, after successfully checkpointing operation, the `MPI_Init` function completes its execution and the application can start to use other MPI directives.

Take and Send Message Logs

When a message arrives in observer PML wrapper, the receive functions (`recv` and `irecv`) start the logging operation. The logging operation consists in packing the message header and data and sends it to a protector through the service channel. After the message has been successfully received

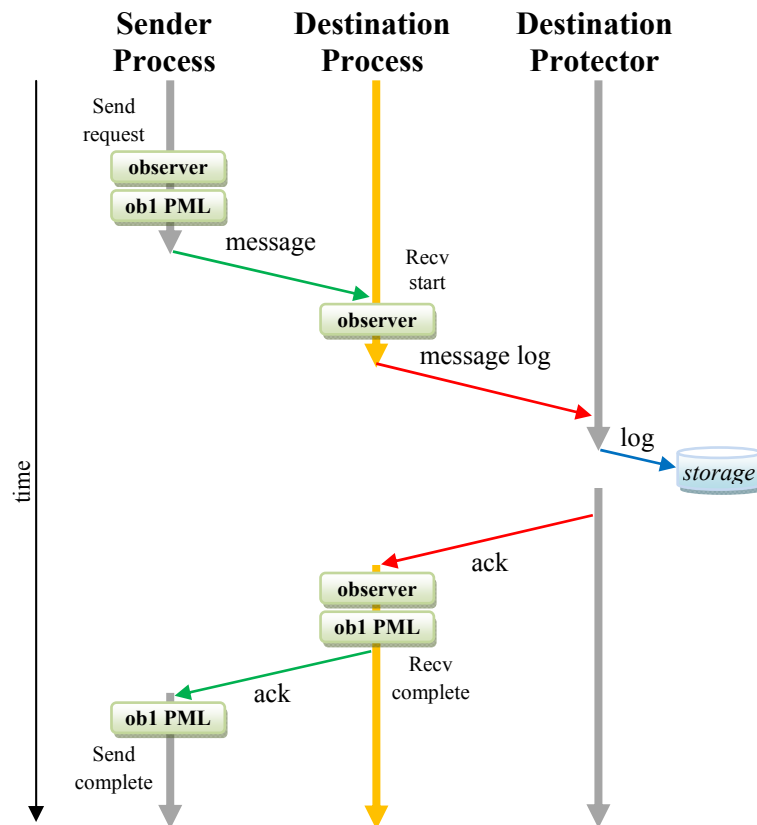


Figure 5-2: Logging operation performed by the observer PML wrapper. Green arrows indicate MPI communication. Red arrows indicate RML communication.

by the protector, the request is forwarded to the ob1 PML component. Figure 5-2 depicts this logging operation.

As shown in Figure 5-2, before the destination process receive the message, the observer PML wrapper send this message the message to its protector through RML. RML communication is represented by the red arrows, while MPI communication is represented by the green arrows and the storage process by the blue arrow.

Take and Send Checkpoints

The decision mechanism that starts a checkpoint operation resides in the protector daemon. However, the checkpointing procedure takes place on the observer. Furthermore, the checkpointing procedure depends on the single component from the SnapC framework as well as the uncoord component from CRCP framework. These components will be described in item 5.1.2 and 5.1.3 respectively.

Nevertheless, the observer PML wrapper contributes with the checkpointing operation. As explained in item 3.2.5, the crcpw PML wrapper, which was incorporated to observer PML wrapper, handles the checkpoint requests in order to starts two tasks as follows:

- a) The first task is the interlayer notification callback (INC). The interlayer notification callback cross all communication layers calling the `ft_event` function which closes active connections and prepare each communication component to take a checkpoint.
- b) The second task is start the inter process coordination. This coordination is performed by the CRCP framework which will be described in item 5.1.3.

Communication Fault Masking

In order to avoid a communication failure (MPI_COMM_ERROR), the observer PML wrapper queries the destination protector asking for its state. This query will start a recovery procedure by the protector if the process is not available. This recovery process will be described in item 5.2.1. Once the query has been performed, the observer retries the communication. This query goes through the service channel.

5.1.2 *single Component on SnapC Framework*

The single component of the Snapshot Coordinator has been derived from the full component. As depicted in item 3.2.5, there are three levels in the snapshot coordination: global, local and application coordination. The single component does not implement the global and local coordination. Only the application coordination is available. The application level coordination in fact does not coordinate processes. It just starts the checkpoint procedure.

The checkpoint procedure involves a lot of frameworks and services. This procedure occurs as shown:

- a) During component loading, the single component registers a callback function to handle SIG_USR1 signals;
- b) The checkpointing procedure starts when the single SnapC component receives a signal from its protector;
- c) Thus, single SnapC component modify a shared variable (OPAL_CR_STATUS_REQUESTED) informing that a checkpoint request has been arrived;

- d) The single SnapC component register a callback function in order to receive checkpointing data, such as the name and location of checkpoint data;
- e) The shared variable starts the observer PML wrapper checkpoint tasks which performs INC and the inter process coordination that lead with the checkpointing itself by the BLCR component of the CRS framework;
- f) After the BLCR component generates the checkpoint file, it call the callback function registered by the single SnapC component informing the name and location of checkpoint data;
- g) The information about checkpoint data is sent for the protector daemon that will be aware with file transferring.
- h) The checkpoint procedure finishes with the correctly file transferring.

5.1.3 *uncoord* Component on CRCP Framework

The *uncoord* component of the Checkpoint/Restart Coordination Protocol has been derived from the *coord* component.

The *coord* component, originally, manages asynchronous message requests draining uncompleted deliveries from the PML and reposting those requests after a checkpointing completion. However, the *uncoord* module does not perform this routine because messages requests can be managed by the message logging procedure performed by the observer PML wrapper. The consistent state is assured by the message logging protocol.

Thus, the uncoord component just response to queries received by external programs used to monitor the fault tolerance status. Clearly, this component is not necessary for the RADIC architecture. However, we keep this component in our implementation to maintain the original fault tolerance structure existent in Open MPI.

5.2 Protector Functionalities

All protector functionalities are performed by the protector daemon. The communication between observer components and the protector daemon goes through RML. The protector daemon derives from the ORTE daemon.

5.2.1 Protector Daemon

There are two classes of functionalities which protector daemon performs. The first class is composed by the active functionalities such *send heartbeat*, *watchdog* and *request a checkpoint*. The second class is composed by passive functionalities, which are launched by an observer request.

When the protector daemon is launched it performs three tasks: start the watchdog timer, start the checkpointing timer and configure the protection chain. The first two tasks will be detailed in specific topics below. The third task concerns with the observer/protector pair definition. As well as the observer PML wrapper, the pair observer/protector is defined as a numerical sequence, similar to RADICMPI method, i.e. *node0* is protected by *node1*, *node1* is protected by *node2*, etc.

Nevertheless, it is possible to implement other algorithms in order to choose a better protector, using the *cartography* framework which includes

network knowledge or the *resource allocation subsystem* framework which can contribute with information about the resource availability on the nodes.

In order to respond to observer requests and performs protector inherent functionalities, in the protector daemon, we have used the ORTE daemon command processor. The ORTE daemon command processor is a routine that verify for incoming requests in order to process those requests.

We have defined four new commands. These commands are as follows:

Table 5-1: New ORTE commands for protector daemon.

Command	Functionality Associated
ORTE_DAEMON_HEARTBEAT_CMD	— Receive and respond to a heartbeat generated by other protector daemon
ORTE_DAEMON_WRITE_LOG_CMD	— Receive and write on the message log file a message header and data sent by an observer PML wrapper
ORTE_DAEMON_TRANSFER_DIR_CMD	— Transfer a directory containing checkpoint data from a node where a protected application process is running
ORTE_DAEMON_VERIFY_STATUS_CMD	— Verify the status of a protected application process

The RADIC protector functionalities where have been implemented as a set commands and functions launched at startup or on demand by the command processor. We detail these commands and functions below, according the functionality associated to it.

Store Checkpoints

In order to store checkpoints, the protector registers a command (ORTE_DAEMON_TRANSFER_DIR_CMD). The single Snapshot Coordinator component calls this command after a checkpointing procedure.

The protector handles this command calling the File Manager framework in order to transfer the directory which contains the checkpoint data and registering a callback function. The callback function called after the directory transfer simply deletes the checkpoint data directory from the observer.

Store Message Logs

During the observer PML wrapper operation, it sends message log requests to the protector packing a message log command request (ORTE_DAEMON_WRITE_LOG_CMD) attached to message header and data.

When a log request (ORTE_DAEMON_WRITE_LOG_CMD) arrives on the protector command processor, the routine associated to this command executes the steps as follows:

- a) Unpack the message header;
- b) Calculate the message data size;
- c) Allocate memory according the data size;
- d) Unpack the message data;
- e) Open message log file;
- f) Write on message log file;
- g) Close the message log file;
- h) Free the memory previously allocated.

The observer PML wrapper uses a synchronous RML send, it assures that the observer will be waiting for the correctly message log before forward the message to the application process.

Send Heartbeats

During startup, the protector daemon defines a timer using the Open MPI common framework. This timer is configured using the heartbeat interval parameter (`orte_hb_interval`). While the protector sets a timer, it registers a callback function which will be launched if the timer expires.

The Open MPI framework handles the timer calling the callback function. The callback function, clearly, is a `send_heartbeat` function.

The `send_heartbeat` function packs a command request (`ORTE_DAEMON_HEARTBEAT_CMD`) and sends it to the heartbeat destination protector through RML. In order to verify the correctly receipt, the protector users a synchronous RML send. Thus, if an error occurs while sending the heartbeat the protector retries the communication before call an application process verification function.

When a heartbeat command (`ORTE_DAEMON_HEARTBEAT_CMD`) request arrives on the protector command processor, the routine associated to this command simply redefines the expiration time of the watchdog timer to the default value configured by the user as a MCA parameter.

Watchdog

During startup, the protector daemon defines a timer using the Open MPI common framework. This timer is configured using the watchdog expire time MCA parameter (`orte_wd_interval`). While the protector sets a timer, it registers a callback function which will be launched if the timer expires.

The Open MPI framework handles the timer calling the callback function, which is an application process verification function, when the timer expires. After process verification, the watchdog timer is configured again.

Recover a Process

The application process recovery in this RADIC implementation, as well as the RADIC architecture, always starts with the application process status verification. The verification procedure could be called by three routines: an observer which receives a `MPI_COMM_ERROR` while trying to send a message, watchdog timer expiration and consecutive heartbeat sending failure. The verification starts when a command request (`ORTE_DAEMON_VERIFY_STATUS_CMD`) arrives on the protector command processor.

During the verification, the failure could be confirmed. Thus, a process recovery procedure is called. Recover a process in RADIC implementation over Open MPI is a simple task. The recovery procedure just calls a BLCR CRS component passing the checkpoint data file as a parameter. The BLCR handles the request launching the application process, and the observer component which is attached to it. Thus, the observer component process the message log stored on the same machine which it revival and continues the computation.

Reestablish Monitoring

After the watchdog timer expiration or consecutive heartbeat failures, the protector daemon executes the same algorithm used to define the observer/protector pairs in order to define its new protector.

Once a new protector has been defined, the protector daemon restarts the heartbeat sending procedure. This routine reestablishes the monitoring mechanism between protectors.

Because the watchdog routine is a passive routine, which reacts to heartbeats, no changes need to be performed at this side.

Chapter 6

Validating and Experimenting the Implementation

In this chapter we present the experiments used to validate the implementation. Furthermore, some performance analyses are made in order to evaluate the principals elements if the implementation which incurs in overhead.

For better comprehension, some explanations about the NAS Parallel Benchmarks are made. Thus important variables are clarified about this benchmark execution. The state size, or process size and the communication pattern of each benchmark were analyzed. To know these variables permit a consistent analysis of our Open MPI RADIC implementation behavior.

6.1 NAS Parallel Benchmark

The NAS Parallel Benchmarks (NPB) is widely used to evaluate parallel machines (Wechsler, 1995). Those reports provide a comparison of execution time as a function of the number of processors, from which execution rate, speedup, and efficiency are easily computed (Wong, et al., 1999).

While extremely valuable, these results only provide an understanding of overall delivered performance. The fixed algorithm and standard Message Passing Interface – MPI (Forum, 1993), programming model of NAS Parallel Benchmarks (Bailey, et al., 1995) make it possible to use these benchmarks as a basis for an in-depth comparative analysis of parallel architectures. However, the current reports still provide only a crude performance comparison

The first step in understanding NAS benchmarks behavior is to isolate the components of application execution time. Of course, these are parallel programs, so we need to work with the time spent on all processors. The numbers presented in this master thesis shows the sum of the execution time over all processors of the NAS benchmarks on our machines, as a function of number of processors.

In this work we have used NAS Parallel Benchmarks version 3.2.1 for MPI. In NPB each benchmark has different classes that define the problem size. We have used class *A*, *B* and *C*, while available. Table 6-1 presents the problem size of each benchmark in function of the class.

Table 6-1: NAS Parallel Benchmark Problem Size (Bailey, et al., 1995)

Benchmark	Class A	Class B	Class C
BT	64^3	102^3	162^3
SP	64^3	102^3	162^3
LU	64^3	102^3	162^3
IS	2^{23}	2^{25}	2^{27}
MG	256^3	256^3	512^3
FT	$256^3 \times 128$	512×256^3	512^3
CG	14000	75000	150000

The problem size is an important variable in parallel computing because it defines the computation time. An application running in serial

mode, its execution time is defined by the problem size. Furthermore, in parallel applications the communication changes the execution time. Thus, calculate the execution time depends on the communication pattern.

Furthermore, Table 6-2, Table 6-3 and Table 6-4 present the process size in memory in function of the number of processors, values are expressed in megabytes. This information is very important to the RADIC architecture because checkpointing operation generate file that has the same size than the process state.

Table 6-2: Process size for NPB benchmarks class A.

# of Processors	BT	SP	IS	LU	MG	FT	CG
1	114 MB	94 MB	155 MB	58 MB	455 MB	467 MB	70 MB
2	n/a	n/a	81 MB	35 MB	248 MB	238 MB	48 MB
4	39 MB	34 MB	45 MB	22 MB	138 MB	123 MB	25 MB
8	n/a	n/a	31 MB	16 MB	66 MB	66 MB	17 MB
9	24 MB	22 MB	n/a	n/a	n/a	n/a	n/a
16	19 MB	18 MB	24 MB	13 MB	33 MB	38 MB	14 MB
25	15 MB	15 MB	n/a	n/a	n/a	n/a	n/a
32	n/a	n/a	20 MB	11 MB	25 MB	24 MB	11 MB

Table 6-3: Process size for NPB benchmarks class B.

# of Processors	BT	SP	IS	LU	MG	FT	CG
1	417 MB	342 MB	598 MB	196 MB	455 MB	+1 GB	446 MB
2	n/a	n/a	303 MB	106 MB	248 MB	927 MB	234 MB
4	125 MB	109 MB	155 MB	59 MB	138 MB	468 MB	132 MB
8	n/a	n/a	81 MB	36 MB	67 MB	238 MB	82 MB
9	67 MB	55 MB	n/a	n/a	n/a	n/a	n/a
16	40 MB	37 MB	51 MB	23 MB	39 MB	124 MB	56 MB
25	31 MB	28 MB	n/a	n/a	n/a	n/a	n/a
32	n/a	n/a	33 MB	17 MB	25 MB	67 MB	36 MB

Table 6-4: Process size for NPB benchmarks class C.

# of Processors	BT	SP	IS	LU	MG	FT	CG
1	+1 GB	+1 GB	+1 GB	728 MB	n/a	n/a	+1 GB
2	n/a	n/a	+1 GB	378 MB	+1 GB	n/a	582 MB
4	436 MB	371 MB	598 MB	197 MB	920 MB	+1 GB	316 MB
8	n/a	n/a	307 MB	109 MB	455 MB	927 MB	168 MB
9	212 MB	176 MB	n/a	n/a	n/a	n/a	n/a
16	127 MB	110 MB	162 MB	61 MB	236 MB	468 MB	100 MB
25	90 MB	80 MB	n/a	n/a	n/a	n/a	n/a
32	n/a	n/a	93 MB	39 MB	126 MB	239 MB	55 MB

Due a memory limitation in the cluster used to run these experiments, application processes with more than 1 gigabyte of memory where discarded.

Communication pattern is a very important variable to RADIC operation, because the logging procedure performance depends on it. In Table 6-5 we depict communication types and message size for each NAS benchmark while it is running in sixteen parallel processors (Faraj, et al., 2002).

Naturally, the number of messages exchanged between processes and size of these messages vary according two factors: the problem size and the number of nodes used. There is no equation that can calculate these variations. It depends on the application implementation.

For better comprehension, in Table 6-5, a static behavior means that the number of these messages does not change according problem size. Nevertheless there is no guarantee that it will not change according the number of processors. On the other hand a dynamic behavior means that the number of messages varies according to problem size.

Table 6-5: NPB communication pattern while using sixteen processors.

Benchmark	Type	Behavior	# of messages	Size	Volume
BT	Point-to-point	Static	0	0	0
		Dynamic	77280	191.32 KB	14.1 GB
	Collective	Static	7	1.58 KB	11.1 KB
		Dynamic	0	0	0
SP	Point-to-point	Static	0	0	0
		Dynamic	154080	161.29 KB	23.7 GB
	Collective	Static	7	1.58 KB	11.1 KB
		Dynamic	0	0	0
LU	Point-to-point	Static	0	0	0
		Dynamic	756132	7.25 KB	5.23 GB
	Collective	Static	18	1.59 KB	28.6 KB
		Dynamic	0	0	0
IS	Point-to-point	Static	15	48 Bytes	720 Bytes
		Dynamic	0	0	0
	Collective	Static	22	63.77 KB	1.37 MB
		Dynamic	11	31.45 MB	346 MB
MG	Point-to-point	Static	0	0	0
		Dynamic	11024	35.67 KB	384 MB
	Collective	Static	100	1.26 KB	126 KB
		Dynamic	0	0	0
FT	Point-to-point	Static	0	0	0
		Dynamic	0	0	0
	Collective	Static	3	338 Bytes	0.99 KB
		Dynamic	8	482.56 MB	3.77 GB
CG	Point-to-point	Static	0	0	0
		Dynamic	47104	49.86 KB	2.24 GB
	Collective	Static	1	38 Bytes	38 Bytes
		Dynamic	0	0	0

A complete description of each benchmark, including its communication behavior, can be found in Appendix II.

6.2 Experiments Plan and Environment

In order to test the implementation we have used the NAS Parallel Benchmarks. The benchmarks utilized was BT, SP, LU, IS, MG, FT and CG.

We have selected these benchmark because their characteristics. The principal characteristics are discussed in next paragraphs.

BT, LU and SP have similar operation: uses much point-to-point communication, meanwhile the message size varies. BT and SP send medium messages sizes and LU sends small messages. These three benchmarks have significant processing time, thus, the communication time is overlapped with the computation. The main difference between BT and SP is the number of messages.

IS and FT has similar communication characteristic: small number of large collective messages. Meanwhile, the message size have different magnitude order, IS sends large messages and FT sends enormous messages. Furthermore, FT needs more computation time than IS.

CG and MG are very similar. Both have point-to-point communication with medium message size. Furthermore, these message are overlapped with computation, meanwhile, there are very sensible to the message latency.

All benchmarks where executed in same cluster. The machine used to run these experiments was a 32-node Linux cluster. Each node is equipped with 1 gigabyte of SDRAM, and one Intel® Pentium® 4 processor running at 3 GHz.

This cluster has a switched single Gigabit Ethernet networks as interconnection network. To launch parallel applications this cluster uses Sun Grid Engine queue manager that is not part of the thirty two nodes where runs applications.

As operating system we have used Linux 2.6.17-1.2142, compiled with GNU C Compiler version 4.0.2. This compiler has been used to compile all

other packages such Open MPI with RADIC capabilities, BLCR and NPB. An exception was made for FORTRAN written benchmarks where we used GNU FORTRAN version 4.0.2. The Open MPI trunk version used was 1.3-Alpha1 release 18675. All experiments save output data to home directory exported to all nodes through NFS.

We have executed a hundred times each combination of number of processors and benchmark application. Furthermore, we assure that the standard deviation is fewer than 10% in worst case. However it is, on most cases, under 2%.

6.3 Validation Experiments

In order to test the RADIC implementation we have executed a set of experiments in different contexts. Three contexts were defined: message delivery and logging mechanism, checkpointing operation and recovery procedure.

For a better comprehension of the execution log generated by our implementation, Figure 6-1 depicts the internal process identification.

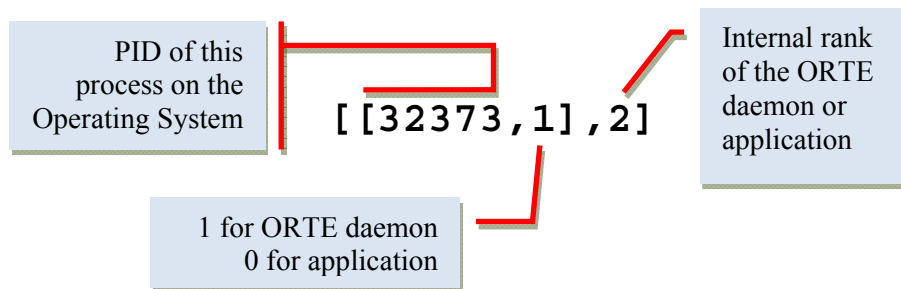


Figure 6-1: Internal process identification example.

Message Delivery and Message Logging

To test the message delivery and logging mechanism we have used a ping program, which forwards a dummy message (1 MB) across the cluster nodes. We chose this program because it provides a controlled environment to test the message passing mechanism. Just one message can leave and arrive on the message passing library at a time. Thus we can trace the calls involved in the communication layers stack. The source code of this program can be analyzed in Appendix III.

The Figure 6-2 depicts an execution log of the ping program. This execution runs with 4 nodes. The yellow mark represents a logging procedure, performed by the message receiver and its protector. In this log, the process 2,

```
[fialho@aogrdini]$ mpirun -am radic -hostfile host -d -np 4 ping 1 1
(1) sent token to (2)
[aogrd02] Sending message log to [[7731,0],3]
[aogrd03] orte:daemon:cmd:processor called by [[7731,1],2] for tag 1
[aogrd03] protectord_log: writing HEADER (1 element of 24 bytes)
[aogrd03] protectord_log: writing BUFFER (1048576 element of 1 bytes)
[aogrd03] orte:daemon:cmd:processor: processing commands completed
(2) sent token to (3)
[aogrd03] Sending message log to [[7731,0],4]
[aogrd04] orte:daemon:cmd:processor called by [[7731,1],3] for tag 1
[aogrd04] protectord_log: writing HEADER (1 element of 24 bytes)
[aogrd04] protectord_log: writing BUFFER (1048576 element of 1 bytes)
[aogrd04] orte:daemon:cmd:processor: processing commands completed
(3) sent token to (4)
[aogrd04] Sending message log to [[7731,0],1]
[aogrd01] orte:daemon:cmd:processor called by [[7731,1],4] for tag 1
[aogrd01] protectord_log: writing HEADER (1 element of 24 bytes)
[aogrd01] protectord_log: writing BUFFER (1048576 element of 1 bytes)
[aogrd01] orte:daemon:cmd:processor: processing commands completed
(4) sent token to (1)
[aogrd01] Sending message log to [[7731,0],2]
[aogrd02] orte:daemon:cmd:processor called by [[7731,1],1] for tag 1
[aogrd02] protectord_log: writing HEADER (1 element of 24 bytes)
[aogrd02] protectord_log: writing BUFFER (1048576 element of 1 bytes)
[aogrd02] orte:daemon:cmd:processor: processing commands completed
Total time: 4.01893
orterun: exiting with status 0
[fialho@aogrdini]$
```

Figure 6-2: Debug log of a ping program showing the message logging mechanism.

running on node `aogrd02` receives a message from the process 1, running on node `aogrd01`. Thus, the observer PML wrapper locate in the message passing library attached to process 2 starts the logging procedure sending a copy of the received message to its protector, located on the node `aogrd03`. The protector running on node `aogrd03` receives the message log and writes it on the log file.

In order to test the correctly message logging operation Figure 6-3 depicts the contents of the `radic.log` file on node `aogrd03`. The `logreader` program is part of the RADIC architecture implementation for Open MPI. This program permits the verification of the `radic.log` contents, showing to the user the message header and, optionally, the message buffer.

```
[fialho@aogrd03]$ logreader -v -f radic.log
Reading message 0 HEADER (24) bytes - [24]
Reading message 0 BUFFER (1048576) bytes - [1048576 * 1]
-----
                        MESSAGE 0 START
-----
Message from: 0      Message TAG: 1      Communicator: 0
Datatype Size: 1     Datatype ID: 4      Data Count:  1048576
-----
                        MESSAGE 0 END
-----
!!!!!!!!!!!!!! (...)
-----
Total of messages: 1    Total of bytes: 1048576
[fialho@aogrd03]$
```

Figure 6-3: `radic.log` printed by `logreader`.

Furthermore, in order to test the message integrity we have used the NAS BT application. Any application which have fixed values and verify the computation could be used. Figure 6-4 present the report generated by the NAS BT application.

Thus, after a verification of these three experiments, we can assure that the message delivery and message logging has been correctly implemented. This confirmation comes during the stress test. To stress the message passing mechanism we choose the NAS LU application.

```

Verification being performed for class A
accuracy setting for epsilon = 0.10000000000000E-07
Comparison of RMS-norms of residual
  1 0.1080634671464E+03 0.1080634671464E+03 0.7101254243319E-14
  2 0.1131973090122E+02 0.1131973090122E+02 0.1098479988995E-14
  3 0.2597435451158E+02 0.2597435451158E+02 0.3282665917769E-14
  4 0.2366562254468E+02 0.2366562254468E+02 0.6605336562438E-14
  5 0.2527896321175E+03 0.2527896321175E+03 0.1574051628196E-13
Comparison of RMS-norms of solution error
  1 0.4234841604053E+01 0.4234841604053E+01 0.6291936058601E-15
  2 0.4439028249700E+00 0.4439028249700E+00 0.4751994418892E-14
  3 0.9669248013635E+00 0.9669248013635E+00 0.1837119946552E-14
  4 0.8830206303977E+00 0.8830206303977E+00 0.1257301343147E-15
  5 0.9737990177083E+01 0.9737990177083E+01 0.7661435871068E-14
Verification Successful

```

Figure 6-4: Report generated by the NAS BT application.

According Table 6-5, LU application has high volume of small messages. To show the results of stress test we cannot depict an execution log, because it very long. Thus we have used the logreader application with the summarize option, which generate only a message summary containing the number of messages and the volume of these messages in bytes, as shown in Figure 6-5.

```

lun jul 3 05:01:26 CEST 2008
Total of messages: 76987      Total of bytes: 378509872
lun jul 3 05:01:37 CEST 2008
Total of messages: 80804      Total of bytes: 397227632
lun jul 3 05:01:48 CEST 2008
Total of messages: 84781      Total of bytes: 416201392
lun jul 3 05:01:59 CEST 2008
Total of messages: 88841      Total of bytes: 435307952
lun jul 3 05:02:10 CEST 2008
Total of messages: 92605      Total of bytes: 453940912
lun jul 3 05:02:21 CEST 2008
Total of messages: 96525      Total of bytes: 472823472

```

Figure 6-5: Summarized output generated by the logreader.

Figure 6-5 depicts the high volume of messages exchanged by one process of the NAS LU application. The yellow marks represent an interval smaller than one minute. In this interval 19.538 messages have been received by the process which this protector daemon protects, which represent 90 megabytes approximately.

Checkpointing Operation

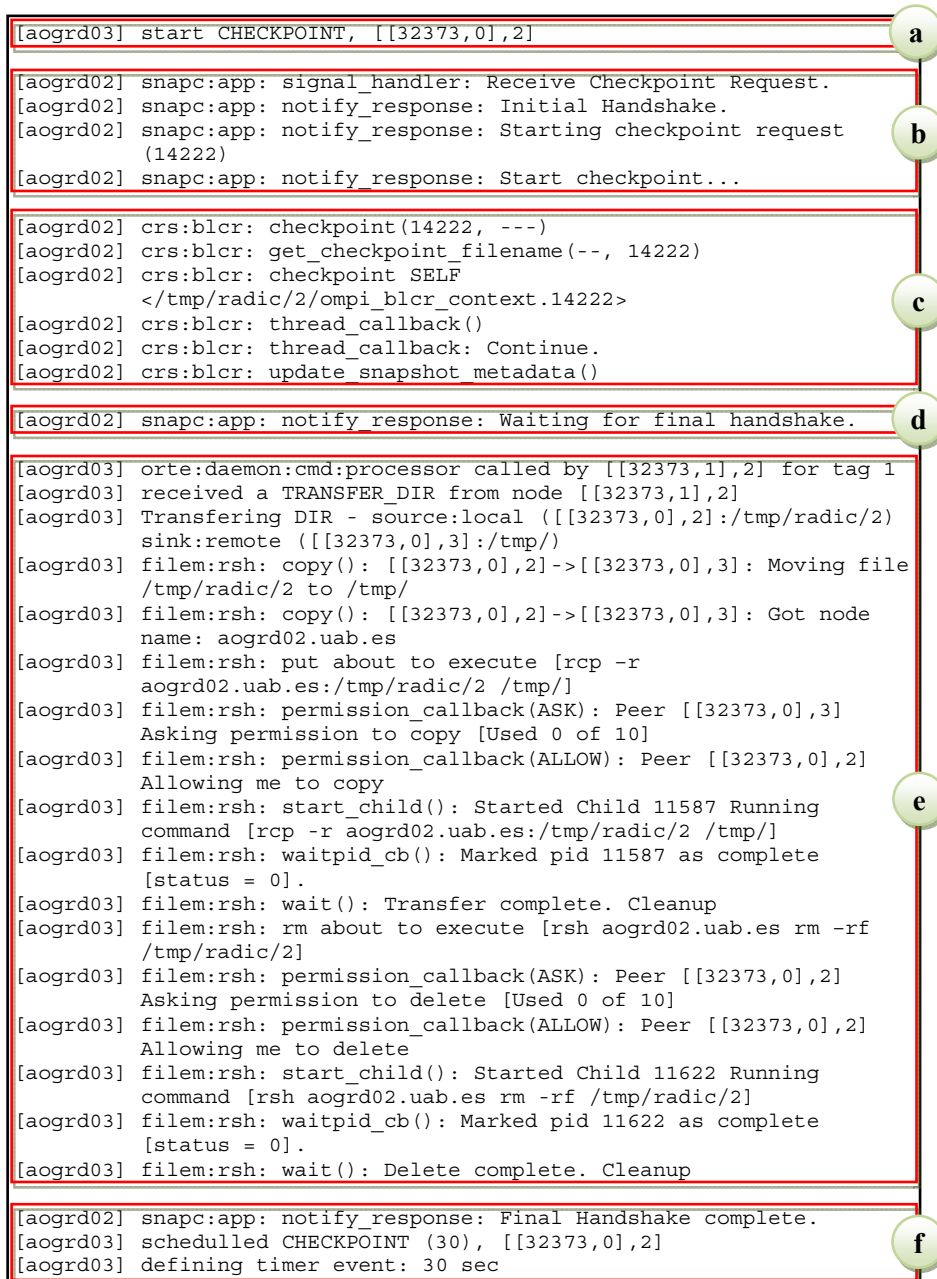
To test the checkpointing operation we have used the ping program described above. We configured RADIC to take a checkpoint using a 30 seconds interval, and an infinite watchdog and heartbeat interval. Thus we can analyze the debug execution log in order to verify the correctly execution of the checkpointing procedure.

The debug execution log is depicted in Figure 6-6. The yellow marks represent the checkpointing operation activities which occurs on the application process and its protector daemon. As we can see, there are six groups of registers, as follows:

- a) Checkpointing request;
- b) Interlayer notification callback function started;
- c) CRS BLCR component operation;
- d) single SnapC component request a file transfer;
- e) File transfer;
- f) Finalization and scheduling of a new checkpoint.

The correct execution of these steps assures the correct interaction of the framework components involved in the checkpointing procedure. In order

to verify the integrity of the checkpoint file, we have used the recovery procedure described below.



```

[aogrd03] start CHECKPOINT, [[32373,0],2]
[aogrd02] snapc:app: signal_handler: Receive Checkpoint Request.
[aogrd02] snapc:app: notify_response: Initial Handshake.
[aogrd02] snapc:app: notify_response: Starting checkpoint request (14222)
[aogrd02] snapc:app: notify_response: Start checkpoint...
[aogrd02] crs:blcr: checkpoint(14222, ---)
[aogrd02] crs:blcr: get_checkpoint_filename(--, 14222)
[aogrd02] crs:blcr: checkpoint SELF
[</tmp/radic/2/ompi_blcr_context.14222>
[aogrd02] crs:blcr: thread_callback()
[aogrd02] crs:blcr: thread_callback: Continue.
[aogrd02] crs:blcr: update_snapshot_metadata()
[aogrd02] snapc:app: notify_response: Waiting for final handshake.
[aogrd03] orte:daemon:cmd:processor called by [[32373,1],2] for tag 1
[aogrd03] received a TRANSFER_DIR from node [[32373,1],2]
[aogrd03] Transferring DIR - source:local ([[32373,0],2]:/tmp/radic/2)
sink:remote ([[32373,0],3]:/tmp/)
[aogrd03] filem:rsh: copy(): [[32373,0],2]->[[32373,0],3]: Moving file
/tmp/radic/2 to /tmp/
[aogrd03] filem:rsh: copy(): [[32373,0],2]->[[32373,0],3]: Got node
name: aogrd02.uab.es
[aogrd03] filem:rsh: put about to execute [rcp -r
aogrd02.uab.es:/tmp/radic/2 /tmp/]
[aogrd03] filem:rsh: permission_callback(ASK): Peer [[32373,0],3]
Asking permission to copy [Used 0 of 10]
[aogrd03] filem:rsh: permission_callback(ALLOW): Peer [[32373,0],2]
Allowing me to copy
[aogrd03] filem:rsh: start_child(): Started Child 11587 Running
command [rcp -r aogrd02.uab.es:/tmp/radic/2 /tmp/]
[aogrd03] filem:rsh: waitpid_cb(): Marked pid 11587 as complete
[status = 0].
[aogrd03] filem:rsh: wait(): Transfer complete. Cleanup
[aogrd03] filem:rsh: rm about to execute [rsh aogrd02.uab.es rm -rf
/tmp/radic/2]
[aogrd03] filem:rsh: permission_callback(ASK): Peer [[32373,0],2]
Asking permission to delete [Used 0 of 10]
[aogrd03] filem:rsh: permission_callback(ALLOW): Peer [[32373,0],2]
Allowing me to delete
[aogrd03] filem:rsh: start_child(): Started Child 11622 Running
command [rsh aogrd02.uab.es rm -rf /tmp/radic/2]
[aogrd03] filem:rsh: waitpid_cb(): Marked pid 11622 as complete
[status = 0].
[aogrd03] filem:rsh: wait(): Delete complete. Cleanup
[aogrd02] snapc:app: notify_response: Final Handshake complete.
[aogrd03] scheduled CHECKPOINT (30), [[32373,0],2]
[aogrd03] defining timer event: 30 sec

```

Figure 6-6: Debug log representing a checkpointing procedure.

Recovery Procedure

There are three possible ways to start the procedure for application status verification, which results in the activation of the recovery procedure: *watchdog expires*, *consecutives heartbeat timeout* and *communication failure detected by an observer*. These possibilities are presented in Table 6-6.

Table 6-6: Failures which starts the procedure for application status verification.

Failure on	Detection Mechanism
Protector which send heartbeats to other protector	Watchdog expires
Protector which answers to heartbeat reception	Consecutives heartbeat timeout
MPI message destination (application process)	MPI communication failure

Unfortunately, the RADIC implementation for Open MPI does not have a failure injection framework. Thus, to generate the recovery scenario, we have killed processes to force the activation of the procedure for application status verification. The three possible scenarios are depicted in Figure 6-7.

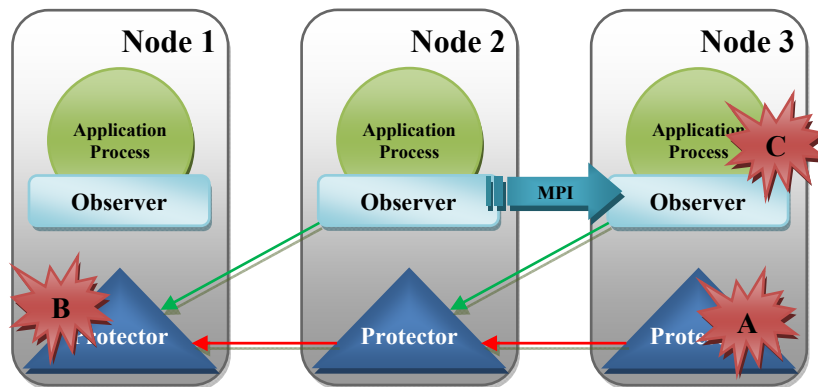


Figure 6-7: Failures which start the procedure for application status verification on node N.

The analysis of the execution log for each scenario permits the verification of the correct fault detection and consequently the status verification procedure calling. Figure 6-9 depicts fault detected by protectors. In contrast, Figure 6-8 depicts a normal operation.

In Figure 6-8, the registers entries grouped with the mark (a) depict the startup routine performed by each protector daemon. The group (b) is a normal operation while protector are sending and receiving heartbeats normally.

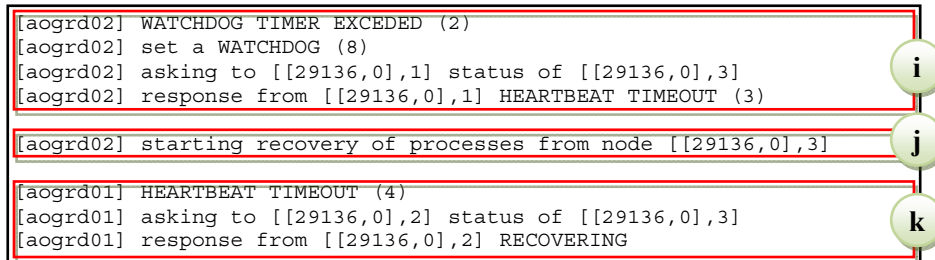
```
[aogrd02] HB_source [[29136,0],3], HB_dest [[29136,0],1], INTERVAL: 5
[aogrd02] scheduling a HEARTBEAT (5)
[aogrd02] defining timer event: 5 sec
[aogrd02] start a WATCHDOG (8)
[aogrd02] defining timer event: 8 sec
[aogrd01] HB_source [[29136,0],2], HB_dest [[29136,0],3], INTERVAL: 5
[aogrd01] scheduling a HEARTBEAT (5)
[aogrd01] defining timer event: 5 sec
[aogrd01] start a WATCHDOG (8)
[aogrd01] defining timer event: 8 sec
[aogrd03] HB_source [[29136,0],1], HB_dest [[29136,0],2], INTERVAL: 5
[aogrd03] scheduling a HEARTBEAT (5)
[aogrd03] defining timer event: 5 sec
[aogrd03] start a WATCHDOG (8)
[aogrd03] defining timer event: 8 sec

[aogrd02] sent a HEARTBEAT to node [[29136,0],1]
[aogrd02] scheduling a HEARTBEAT (5)
[aogrd02] defining timer event: 5 sec
[aogrd01] sent a HEARTBEAT to node [[29136,0],3]
[aogrd01] scheduling a HEARTBEAT (5)
[aogrd01] defining timer event: 5 sec
[aogrd01] received a HEARTBEAT from node [[29136,0],2]
[aogrd01] set a WATCHDOG (8)
[aogrd03] sent a HEARTBEAT to node [[29136,0],1]
[aogrd03] scheduling a HEARTBEAT (5)
[aogrd03] defining timer event: 5 sec
[aogrd02] received a HEARTBEAT from node [[29136,0],3]
[aogrd02] set a WATCHDOG (8)
[aogrd02] received HEARTBEAT confirmation from node [[29136,0],1]
[aogrd03] received HEARTBEAT from node [[29136,0],1]
[aogrd03] set a WATCHDOG (8)
[aogrd01] received HEARTBEAT confirmation from node [[29136,0],3]
[aogrd03] received HEARTBEAT confirmation from node [[29136,0],2]
```

Figure 6-8: Execution log of a normal heartbeat watchdog operation.



Figure 6-9: Debug log representing the status verification procedure.



```

[aogrd02] WATCHDOG TIMER EXCEDED (2)
[aogrd02] set a WATCHDOG (8)
[aogrd02] asking to [[29136,0],1] status of [[29136,0],3]
[aogrd02] response from [[29136,0],1] HEARTBEAT TIMEOUT (3)

[aogrd02] starting recovery of processes from node [[29136,0],3]

[aogrd01] HEARTBEAT TIMEOUT (4)
[aogrd01] asking to [[29136,0],2] status of [[29136,0],3]
[aogrd01] response from [[29136,0],2] RECOVERING

```

Figure 6–9 (cont.): Debug log representing the status verification procedure.

Figure 6-9 depicts the debug log of an execution where a failure has been injected in the protector resident in node aogrd03. The Open MPI architecture automatically kills the application processes if the ORTE daemon, represented by the protector daemon in the RADIC implementation, is not running. It occurs because the parallel environment no exists anymore when the ORTE daemon stops.

According the debug log represented in Figure 6-9, two protectors have detected the failure: the protector resident in nodes aogrd01 and aogrd02. It occurs because this execution uses 3 nodes. A complete analysis of this log is presents as follows:

- a) Startup routine execute by nodes aogrd01, aogrd02, and aogrd03. After this, a fault has been injected in node aogrd03.
- b) In continuation, nodes aogrd01 and aogrd02 sent heartbeats. However node aogrd02 does not receive a heartbeat, and node aogrd01 does not receive a confirmation;
- c) Node aogrd01 detects a heartbeat timeout and asks to aogrd02 in order to verify if it has detected a fault. Furthermore node aogrd02 answers in the negative because it does not have detect a watchdog expiration yet;

- d) The node `aogrd02` watchdog has expired. Thus, it asks to node `aogrd01` in order to verify if it has detected a fault. Node `aogrd01` answers that it has detected one heartbeat timeout;
- e) Another turn of heartbeats on the system;
- f) Node `aogrd01` detects the second heartbeat timeout and asks to node `aogrd02` to verify if it has detected a fault. Node `aogrd02` confirms a fault. Nevertheless, it does not start the recovery procedure because it is just one watchdog expiration and two heartbeat timeout;
- g) One more turn of heartbeats on the system;
- h) Node `aogrd01` detects the third heartbeat timeout and asks to node `aogrd02` to verify if it has detected a fault. Node `aogrd02` confirms a fault. Nevertheless, it does not start the recovery procedure because it is just one watchdog expiration and three heartbeat timeout. There are necessary at least 3 heartbeat timeouts and 2 watchdog expirations;
- i) The node `aogrd02` watchdog has expired again. It asks to node `aogrd01` in order to verify if it has detected a fault. Node `aogrd01` answers that it has detected three heartbeat timeouts. It is enough to start a recovery procedure;
- j) Node `aogrd02` starts the recovery procedure;
- k) Node `aogrd01` detects the fourth heartbeat timeout and asks to node `aogrd02` to verify if it has detected a fault. Node `aogrd02` answers that it is recovering the failed application process.

As we can see, the failure detection using the heartbeat mechanism is very slow. To accelerate this mechanism, a possible solution is decrease the heartbeat and watchdog cycle. Thus, RADIC has a failure detection mechanism in the observer which, normally, is faster than the heartbeat/watchdog mechanism.

Unfortunately, due the high log volume generated while debugging the message passing library we cannot depict here the observer fault detection. Furthermore, we can depict the protector reception of a verification status request. This mechanism recovers the process immediately because the observer retries the communication three times. In Figure 6-10 we present this operation. We have used a NAS BT application to represent this failure detection.

```
[aogrd02] recovery request from process [[29136,1],1] to [[29136,1],3]
[aogrd02] starting recovery of process [[29136,1],3]
[aogrd02] recovery request from process [[29136,1],2] to [[29136,1],3]
```

Figure 6-10: Debug log of a protector daemon representing a recovery request from an application process.

As shown in Figure 6-10, node aogrd02 receives a recovery request from process application with rank 1 and 2 to recover application process 3. The first recovery request received starts the recovery procedure, and the second request is discarded because a recovery is already running. This

```
[aogrd02] starting recovery of process [[29136,1],3]
[aogrd02] process [[29136,1],3] is <mpi_bldr_context.15825>
[aogrd02] crs:blcr: restart(15825, ---)
[aogrd02] crs:blcr: get_checkpoint_filename(--, 15825)
[aogrd02] crs:blcr: checkpoint SELF
[aogrd02] </tmp/radic/3/mpi_bldr_context.15825>
[aogrd02] crs:blcr: thread_callback()
[aogrd02] crs:blcr: thread_callback: Continue.
[aogrd02] process [[29136,1],3] recovered
```

Figure 6-11: Debug log of a protector daemon recovering an application process

recovery procedure will recover only the process application 3, not all application processes located in node `aogrd03`.

After a successfully failure detection the recovery procedure is executed. The debug log generated by this procedure is depicted in Figure 6-11.

6.4 Evaluation Experiments

In order to test the RADIC implementation we have executed the NAS Parallel Benchmarks in different contexts, changing the number of nodes used in computation and changing the problem size. We have defined two evaluation experiments.

The first group of experiments is guided to evaluate the overhead introduced by RADIC fault tolerance according the application communication pattern. Thus we compare the execution time and number of operation per second of an application while using RADIC fault tolerance or not. These experiments run in fault free execution.

The second experiments evaluate the checkpointing procedure overhead. The checkpoint overhead is caused by the variables present in the checkpointing operation, such process size, disks, network and memory (Vaidya, 1997). Here we are going to evaluate the time spent for checkpointing operation and the *mean time to recovery* – MTTR.

6.4.1 Evaluation According Communication Pattern

To evaluate the RADIC implementation according the application communication pattern we have executed the applications described in item

6.2. Table 6-7 depicts the combinations of application, problem size and number of processors used to execute these experiments.

Table 6-7: Experiments executed to evaluate the overhead introduced by RADIC in a fault free execution according the application communication pattern.

Application	Problem Size (class)	Number of Nodes
BT	A, B and C	4, 9, 16 and 25
CG	A, B and C	4, 8, 16 and 32
FT	A and B	4, 8, 16 and 32
IS	A, B and C	4, 8, 16 and 32
LU	A, B and C	4, 8, 16 and 32
MG	A, B and C	4, 8, 16 and 32
SP	A, B and C	4, 9, 16 and 25

In the graphs presented in this chapter, the blue line is executions without fault tolerance and the red lines execution with RADIC architecture.

BT, LU and SP

As explained in item 6.1 BT LU and SP benchmark application have similar communication pattern. However, the number of message and message size changes.

The BT application while using RADIC fault tolerance has a different behavior which depends on the problem size. With small problems the RADIC overhead introduced by first checkpoint operation and message logging is more significant than while using bigger problems. Due to BT communication pattern – asynchronous communication overlapped with computation, in small problem size application we expect an overhead with must be dispelled while the problem size increases. Figure 6-12 depicts this affirmative.

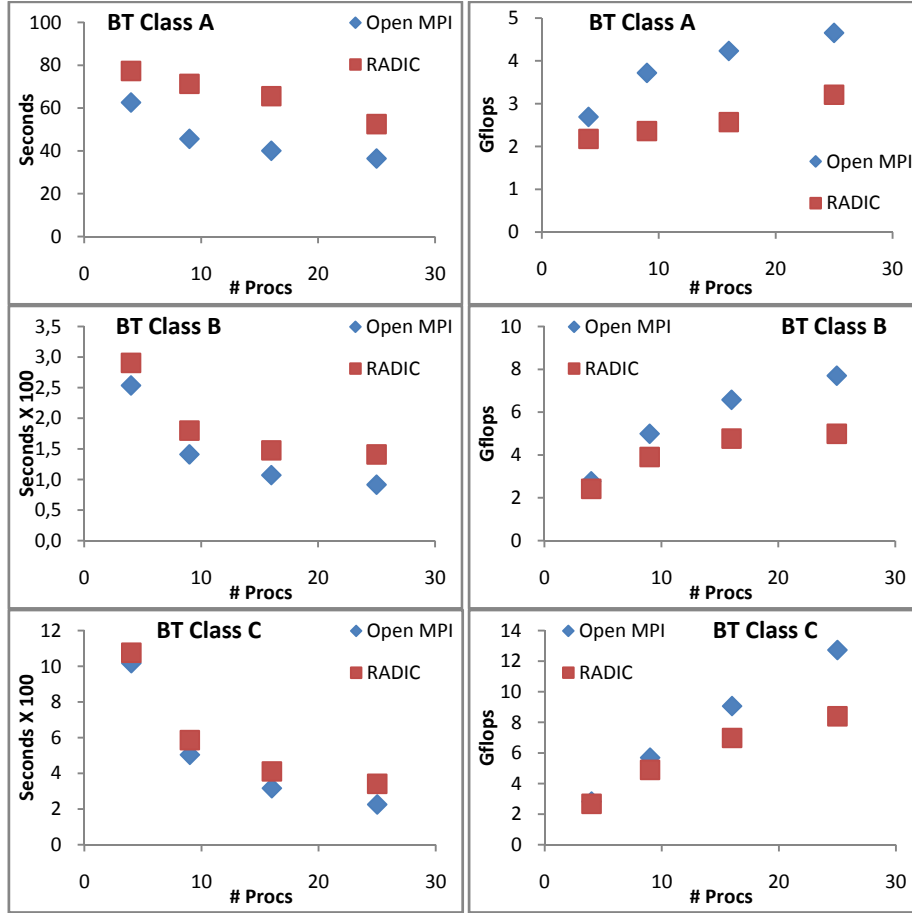


Figure 6-12: Execution of NAS BT benchmark application class A, B and C in 4, 9, 16 and 25 nodes.

As we can see in Table 6-8, Table 6-9 and Table 6-10, BT application class A have an average overhead of 46.9% while using RADIC, in comparison with the same application without fault tolerance. Furthermore, using bigger problems, the same application have 33.5% and 25.9% for class B and C, respectively. These tables confirm our expectation about BT application while analyzing its overhead according the problem size perspective.

Table 6-8: Overhead introduced by RADIC fault tolerance in NAS BT application class A while using 4, 9, 16 and 25 processors.

Processors	Execution time without RADIC	Execution time with RADIC	Overhead
4	62.59 s	77.34 s	23.56%
9	45.61 s	71.31 s	56.34%
16	40.04 s	65.53 s	63.66%
25	36.41 s	52.45 s	44.05%

Table 6-9: Overhead introduced by RADIC fault tolerance in NAS BT application class B while using 4, 9, 16 and 25 processors.

Processors	Execution time without RADIC	Execution time with RADIC	Overhead
4	253.60 s	290.66 s	14.61%
9	140.92 s	179.76 s	27.56%
16	106.95 s	147.38 s	37.80%
25	91.32 s	140.65 s	54.01%

Table 6-10: Overhead introduced by RADIC fault tolerance in NAS BT application class C while using 4, 9, 16 and 25 processors.

Processors	Execution time without RADIC	Execution time with RADIC	Overhead
4	1017.41 s	1075.82 s	5.74%
9	503.68 s	586.51 s	16.44%
16	316.29 s	410.88 s	29.90%
25	225.16 s	341.34 s	51.60%

Analyzing the one class isolated, the same affirmative can be made while we compare Table 6-1, which depicts the problem size, and Table 6-10, for example. In Table 6-10 we can understand that the overhead increase while the problem per processor decrease. Thus, we can affirm that to NAS BT application bigger problems results in smaller overhead while using RADIC fault tolerance.

As example of the NAS BT application, to LU and SP application the same behavior is verified. This behavior is depicted in Figure 6-13 and Figure 6-14. In Figure 6-13 – Class A, we can note that the application reaches its scale limit. Thus, increase the number of nodes does not reduce the execution time, as well as does not increase the application gigaflops per second. The processor becomes waiting for communication. The ratio between computation and communication is not adequate. In this case the application turns *communication bound*. It occurs because the application reaches its scale limit.

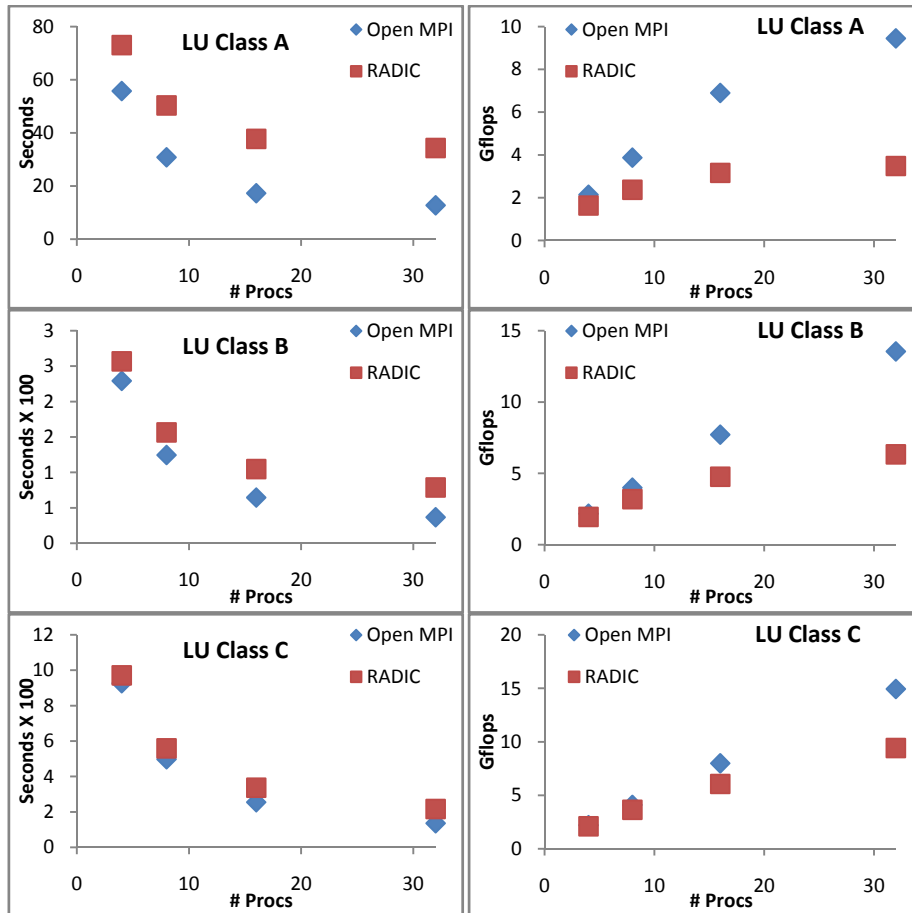


Figure 6-13: Execution of NAS LU benchmark application class A, B and C in 4, 8, 16 and 32 nodes.

A communication bound application is the worst situation for the RADIC fault tolerance because the message logging mechanism increase the time spent for message delivery. Thus, this increase in the communication time incur in an increase to the execution time.

In Figure 6-14 – Class A, as explained while analyzing similar application – BT and LU, the execution time grows because the problem is too small. The application becomes communication bound. In Table 6-11 we can verify that the overhead added to the message delivery by the RADIC message logging mechanism increases the execution time while more processors are used to runs the application. This problem is not related to the RADIC architecture. The application itself has a scale limit which was reached.

Table 6-11: Overhead introduced by RADIC fault tolerance in NAS SP application class A while using 4, 9, 16 and 25 processors.

Processors	Execution time without RADIC	Execution time with RADIC	Overhead
4	72.68 s	102.11 s	40.49%
9	64.63 s	108.60 s	68.03%
16	54.42 s	112.01 s	105.82%
25	57.87 s	115.95 s	100.36%

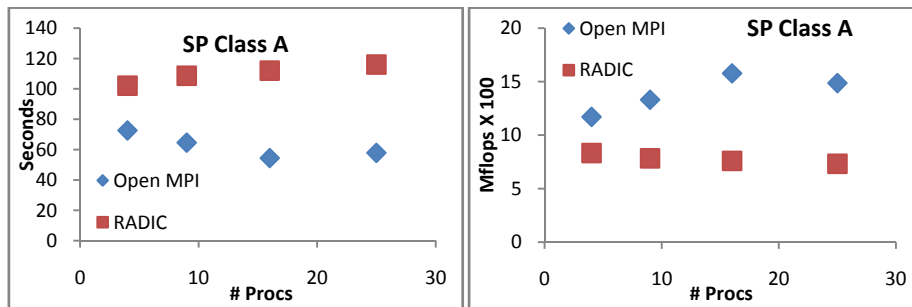


Figure 6-14: Execution of NAS SP benchmark application class A, B and C in 4, 9, 16 and 25 nodes.

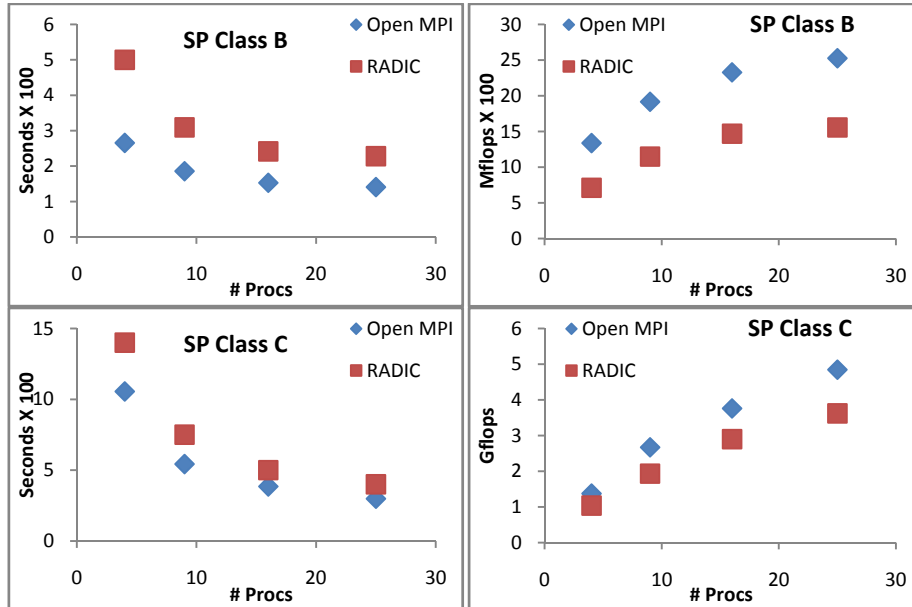


Figure 6–14: Execution of NAS SP benchmark application class A, B and C in 4, 9, 16 and 25 nodes.

IS and FT

As explained in item 6.1, IS and FT applications have similar communication pattern. Nevertheless, they implement different algorithms. IS application was design to still computing while messages does not arrive. These messages are asynchronous, thus, sender neither receiver stops computation in order to communicate. In contrast, FT application implements synchronous communication.

We would not expect changes in the execution time of the IS application because due its algorithm – an integer array sorting, the application could continue sorting the entire array while messages does not arrive in the application layer.

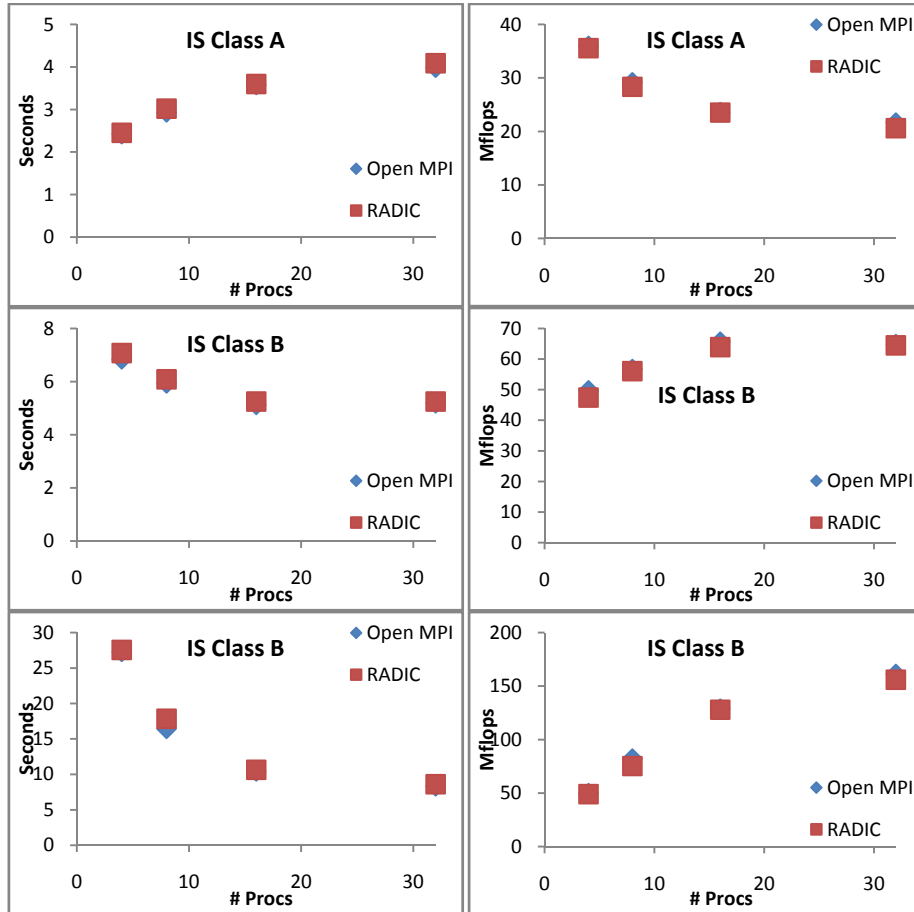


Figure 6-15: Execution of NAS IS benchmark application class A, B and C in 4, 8, 16 and 32 nodes.

As shown in Figure 6-15 and Figure 6-16, the difference between the algorithms implemented by IS and FT reflects on the overhead introduced by the message logging mechanism of the RADIC architecture. IS application execution time is indifferent to the message logging mechanism, while the FT application is affected according the problem size.

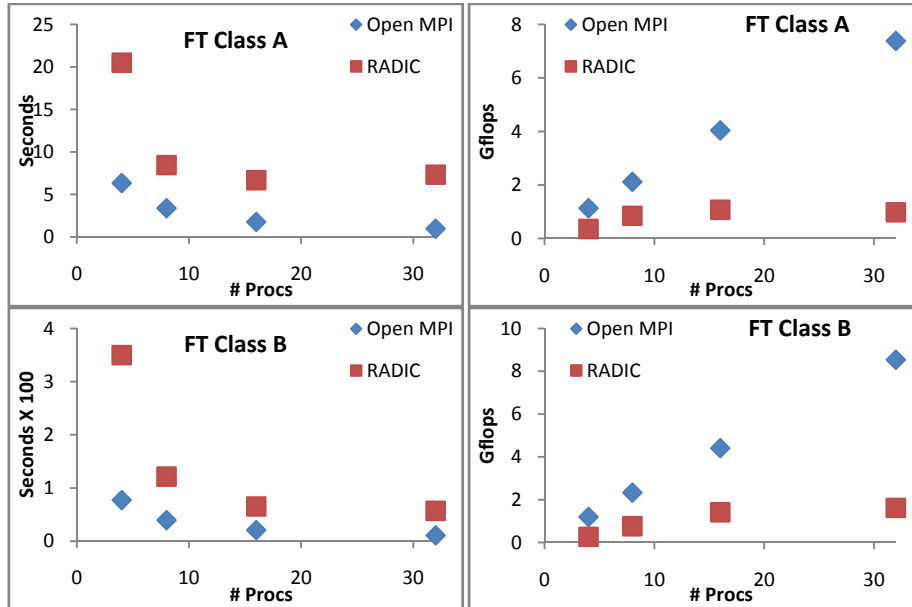


Figure 6-16: Execution of NAS FT benchmark kernel class A and B in 4, 8, 16 and 32 nodes.

In Figure 6-16, a common problem which occurs to applications designed with the master/worker paradigm is depicted. The first phase of the master/worker paradigm is the data distribution. This operation is basically communication between one process – master, and all other processes – workers. Normally this operation is made using collective communications.

Because the RADIC architecture doubles the message latency, in order to log these messages, the master communication channel could be jammed with the logging operation of a worker process while it is trying to distribute data to the workers.

CG and MG

CG and MG are benchmarks kernels which are extremely sensible to the message latency. There are not developed to be executed as applications.

Normally these kernels run on symmetric multiprocessing environments or on parallel machines equipped with a high performance network.

In CG and MG kernels, the communication time and computation time is very similar, and the communications is overlapped with computation. Any increase in the message latency will downgrade the application performance.

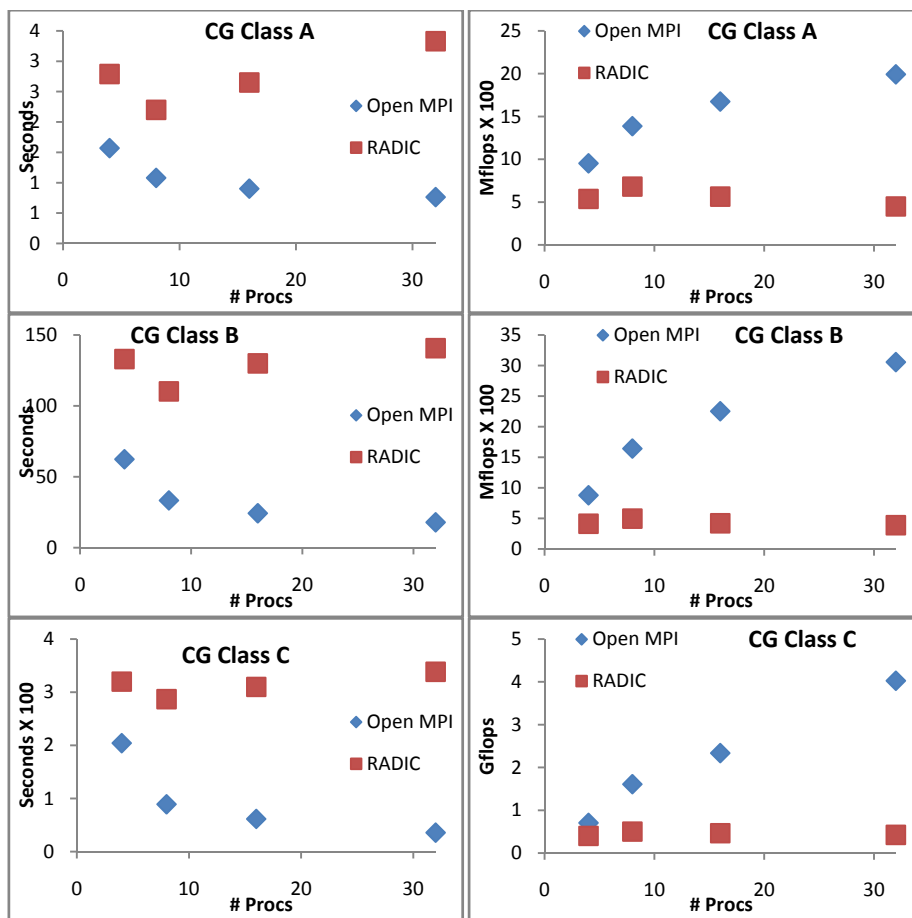


Figure 6-17: Execution of NAS CG benchmark kernel class A, B and C in 4, 8, 16 and 32 nodes.

The environment where we run these kernels is not adequate. Thus, the overhead introduced by the message logging is significant. In Table 6-12 we depict the overhead introduced by RADIC in NAS CG kernel class C. This situation occurs because two different reasons: the first is the application sensibility to message latency increases, and the second is because the network was jammed with the excess number of messages and message log.

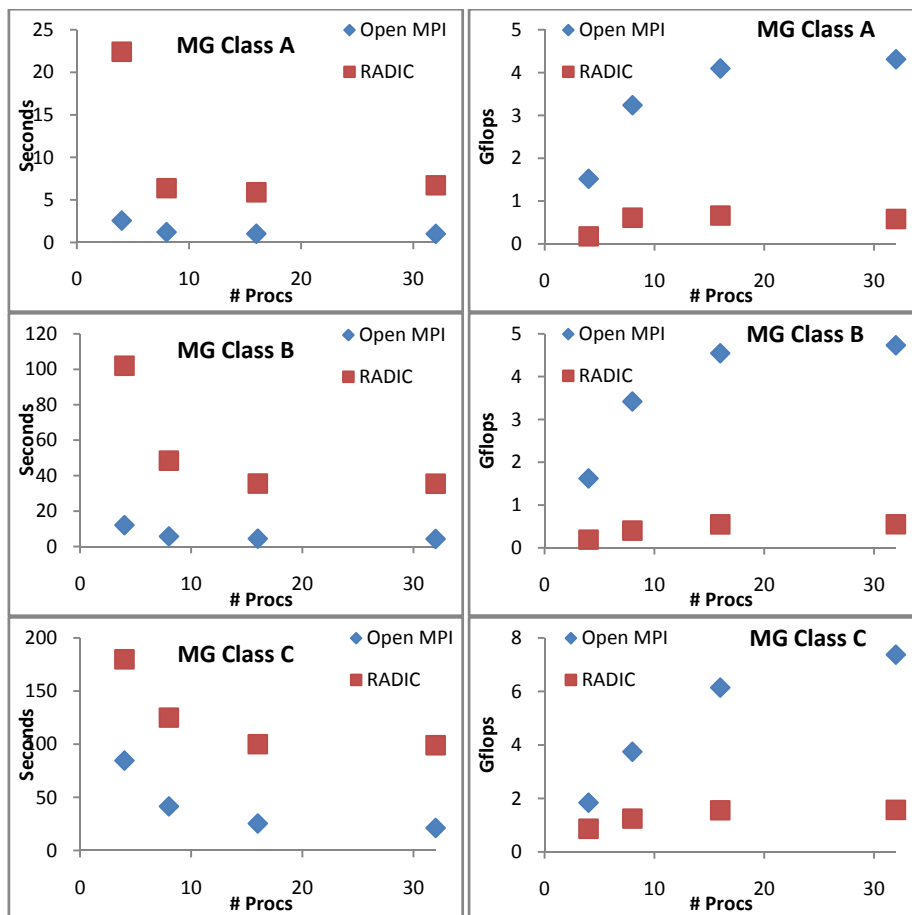


Figure 6-18: Execution of NAS CG benchmark kernel class A, B and C in 4, 8, 16 and 32 nodes.

Table 6-12: Overhead introduced by RADIC fault tolerance in NAS CG kernel class C while using 4, 8, 16 and 32 processors.

Processors	Execution time without RADIC	Execution time with RADIC	Overhead
4	204.30 s	320.41 s	56.83%
8	89.18 s	286.94 s	221.75%
16	61.45 s	309.76 s	404.08%
32	35.59 s	338.55 s	851.25%

6.4.2 Evaluation According Process Size

To evaluate the RADIC implementation according the process size we have executed the applications described in item 6.2. Table 6-2, Table 6-3 and Table 6-4 depicts the process size, or state size, of all NAS applications and kernel. We have selected some applications and kernels in order to evaluate the time spent in checkpointing and recover it.

The applications and kernel selected are depicted in Table 6-13. We choose this set of applications and kernels because the process size of them is, approximately, equidistant.

Table 6-13: Application and kernels selected to analyze the checkpointing time and the mean time to recovery – MTTR.

Application	Process Size	Application	Process Size
LU class A	22 MB	IS class B	155 MB
SP class A	34 MB	LU class C	197 MB
IS class A	45 MB	CG class C	316 MB
LU class B	59 MB	SP class C	371 MB
SP class B	109 MB	FT class B	468 MB
FT class A	123 MB	IS class C	598 MB
MG class B	138 MB	MG class C	920 MB

On these experiments we have used applications running on four nodes because the wide range of process size. Meanwhile, the number of nodes is irrelevant to evaluate checkpointing time and mean time to recovery (Kalaiselvi, et al., 2000).

Figure 6-19 depicts the time spent to take a checkpoint according the process size. As we can see, this time is linear to the process size. The time necessary to recovery a process from a previous checkpoint is linear also. Furthermore, recover a process is faster than take a checkpoint.

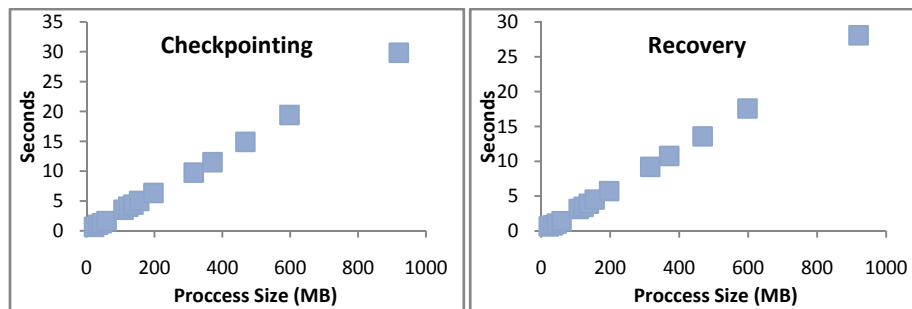


Figure 6-19: Time spent to take a checkpoint and recover a process according process size.

This behavior is expected because according to the literature (Kalaiselvi, et al., 2000), the recovery and checkpointing time depends on the storage media, memory characteristics and operating system. These times are not related to the application.

Furthermore, after take a checkpoint, the file generated must be transferred to the protector node. The transfer time depends on the networks occupation, thus, depends on the application communication pattern. If one parallel computer has different networks for MPI messages, message logging and checkpointing operation; predict the checkpoint transfer time is a simple task. Nevertheless, it is not out case.

During our experiments, we have measured the transferring time of the checkpoints in two situations: the first the *initial checkpoint* taken during the fault tolerance startup, and second the checkpoints taken while other processes are communicating.

The first situation involves all processes sending its checkpoint files to other nodes. Furthermore, he has measured the time spent only for one process.

The second situation involves one process sending its checkpoint file to a protector. However, on the same node where its protector is located, there is other process. This process is communicating with other processes on the parallel application.

Figure 6-20 depicts the transfer time needed by one process to transfer a checkpoint file to its protector during the fault tolerance startup. Because this initial checkpoint does not be affected by application communication we have grouped all results in one graph. As we can see, the time necessary to transfer the checkpoint file grows exponentially according to the process size.

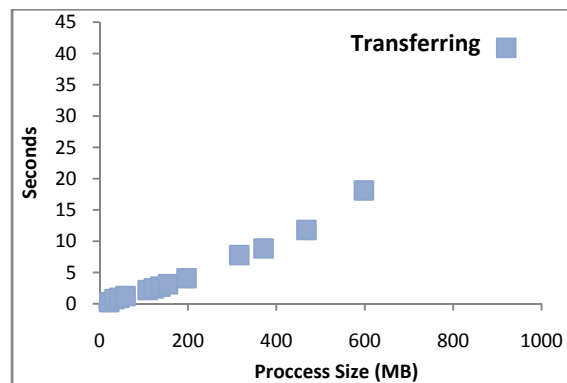


Figure 6-20: Time spent to send a checkpoint according process size.

The exponentially growth verified in Figure 6-20 occurs because the congestion caused by the transference reduce the network performance. Small checkpoint files are not affected by the congestion, as shown in Figure 6-21. The black line represents the linear tendency curve.

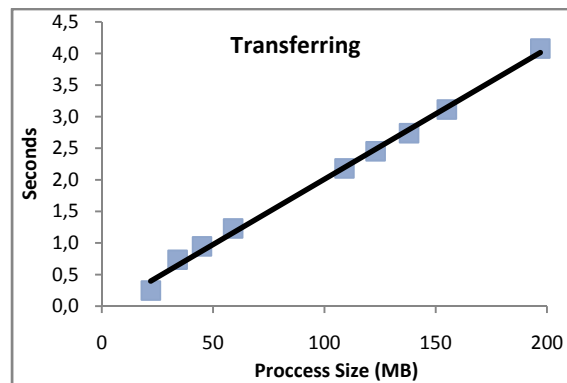


Figure 6-21: Linear tendency of transfer time for small checkpoint files.

Normally, the checkpoint interval in RADIC architecture is configured with the same value for all application processes. This configuration result that all application processes will be checkpointed at the same time, approximately. Furthermore, active communication could change it. In long running applications checkpoints may be taken is different moments. It occurs because the distributed characteristic of the RADIC architecture does not permit checkpoint coordination or synchronization.

In order to analyze the scenario where one process is sending a checkpoint file while the process located on the same node where its protector is located is communicating, we have configured different checkpoint intervals for each application process.

We have discarded applications with small execution time, because on these applications the checkpoint transfer time could be greater than the

execution time. When the application reaches the end of its execution and finalize, the parallel environment provided by the protector daemon is disassembled and active checkpoint transfer cancelled.

Thus, the applications which we have selected to evaluate the competition between the checkpoint transfer and MPI communication are: LU class B and C, SP class B and C, and CG class C. Table 6-14 depicts the execution time of these applications without checkpoint transfer intrusion.

Table 6-14: Execution time while running the selected application using 4 processors to evaluate competition between the checkpoint transfer and MPI communications.

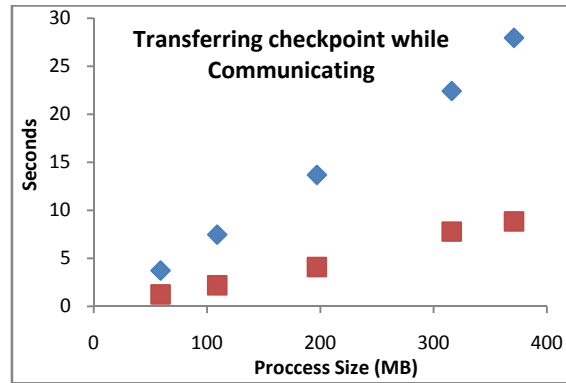
Application	Execution Time
LU class B	256.72 s
LU class C	970.92 s
SP class B	499.45 s
SP class C	1405.74 s
CG class C	320.41 s

Figure 6-22 depicts the expected behavior the transfer time of checkpoint files. The competition between checkpoint file transfer and MPI communication increases the time of the first, and probably of the second also. The same behavior described about large checkpoint file transfers – where the transfer time grows exponentially, is verified while the checkpoint transfer competes with MPI communication.

As expected, the execution time of these applications grows while checkpoints are being transferred at the same time that applications are communicating. Table 6-15 depicts the execution time with and without the interference introduced by the checkpoint file transfer. We have configured the RADIC architecture to take and send checkpoint at every three minutes – 180 seconds interval.

Table 6-15: Comparison between applications execution time while sending checkpoint files or not.

Benchmark	Execution Time without checkpoint transfer	Execution Time with checkpoint transfer (overhead)	# of checkpoint transferred
LU class B	256.72 s	261.12 s (1.71%)	1+1
LU class C	970.92 s	1058.82 s (9.05%)	1+5
SP class B	499.45 s	513.87 s (2.89%)	1+1
SP class C	1405.74 s	1577.55 s (12.22%)	1+8
CG class C	320.41 s	357.52 (11.58%)	1+1

**Figure 6-22: Comparison between the checkpoint transfer with (blue line) and without (red line) MPI communication.**

Chapter 7

Conclusions

The current trend for the parallel computers indicates that the size and 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 parallel applications.

The RADIC – Redundant Array of Independent Fault Tolerance Controllers, architecture was studied as a fault tolerance solution for messaging passing systems providing transparency, decentralization, scalability and flexibility. We have presented the concepts of this architecture, which it bases on two kinds of processes working together to provide fault tolerance: *observers* and *protectors*. We saw the *modus operandi* of these processes and their issues besides a practical implementation of the architecture.

During this thesis we show how we have incorporated the RADIC fault tolerance architecture in Open MPI, a well-know and wide-used message passing library. We present the study made before chosen Open MPI architecture as the bases to start de implementation and the principal characteristics which a message passing implementation should be doted in order to support the RADIC architecture.

Our implementation follows the original RADICMPI implementation while we have implemented a receiver based message logging and uncoordinated checkpoint protocols. Furthermore, we take advantage of all communication primitives supported by the Open MPI implementation. As well as all other network devices and frameworks supported by Open MPI.

Besides the analysis we have depicted details about our implementation. These details permits one verify how to adapt the RADIC functionalities to yours necessities.

We validate our implementation using an execution debug log provided by the message passing library and parallel environment. And to evaluate we have defined two variables: the message logging mechanism and the checkpointing operation.

Our experiments shown that the message logging mechanism increase the message latency according the communication patter of the parallel application. For long running applications the overhead generated by the fault tolerance architecture, proportionally, is smaller than short time execution applications.

The evaluation of the checkpointing operation shown that the time spent in checkpointing a process depends on the process size. Furthermore, the transfer of the checkpoint files affects the message passing communication. These factors are related and cannot be analyzed separately. Meanwhile the time spent to recover a process is smaller than the time necessary for checkpointing it.

Finally we prove that Duarte (Duarte, 2007) affirmation can be verified. RADIC can be implemented in an existent message passing library. And Open MPI provides all the resources and frameworks necessary to do it.

Definitively this work is the first step to define the RADIC architecture model. Analyze RADIC functionalities and the relation between them starts to compose the initials variable of the RADIC equation. The RADIC model is the bases to configure and tune the architecture.

Actually, our implementation can be used by any user, developer or researches interested in provide fault tolerance for its parallel applications.

7.1 Future Works

The RADIC architecture seems solve all fault tolerance problems. Thus, one may think there are no open lines of future works to develop. It is not completely true. The RADIC architecture, conceptually, is a solution for many fault tolerance problems. Furthermore, this concept can be expanded.

It is necessary develop a model of the overhead introduced by the RADIC architecture. The model could permit the execution time prediction to a well-know application running on specific parallel computer. Of course it is not a simple task. However, it is the start for other open lines.

Having a model as a base, it is possible to define configuration parameters, according the parallel computer, application knowledge and user necessities. More than configuration parameters, a model permits dynamic tuning of these parameters, in order o achieve a better efficiency.

Besides the model definition, and RADIC configuration and tuning, exists the possibility to follow expanding the RADIC architecture. Santos (Santos, et al., 2008) has did it adding spare nodes to RADIC, which permits the recovery of failed processes in unused nodes to maintain the same performance experimented before the fault, as well as perform a preventive maintenance of the nodes.

A desirable possible expansion of the RADIC architecture includes support to transient failures. Actually, experiment this type of failure with RADIC, may result in unnecessary process migration. A heuristic decision algorithm may analyze the failures history of a node in order to discard or maintain this node in use.

The RADIC architecture takes advantage of the new developments on message logging, checkpointing and network performance. Furthermore, there exists a problem which is generated by any message logging and checkpoint protocol which is the increase of data transferred by the network.

In the future, a new feature can be added to RADIC which permits to analyze the message latency to change the behavior of its messaging passing library to reach a better performance on message logging, checkpointing and MPI communication.

Besides all improvements which can be made in the RADIC architecture a fault injection framework is necessary to test RADIC functionalities. This frameworks must permits one injects controlled faults in specific message passing procedure.

Appendix I

In order to study what is fault tolerance we need to understand what a fault is. Sometimes fault, error and failure are mentioned wrongly. By definition, failure is the perception of undesirable behavior, meaning that something does not act as expected, an error is a consequence generated by a failure and finally, fault is a recoverable state that can generate an error if no action is made to avoid it. Thus, a fault can cause an error, which may cause a failure.

Fault tolerance may be defined as the ability to avoid failures despite existence of errors generated by some fault. Fault tolerance was two basic goals: to increase the system overall reliability and to increase system availability (Jalote, 1994).

Any scheme made to manage fault tolerance in message passing systems implies in some kind of redundancy. Redundancy can be understood as two resources doing the same action at the same time. In case of failure in one of them the other continues working. This approach is defined as active redundancy. Another possibility is to use only one resource doing the action and at predefined time interval update other resource with the actual state. This approach is defined as standby redundancy.

For message passing systems, the traditional method used to implement standby redundancy is rollback-recovery protocols. Due to the distributed environment behavior present in message passing systems, define a consistent state to rollback must consider the computational state of all processes and the interactions between them. Figure A-1 presents the functioning of a simple

message passing system with four processes exchanging messages (diagonal arrows) along an execution timeline (horizontal arrows).

Parallel computers using message passing are more susceptible to failures effects. In these architectures a fault may occur in a computation node or either in a communication network. If the fault occurs in the network the system behavior depends of its implementation, whose can provide or not timeout and retry mechanisms and it the fault is transient or not. When a node fails, the processing assigned to it will be lost, this may incur in inaccurate or

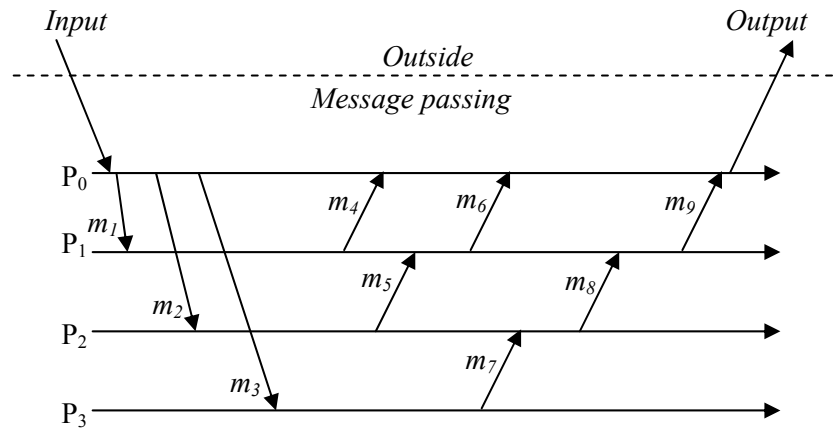


Figure A-1: A message passing system with four processes exchanging messages.

incorrect result and indeed a completely crash of the system aborting all computation made before the fail.

The main difference between different rollback-recovery protocols in message passing systems is the efficiency that they present in the absence and in the present of failures. There are two major groups of rollback-recovery protocols for message passing systems: checkpoint based and log based. Some implementations have merged first and second advantages in a hybrid rollback-recovery protocol.

Rollback-Recovery Fundamentals

Rollback-recovery is a technique to provide fault tolerance based in backing the program execution to a previous point before the failure, and then retry the computation. According Shooman (Shooman, 2002) there are four basic types of rollback-recovery techniques:

Reboot/Restart Techniques

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

Journaling Technique

It bases in periodically storing of all system inputs. In case of failure the processing may be repeated automatically. This technical is a usual feature in most word processors.

Retry Techniques

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

Checkpointing Techniques

Checkpointing techniques can be considered as an improvement of reboot one. In this approach a system state snapshot is saved periodically, so the application or the system just need to back to the most recent checkpoint before the failure.

Due to the characteristics of the applications running in parallel systems, usually executing during a long time, checkpoint approach becomes more suitable for these systems. But performs a checkpoint is more difficult in distributed systems in comparison with centralized ones (Kalaiselvi, et al., 2000). This difficult exists because distributed systems are composed by a set of independent processors with individual execution timelines and there is not a global synchronized clock between them.

Basic Concepts

For a better understood about rollback-recovery techniques, we should introduce some important concepts applied to distributed systems environment.

Checkpoint

Also known as recovery point or snapshot, checkpoints may be considered as a process state at one moment. Checkpointing is the checkpoint storing procedure. Is this procedure all information necessary to re-spawn the process is stored in a stable storage. This information is a copy of the memory used by the process, that include variables and registers values, control point, threads, sometimes application code and etc. In case of failure, the fault tolerance system use this saved state to recover the process. In single machines checkpointing is not a complex issue, but when applied in a

distributed context it is not quite simple. As the processes communicate between themselves, each checkpoint must consider the relevant communications exchanged.

Stable Storage

Use checkpoint to perform roll-back recovery generally requires that system must be available after the failure. In order to provide this feature fault tolerance techniques suppose the existence of a stable storage which is not affected by the system failures. Although stable storage is usually confused with physical disk storage, it is just an abstract concept (Elnozahy, et al., 2002). A stable storage may be implemented in different ways:

- a) It may be a disk array using RAID, allowing tolerates any number of num transient failures;
- b) If using a distributed system, a stable storage may be performed by the memory of a neighbor node;
- c) If it just needs to tolerate transient faults, a stable storage may be implemented using a disk in the local machine.

Consistent System State

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

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

Recovery Line

In a distributed system, to be considered a consistent recovery line and so be used as a recovery point, a set of checkpoints must satisfy the following restriction (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 succeeding 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 A-2 shows an example of two global system states. In this figure, the cylinders indicate the checkpoints of the processes. In Figure A-2a, the global system state indicates that the process P_1 sent a message m_4 , but process P_0 has not yet received it. In such situation, if process P_0 fails and rolls back to the state represented by the checkpoint C_0 , the system goes to an inconsistent global state because the item c) was not satisfied.

The consistency of the global system state depends on how the recovery protocol deals with in transit message. If the rollback-recovery protocol

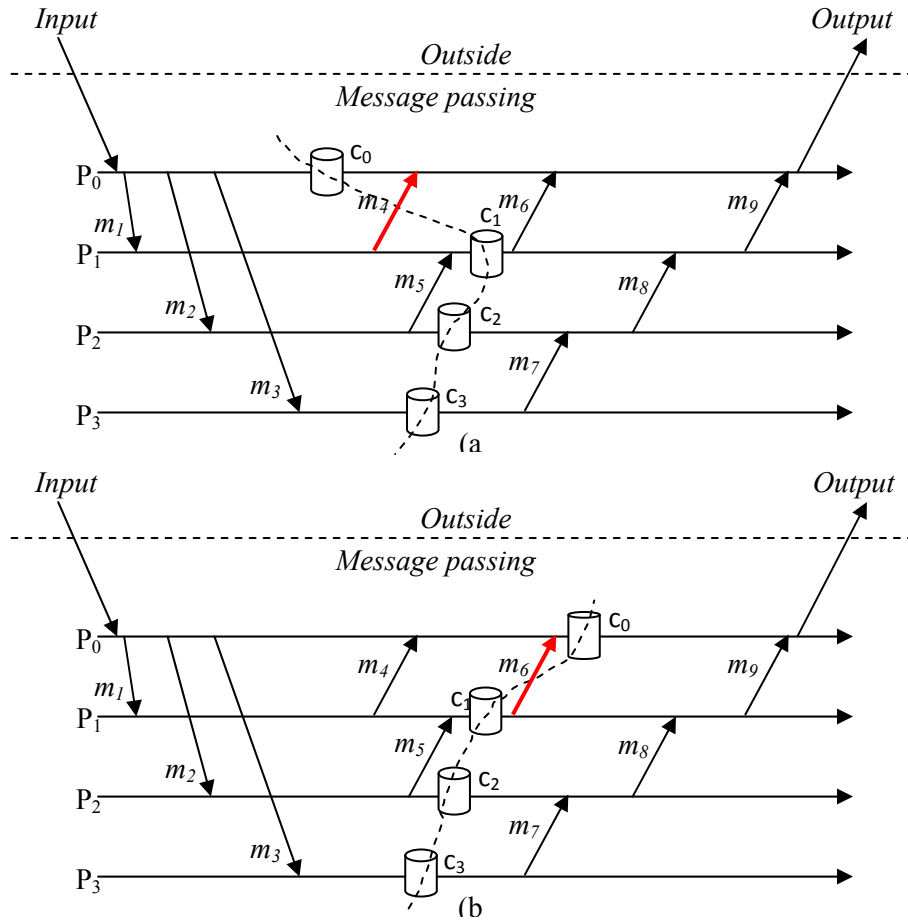


Figure A-2: Examples of inconsistent global system state, (a).caused by a lost message and (b) by an orphan message.

assume that the message channels are reliable, then the global system state in Figure A-2a is inconsistent and m_4 is a *lost message*. On the other hand, if the message channels are unreliable, this global system state is consistent and m_4 is an *in-transit message*.

The example of Figure A-2b also show a inconsistent global system state because the state of process P_0 considers that P_0 has received message m_6 but the state of process P_1 does not consider that P_1 has sent message m_6 . In this case message m_6 is an *orphan message*.

In-Transit Message

In-transit messages are messages that have started before the checkpointing but were not completely transferred. So, these messages are 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:

- a) An implementation based on a reliable communication protocol. In this case a communication channel protocol ensures the reliability of message delivery during failure-free execution, but it cannot the reliability of message delivery in the presence of failures. For instance, a conventional communication protocol will generate a timeout and 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 message available to the intended receiver process after it recovers.
- b) An implementation based on an unreliable communication protocol. 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.

Domino Effect

The domino effect (Koren, et al., 2007) may occur when the processes of a distributed application take their checkpoints in an uncoordinated manner. When a failed process rolls back to its most recent checkpoint, its state may not reflect a communication with other processes, forcing these processes to roll back to checkpoint prior this communication. This situation may continue happening until reach the initial of the execution. Following, we exemplify this happening by the situation depicted in Figure A-3 that shows

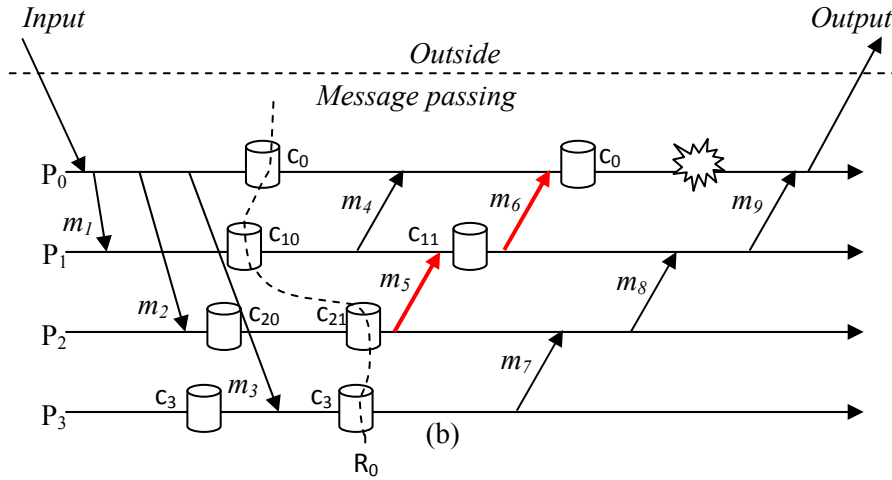


Figure A-3: Domino effect.

an execution in which processes take their checkpoints (represented by cylinders) without coordinating with each other.

We consider the process starts as an initial checkpoint. Suppose that process P_0 fails and rolls back to checkpoint c_{0l} . The rollback of P_0 invalidates

the sending of message m_6 , and so P_1 must roll back to checkpoint c_{11} 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. 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 recovery line R_0 shown in Figure A-3 represents the recovery line of the system in case of a failure in P_0 .

Logging Protocols

Log-based rollback recovery is a strategy used to avoid the domino effect caused by uncoordinated checkpoints. Logging protocols is a set of protocols whose take message logs besides checkpoints. Such protocols are based on the *piecewise deterministic* (PWD) assumption (Strom, et al., 1985). Under this assumption, the rollback recovery protocol can identify all the non-deterministic events executed by each process. For each non-deterministic 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.

Checkpointing Based Protocols

The goal of rollback-recovery protocols based on checkpoint is to restore the system to the most recent consistent global state of the system, in other words, the most recent recovery line. Since such protocols do not rely on the PWD assumption, they do not care about non-deterministic events, that

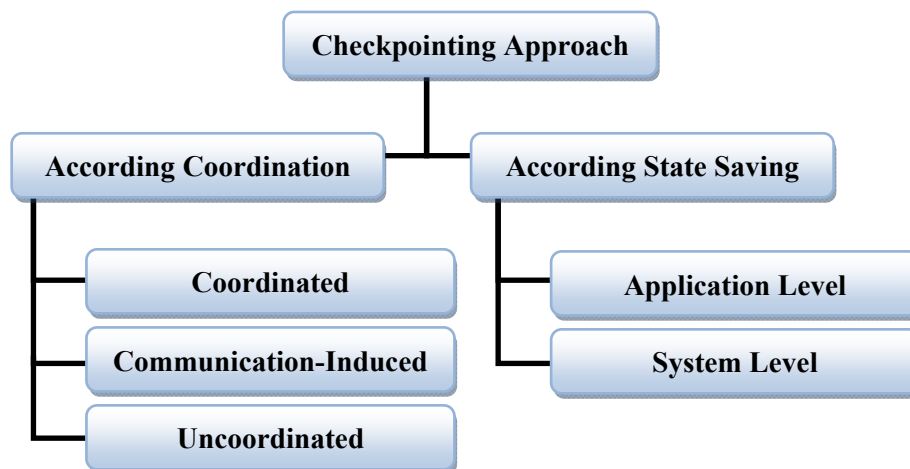


Figure A-4: Different checkpointing approaches.

it means they do not need to detect, log or replay non-deterministic events. Therefore, checkpoint-based protocols are simpler to implement and less restrictive than message-log methods.

Figure A-4 shows a classification scheme for checkpoint approaches based on where is performed: application level or system level – or in the coordination strategy: uncoordinated, communication induced or coordinated. The next topics explain the three categories of the checkpointing strategies used by the checkpoint-based protocols: *uncoordinated*, *coordinated* and *communication-induced*.

Coordinated Checkpointing

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

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

Although straightforward, this scheme can yield in a large overhead. An alternative approach is to use a non-blocking checkpoint scheme like the proposed in (Chandy, et al., 1985) and in (Elnozahy, et al., 2002). However, non-blocking schemes must prevent the processes from receiving application messages that make the checkpoint inconsistent.

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

Uncoordinated Checkpointing

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

The uncoordinated strategy simplifies the checkpoint mechanism of the rollback-recovery protocol because it gives independence for each process manages 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.

Communication-Induced Checkpointing

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

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

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

Indexed-based coordination protocols

These protocols assign timestamps to local and forced checkpoints such that checkpoints with the same timestamp at all processes form a consistent

state. The timestamps are piggybacked on application messages to help receivers decide when they should force a checkpoint (Elnozahy, et al., 2002).

In CIC, each process has a considerable autonomy in taking checkpoint. Therefore, the use of efficient policies in order to decide when to take checkpoints can lead to a small overhead in the system. Since these protocols do not require processes to participate in a globally coordinated checkpoint, they can, in theory, scale up well in systems with a large number processes (Elnozahy, et al., 2002).

Model-based protocols

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

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

Comparing Checkpointing 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.

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, et al., 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 non-deterministic event (Jalote, 1994). Each non-deterministic event relates to a unique *determinant*. In distributed systems, the typical non-deterministic 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 A-5, the

execution of process P_0 is a sequence of four deterministic intervals. The first one is the process' creation and the other three starts with the receipt of m_4 , m_6 and m_9 (represented by red arrows). The initial state of the process P_0 is the unique determinant for sending m_1 , m_2 and m_3 (represented by blue arrows).

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 non-deterministic events

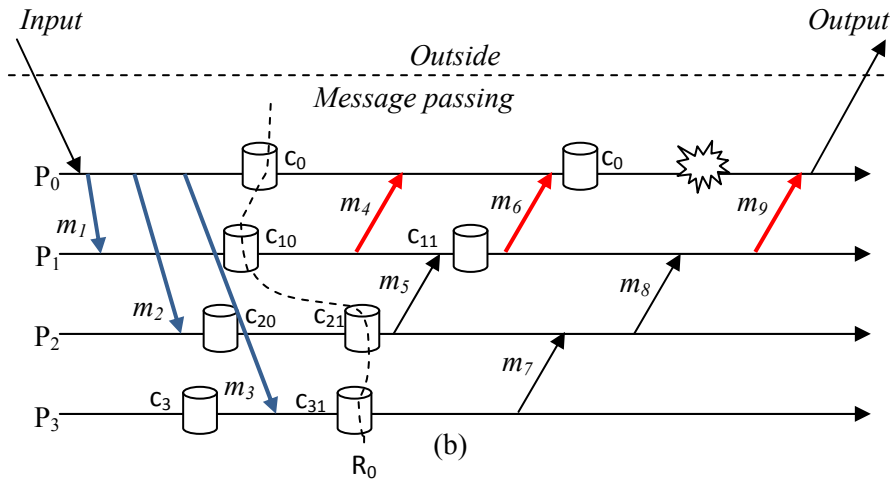


Figure A-5: About P_0 , deterministic events in blue and non-deterministic events in red.

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 non-deterministic 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*.

Pessimistic Log Based Protocol

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

Processes also take periodic checkpoints in order to limit the amount of work that the faulty process has to repeat during recovery. If a failure occurs, the process restarts from its most recent checkpoint. During the recovering

procedure, the process uses the logged determinants to recreate the pre-failure execution, without needing any synchronization between the processes. The checkpoint period implies directly in the overhead imposed by fault tolerance, creating a dilemma: if checkpoints is taken in short periods, it will cause greater overhead during a failure-free execution, but less *expensive* will be the recovery process.

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

- 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.

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

complex, needing to involve each machine that has communicated with the failed process.

Optimistic Log Based Protocols

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

If a process fails, the determinants in its volatile log will be lost, and the state intervals started by the non-deterministic 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 (Elnozahy, et al., 2002) and detailed below:

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 assembles its dependencies to obtain the complete dependency information.

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

Asynchronous Recovery

In this scheme, a recovery process broadcasts a rollback announcement to start a new incarnation. Every process that receives a rollback

announcement checks if it has become an orphan because of the announcement and then, if necessary, it rolls back and broadcasts its own rollback announcement.

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

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.

Appendix II

Bellow we present the information about NAS benchmark kernel or application used to validate and experiment our implementation.

BT and SP Applications

The SP and BT algorithms have a similar structure: each solves three sets of uncoupled systems of equations, first in the x , then in the y , and finally in the z direction. These systems are scalar penta-diagonal in the SP code, and block tri-diagonal with 5x5 blocks in the BT code (Bailey, et al., 1995).

The NPB implementations of SP and BT solve these systems using a multi-partition scheme because it provides good load balance and uses coarse grained communication.

In the multi-partition algorithm each processor is responsible for several disjoint sub-blocks of points, or *cells*, of the grid. The cells are arranged such that for each direction of the line solve phase the cells belonging to a certain processor will be evenly distributed along the direction of solution. This allows each processor to perform useful work throughout a line solve, instead of being forced to wait for the partial solution to a line from another processor before beginning work. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore the granularity of communications is kept large and fewer messages are sent.

Both the SP and BT codes require a square number of processors.

CG Kernel

CG is a benchmark to compute the random sparse conjugate gradient. This benchmark requires the repeated solution to $A * X = F$, where A is a random sparse matrix. The approach used to parallelize is to use a matrix as the connectivity graph for the elements in the vector. By a preprocessing step the algorithm divides the vector into segments of nearly equal size that minimizes the communication between subsets. The resulting algorithm can have a low communication to computation ratio, according the number of nodes used (Bailey, et al., 1995).

CG runs on a power-of-two number of processors.

FT Kernel

FT is a kernel for Fast Fourier Transform (FFT). The implementation of the 3-D FFT PDE benchmark follows a fairly standard scheme. The 3-D array of data is distributed according to z -planes of the array; one or more planes are stored in each processor (Bailey, et al., 1995).

The forward 3-D FFT is then performed as multiple 1-D FFTs in each dimension, first in the x and y dimensions, which can be done entirely within a single processor, with no inter-process communication. An array transposition is then being performed, which amounts to an all-to-all exchange, wherein each processor must send parts of its data to every other processor.

The final set of 1-D FFTs is then performed. A conventional Stockham-transpose-Stockham scheme is used for the 1-D complex FFTs. This procedure is reversed for inverse 3-D FFTs.

FT runs on a power-of-two number of processors.

IS Application

The IS benchmark is an Integer Sort benchmark. It models the ranking step of a counting sort (kind of bucket sort) application, which occurs for instance in particle simulations. The IS benchmark takes a list L of small integers as an input and computes for every element $x \in L$ the rank $r(x)$ as its position in the sorted list (Grün, et al., 1998).

The NAS Integer Sort (IS) benchmark spends over 85% of its time in the two loops copying an array into a temporary buffer, and counting the number of occurrences of each value in the array. It was design to explorer the multi-threading processors characteristics.

IS runs on a power-of-two number of processors.

LU Application

The LU benchmark solves a finite difference discretization of 3-D compressible Navier-Stokes equations through a block-lower-triangular block-upper-triangular approximate factorization of the original difference scheme (Saphir, et al., 1996).

This code requires a power-of-two number of processors. A 2-D partitioning of the grid onto processors occurs by halving the grid repeatedly in the first two dimensions, alternately x and then y , until all power-of-two processors are assigned, resulting in vertical pencil-like grid partitions on the individual processors.

The ordering of point based operations constituting the SSOR procedure proceeds on diagonals which progressively sweep from one corner on a given z -plane to the opposite corner of the same z -plane, thereupon proceeding to the next z -plane. Communication of partition boundary data

occurs after completion of computation on all diagonals that contact an adjacent partition. This constitutes a diagonal pipelining method. It results in a relatively large number of small communications of 5 words each (Bailey, et al., 1995).

The described benchmark has been included because it is very sensitive to the small message communication performance of an MPI implementation.

MG Kernel

The Multi-Grid kernel benchmark is based on the NX reference implementation from 1991. Four critical subroutines – the smoother, the residual calculation, the residual projection, and the tri-linear interpolation of the correction, were optimized for both vector and RISC processors (Bailey, et al., 1995).

This code requires a power-of-two number of processors. The partitioning of the grid onto processors occurs such that the grid is successively halved, starting with the y dimension and then the x dimension, and repeating until all power-of-two processors are assigned.

Appendix III

```
#include </softs/radic_openmpi/include/mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 1048576

int main (int argc, char *argv[]) {
    double time_end, time_start;
    int count, rank, x;
    char *buffer;
    MPI_Status status;

    if (3 > argc) {
        printf("\n\n Insuficient arguments (%d)", argc);
        printf("\n\n ping <times> <delay>\n\n");
        exit(1);
    }
    buffer = (char *) malloc(SIZE * sizeof(char));
    for (x = 0; x < SIZE; x++) {
        strcpy(&buffer[x], "!");
    }
    if (MPI_SUCCESS == MPI_Init(&argc, &argv)) {
        time_start = MPI_Wtime();
        MPI_Comm_size (MPI_COMM_WORLD, &count);
        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
        for (x = 1; x <= atoi(argv[1]); x++) {
            if (rank == 0) {
                printf("(%)d sent token to (%d)\n", rank, rank+1);
                fflush(stdout);
                sleep(atoi(argv[2]));
                MPI_Send(buffer, SIZE, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
                MPI_Recv(buffer, SIZE, MPI_CHAR, count-1, 1, MPI_COMM_WORLD,
                        &status);
            } else {
                MPI_Recv(buffer, SIZE, MPI_CHAR, rank-1, 1, MPI_COMM_WORLD,
                        &status);
                printf("(%)d sent token to (%d)\n",
                        rank, (rank == (count-1) ? 0 : rank+1));
                fflush(stdout);
                sleep(atoi(argv[2]));
                MPI_Send(buffer, SIZE, MPI_CHAR,
                        (rank == (count-1) ? 0 : rank+1), 1, MPI_COMM_WORLD);
            }
        }
        time_end = MPI_Wtime();
        MPI_Finalize();
    }
    if (rank == 0) {
        printf("Total time: %f\n", time_end - time_start);
    }
    return 0;
}
```


References

Agbaria A.M. and Friedman R. Starfish: fault-tolerant dynamic MPI programs on clusters of workstations [Journal] // High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on. - 1999. - pp. 167-176.

Alvisi L. [et al.] An analysis of communication induced checkpointing [Journal] // In Proceedings of The 29th Annual International Symposium on Fault-Tolerant Computing. - Winsconsin, USA : [s.n.], June 15-18, 1999. - pp. 242-249.

Alvisi L. and Marzullo K. Message logging: pessimistic, optimistic, causal, and optimal [Journal] // Software Engineering, IEEE Transactions on. - Feb 1998. - Vol. 24. - pp. 149-159.

Bailey David H. [et al.] The NAS Parallel Benchmarks 2.0 [Article] // Technical Report NAS-95-010. - [s.l.] : NASA Ames Research Center, 1995.

Batchu Rajanikanth [et al.] MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware [Journal] // Cluster Computing. - 2004. - Vol. 7. - pp. 303-315.

Bernholdt David E. [et al.] A Component Architecture for High-Performance Scientific Computing [Journal] // International Journal of High Performance Computing Applications. - [s.l.] : SAGE Publications, 2006. - 2 : Vol. 20. - pp. 163-202.

Bosilca G. [et al.] MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes [Journal] // Supercomputing, ACM/IEEE 2002 Conference. - 16-22 Nov. 2002. - pp. 29-29.

Bouteiller Bouteiller [et al.] MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging [Conference]. - [s.l.] : IEEE Computer Society, 2003. - p. 25.

- Castain R. H. [et al.]** The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing [Journal] // Proceedings, 12th European PVM/MPI Users' Group Meeting. - Sorrento, Italy : [s.n.], September 2005.
- Chandy K. Mani and Lamport Leslie** Distributed snapshots: determining global states of distributed systems [Journal] // ACM Trans. Comput. Syst.. - [s.l.] : ACM, 1985. - Vol. 3. - pp. 63-75.
- Coti Camille [et al.]** Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI [Journal] // Supercomputing, 2006. SC '06. Proceedings of the ACM/IEEE SC 2006 Conference. - Nov. 2006. - pp. 18-18.
- Duarte A.** RADIC: a powerful fault-tolerant architecture.[PhD Thesis] - 2007.
- Duarte Angelo, Rexachs Dolores and Luque Emilio** An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI [Journal] // Recent Advances in Parallel Virtual Machine and Message Passing Interface. - 2006. - pp. 150-157.
- Duarte Angelo, Rexachs Dolores and Luque Emilio** Functional Tests of the RADIC Fault Tolerance Architecture [Journal] // Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on. - 7-9 Feb. 2007. - pp. 278-287.
- Elnozahy E. N. (Mootaz) [et al.]** A survey of rollback-recovery protocols in message-passing systems [Journal] // ACM Computing Surveys. - New York, NY, USA : ACM, 2002. - 3 : Vol. 34. - pp. 375-408. - <http://doi.acm.org/10.1145/568522.568525>. - 0360-0300.
- Elnozahy E. N. and Zwaenepoel W.** On the use and implementation of message logging [Journal] // Proceedings of The 24th International Symposium on Fault-Tolerant Computing. - Austin, USA : IEEE Press, June 15-17, 1994. - pp. 298-307.

Fagg Graham and Dongarra Jack FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World [Journal] // Recent Advances in Parallel Virtual Machine and Message Passing Interface. - 2000. - pp. 346-353.

Fagg Graham E. [et al.] Fault Tolerant Communication Library and Applications for High Performance Computing [Journal] // Proceedings of the 4th Los Alamos Computer Science Institute Symposium (LACSI'03). - Santa Fe, NM, USA : [s.n.], October 27-29, 2003.

Fagg Graham E. [et al.] Scalable Fault Tolerant MPI: Extending the Recovery Algorithm [Journal] // Recent Advances in Parallel Virtual Machine and Message Passing Interface. - Sorrent, Italy : Springer Berlin / Heidelberg, October 03, 2005. - Vol. 3666/2005. - pp. 67-75. - 978-3-540-29009-4.

Fagg Graham E., Bukovsky Antonin and Dongarra Jack J. HARNESS and fault tolerant MPI [Journal] // Parallel Comput.. - [s.l.] : Elsevier Science Publishers B. V., 2001. - Vol. 27. - pp. 1479-1495.

Faraj Ahmad and Yuan Xin Communication Characteristics in the NAS Parallel Benchmarks [Journal] // Proceeding: Parallel and Distributed Computing and Systems. - Cambridge, USA : [s.n.], 2002.

Forum MPI MPI: A message passing interface [Journal] // Supercomputing '93. Proceedings. - November 15-19, 1993. - pp. 878-883.

Forum MPI MPI: Extensions to the message passing interface [Journal]. - July 1997.

Gabriel Edgar [et al.] Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation [Journal] // Recent Advances in Parallel Virtual Machine and Message Passing Interface. - 2004. - pp. 97-104.

Graham Richard L. [et al.] A network-failure-tolerant message-passing system for terascale clusters [Journal] // International Journal Parallel Programming. - Norwell, MA, USA : Kluwer Academic Publishers, 2003. - 4 : Vol. 31. - pp. 285-303. - 0885-7458.

- Graham Richard L., Woodall Timothy S. and Squyres Jeffrey M.** Open MPI: A Flexible High Performance MPI [Journal] // Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics. - Poznan, Poland : [s.n.], September 2005.
- Grün Thomas and Hillebrand Mark A.** NAS Integer Sort on multi-threaded shared memory machines [Journal] // Proceedings of the 4th International Euro-Par Conference on Parallel Processing. - London, UK : Springer Berlin / Heidelberg, 1998. - Vol. 1470. - pp. 999-1009. - 978-3-540-64952-6.
- Hargrove Paul H. and Duell Jason C.** Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters [Journal] // Proceedings of SciDAC 2006. - 2006.
- Hélary Jean-Michel, Mostéfaoui Achour and Raynal Michel** Virtual Precedence in Asynchronous Systems: Cencept and Applications [Journal] // Proceedings of the 11th International Workshop on Distributed Algorithms. - London, UK : Springer-Verlag, 1997. - pp. 170-184. - 3-540-63575-0.
- Hursey J. [et al.]** The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI [Journal] // Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. - 26-30 March 2007. - pp. 1-8.
- Hursey Joshua, Squyres Jeffrey M. and Lumsdaine Andrew** A Checkpoint and Restart Service Specification for Open MPI [Report] / Indiana University. - Jul 2006.
- Jalote Pankaj** Fault Tolerance in Distributed Systems [Book]. - Englewood Cliffs, USA : Prentice Hall, 1994.
- Kalaiselvi S and Rajaraman V** A survey of checkpointing algorithms for parallel and distributted computers. [Journal] // Sadhana. - Sādhanā : [s.n.], 2000. - pp. 489-510.
- Keller Rainer [et al.]** Towards Efficient Execution of MPI Applications on the Grid: Porting and Optimization Issues [Journal] // Journal of Grid Computing. - [s.l.] : Springer Netherlands, June 2003. - 2 : Vol. 1. - pp. 133-149. - 1570-7873.

- Koren I. and Krishna C. M.** Fault Tolerant Systems [Book]. - San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007.
- Litzkow M., Livny M. and Mutka M. W.** Condor - A Hunter of Idle Workstations [Journal] // Proceedings of the 8th International Conference of Distributed Computing Systems. - San Jose, CA, USA : [s.n.], June 1988. - pp. 104-111.
- Rao S., Alvisi L. and Vin H.M.** Egida: an extensible toolkit for low-overhead fault-tolerance [Journal] // Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on. - 1999. - pp. 48-55.
- Sankaran S. [et al.]** The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing [Journal] // International Journal of High Performance Computing Applications. - USA : Sage Science Press, 2005. - 4 : Vol. 19. - pp. 479-493 . - 1094-3420.
- Santos G. [et al.]** Providing non-stop service for message-passing based parallel applications with RADIC [Conference]. - [s.l.] : IEEE Computer Society, 2008. - pp. 653-658.
- Saphir William, Woo Alex and Yarrow Maurice** The NAS Parallel Benchmarks 2.1 Results [Article] // Technical Report NAS-96-010. - [s.l.] : NASA Ames Research Center, 1996.
- Shooman Martin** Reliability of Computer Systems and Networks. Fault Tolerance: Analysis, and Design. 1st. ed. [Book]. - New York, USA : John Wiley & Sons, Inc., 2002.
- Squyres Jeffrey M. and Lumsdaine Andrew** A Component Architecture for LAM/MPI [Journal] // In Proceedings, 10th European PVM/MPI Users' Group Meeting. - Venice, Italy : Springer-Verlag, September / October 2003. - pp. 379-387. - 2840.
- Stellner G.** CoCheck: checkpointing and process migration for MPI [Journal] // Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International. - 15-19 Apr 1996. - pp. 526-531.

Strom Rob and Yemini Shaula Optimistic recovery in distributed systems [Journal] // ACM Transactions Computer Systems. - New York, NY, USA : ACM, 1985. - 3 : Vol. 3. - pp. 204-226. - <http://doi.acm.org/10.1145/3959.3962>. - 0734-2071.

Vaidya Nitin H. Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme [Journal] // IEEE Transactions on Computers. - [s.l.] : IEEE Computer Society, August 1997. - 8 : Vol. 46. - pp. 942-947. - 0018-9340.

Wechsler Elisabeth NAS Parallel Benchmarks Set The Industry Standard for MPP Performance [Article]. - Jan-Feb 1995. - 8 : Vol. 2.

Wong Frederick C. [et al.] Architectural requirements and scalability of the NAS parallel benchmarks [Journal] // Proceedings of the 1999 ACM/IEEE conference on Supercomputing. - Portland, Oregon, United States : ACM, 1999. - p. 41. - 1-58113-091-0.

Woodall T. S. [et al.] TEG: A High-Performance, Scalable, Multi-Network Point-to-Point Communications Methodology [Journal] // Proceedings, 11th European PVM/MPI Users' Group Meeting. - Budapest, Hungary : [s.n.], September 2004. - pp. 303-310.

Zambonelli F. On the effectiveness of distributed checkpoint algorithms for domino-free recovery [Journal] // Proceedings of the Seventh International Symposium on High Performance Distributed Computing. - 1998. - pp. 124-131.