Universitat Autònoma de Barcelona

Escola Tècnica Superior d'Enginyeria
Departament d'Arquitectura d'Ordinadors
i Sistemes Operatius

Performability issues of fault tolerance solutions for message-passing systems: the case of RADIC

Thesis submitted by Guna Alexander Silva dos Santos in partial fulfillment of the requirements for the degree of Doctor of Philosophy per the Universitat Autònoma de Barcelona. This work was advised by Dr. Dolores Isabel Rexachs del Rosario

Barcelona, May 2009

Performability issues of fault tolerance solutions for message-passing systems: the case of RADIC

Thesis submitted by Guna Alexander Silva dos Santos in partial fulfillment of the requirements for the degree of Doctor of Philosophy per the Universitat Autònoma de Barcelona. This work was developed in the Computer Architecture and Operating Systems department of the Universitat Autònoma de Barcelona in option A – "Computer Architecture and Parallel Processing" of the PhD Informatics program, being advised by Dr. Dolores Isabel Rexachs del Rosario.

Barcelona, May 2009

Thesis Advisor

Dr. Dolores Isabel Rexachs del Rosario

To my family.They give me the solid basis that allows me to reach here.

And especially to my beloved angel, who made me strong enough to support this journey.

Acknowledgments

Without doubt, the doctorate process was the greatest challenge of my life until now. It was 1379 tough days since I started this journey. Along this period, many people helped me in some manner, and I am so thankful to all of them, even if not mentioned here. I will try to write my acknowledgments in the native language of each person, or in english when not.

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

Gostaria de agradecer aos meus pais e a minha família. Sr. Eliezer e D. Jane me deram toda a base emocional e de conhecimento para poder chegar aqui. Sua dedicação durante minha vida construiu a pessoa que sou hoje. Meus irmãos, Maria e Rafa, sempre foram a minha torcida, alegrando-se por cada vitória e confortando-me em momentos difíceis.

Creo que tuve los mejores tutores que un doctorando podría tener. Emilio Luque y Dolores Rexachs son corresponsables por el éxito de este trabajo. Gracias Emilio por todos tus *insights* y por regalarme parte de tus conocimientos. Lola, tu participación en este trabajo ha ido mucho más allá de una tutoría, además de tu sabiduría me has regalado con cariño, cuidados y atención que solo regalan las madres, muchas gracias.

Ao meio deste caminho, conheci uma pessoa que veio a ser meu porto seguro durante a parte mais difícil desde doutorado. Natasha, meu anjo, muito obrigado por estar sempre ao meu lado, tentando animar-me, evitando que outros problemas me atrapalhassem e eu pudesse focar somente neste trabalho. Obrigado por estar comigo nos momentos mais difíceis que passei aqui. Obrigado por me fazer feliz. Te amo.

Eduardo Argollo, meu grande amigo. Creio que se não fosse por ele eu não teria vindo para cá e não teria conseguido o que consegui. Muito obrigado por todas as dicas sobre doutorado, ensinando-me o "caminho das pedras" que sei que você teve de descobrir sozinho. Muito obrigado por todos os momentos agradáveis (Nenhum tchau, puéin puéin, reuniões no kebab).

Agradeço a Angelo por suas "aulas" de RADIC e momentos de descontração. Outro agradecimento especial para Genaro, nosso "guru" e meu companheiro de casa. Obrigado por suas dicas em programação C, conceitos de Linux, idéias, etc. Leonardo Fialho, o mais recente desta lista de "novos" amigos de doutorado, apesar do pouco tempo, gostaria de agradecer por toda a sua colaboração no meu trabalho, sugerindo, debatendo, etc..

Hi ha unes persones a les que jo els agraeixo molt: Dani Ruiz, Jordi Valls i després Javier Navarro, el P.T. Moltes gràcies per totes les coses i tot el suport tècnic, l'ajuda amb els meus problemes amb el cluster, BLCR i etc.

Thanks to everyone that helped this dream came true.

Guna Alexander

Barcelona, July 2009

Resumen

¿Es adecuado un sistema rápido pero poco robusto?¿Es adecuado un sistema disponible pero lento? Estas dos cuestiones representan la importancia de prestaciones y disponibilidad en *clusters* de computadores.

Esta tesis se enmarca en el estudio de la relación entre prestaciones y disponibilidad cuando un *cluster* de computadores basado en el modelo de paso de mensajes, usa un protocolo de tolerancia a fallos basado en *rollback-recovery* con log de mensajes pesimista. Esta relación también es conocida como *performability.*

Los principales factores que influyen en la *performability* cuando se usa la arquitectura de tolerancia a fallos RADIC son identificados y estudiados. Los factores fundamentales son la latencia de envío de mensajes que se incrementa cuando se usa el log pesimista, que implica una perdida de prestaciones, como también la replicación de los datos redundantes (*checkpoint* y log*)* necesaria para el incremento de la disponibilidad en RADIC y el cambio de la distribución de procesos por nodo causada por los fallos, que pueden causar degradación de las prestaciones así como las paradas por mantenimiento preventivo.

Para tratar estos problemas se proponen alternativas de diseño basadas en análisis de la *performability*. La pérdida de prestaciones causada por el log y la replicación ha sido mitigada usando la técnica de *pipeline*. El cambio en la distribución de procesos por nodo puede ser evitado o restaurada usando un mecanismo flexible y transparente de redundancia dinámica que ha sido propuesto, que permite inserción dinámica de nodos *spare* o de repuesto.

Los resultados obtenidos demuestran que las contribuciones presentadas son capaces de mejorar la *performability* de un *cluster* de computadores cuando se usa una solución de tolerancia a fallos como RADIC.

Abstract

Is a fast but fragile system good? Is an available but slow system good? These two questions demonstrate the importance of performance and availability in computer clusters.

This thesis addresses issues correlated to performance and availability when a roll-back-recovery pessimistic message log based fault tolerance protocol is applied into a computer cluster based on the message-passing model. Such a correlation is also known as performability.

The root factors influencing the performability when using the RADIC (Redundant Array of Distributed Independent Fault Tolerance Controllers) fault tolerance architecture are raised and studied. Factors include the message delivery latency, which increases when using pessimistic logging causing performance overhead, as also in the redundant data (logs and checkpoints) replication needed to increase availability in RADIC and the process per node distribution changed by faults, which may cause performance degradation and preventive maintenance stops.

In order to face these problems some alternatives are presented based on a performability analysis. Using a pipeline approach the performance overhead of message logging and the redundant data replication were mitigated. Changes in the process per node distribution can be avoided or restored using the flexible and transparent mechanism for dynamic redundancy proposed, or using a dynamic insertion of spare or replacement nodes.

The obtained results show that the presented contributions could improve the performability of a computer cluster when using a fault tolerance solution such as RADIC.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

High availability and high performance computer clusters are two relevant subjects in the parallel computing area. This thesis addresses issues correlated to performance and availability when a rollback-recovery pessimistic message log based fault tolerance protocol is applied into a computer cluster based on the message-passing model. Assuming a hypothesis that the effective performance of a high performance computer depends on its availability and that providing high availability implies a performance overhead, the root causes of such an overhead are studied, including the performance degradation caused by faults. This work presents different levels and organizations for adapting the fault tolerance solution to user requirements, allowing a reduction in the imposed performance overhead, enhancing availability and avoiding performance degradation due to faults.

1.1. Background

Since their creation, computers have played an important and increasing role in solving complex problems. Following the computers evolution, new and more complex problems can be solved each day. Indeed, it seems that despite the growing power of computers applications will always need more resources and large periods of execution time.

This demand for computational power has leaded to the improvement of the High Performance Computing (HPC) area, generally represented by the use of parallel systems running specifically-designed applications. For this reason, the design of parallel systems has

commonly been oriented to achieve the highest performance possible. As shown in TABLE 1-1 as extracted from the Top500 site (TOP500.ORG, 2008), the most usual architectural design of current parallel systems is the computer cluster, which has been adopted by more than 80% of the 500 fastest supercomputers in many areas of knowledge. In order for the computing power of these machines to be effective, it is also important that these computers suffer a minimum of interruptions, i.e., they must be available to perform useful work as much time as possible.

TABLE 1-1: Architecture share of the fastest 500 supercomputers. Source www.top500.org

Architecture	Count	Share %
Constellations	2	0.4
MPP (Massively Parallel Processing)	88	17.6
Cluster	410	82.0

In order to achieve more computing power it is usual to aggregate a large number of computing elements. The problem of this approach is that as more elements have a system, the probability of faults grows. As the number of computing elements of the computer clusters steady increases, faults are already one of the major concerns when designing parallel systems. Taking into consideration that the system mean time between failure (*SMTBF*) of a computer cluster is given by the average mean time between failures of all nodes ($\overline{MTBF}$) divided by the number of cluster's nodes, and supposing that a failure in some node causes a system stop (fail-stop semantic) that takes time to be repaired defined by the mean time to repair (*MTTR*), the overall availability (A_{System}) can be given by the Equation (1). This equation allows to deduce that as more elements have a system, so its availability decreases, This

issue is the reason by why availability and fault tolerance have been widely studied in the past.

$$A_{System} = \frac{SMTBF}{SMTBF + MTTR} = \frac{\overline{MTBF}\ /N}{\overline{MTBF}\ /N + MTTR} \quad (1)$$

Computer clusters may be considered as a class of computing systems with degradable performance (NAGARAJA, K. et al., 2005) i.e., under some circumstances during a determined utilization period, the system may present different performance levels. Such performance degradation is generally caused by faults occurrence, which may also affect the system availability if they have generated an interruption.

Until now, efforts have been focused on providing high availability to computer clusters (GEIST, A. and Engelmann, C., 2002), (CHAKRAVORTY, S. et al., 2006), (NAGARAJAN, A. B. et al., 2007). The solutions resulting from these efforts are commonly based on rollback-recovery techniques (AGBARIA, A. and Friedman, R., 1999), (DUARTE, A. et al., 2006), (BOUTEILLER, A. et al., 2006) and they have shown their efficacy in improving computer cluster availability. However they impose some kind of performance overhead because of their related activities, such as process state saving, messages exchange logging or system health monitoring. In these solutions, performance is commonly analyzed separately from the availability and it is not a concern in many cases.

It is not trivial to evaluate performance completely dissociated from availability when analyzing an entire computing system, because the perceived system performance can be affected by the system availability. Deriving from this assumption, according to Meyer in "On Evaluating the Performability of Degradable Computing Systems" (MEYER, J. F.,

1980) *performability* is considered as a more real, complete and accurate measurement for evaluating degradable systems such as computer clusters.

In "Performability evaluation: where it is and what lies ahead" (MEYER, J. F., 1995) Meyer initially defines *performability* as a *"term referred to a class of (probability) measures that quantify a system's 'ability to perform' in the presence of faults."* This definition takes into consideration systems that gracefully degrade in the presence of faults such as the computer clusters mentioned before. Moreover, Meyer says that *"...such degradation may result directly from fault-caused errors, may be due to additional computational demands associated with error processing, or may be the consequence of subsequent fault-related action such as reconfiguration and repair."* This work addresses the last two cases: evaluating the performance overhead demanded by the RADIC fault tolerance architecture (DUARTE, A., 2007) and the degradation caused by the repair and reconfiguration process. To evaluate performability, Meyer also say that it *"can be either model-based or conducted experimentally via measurements of an actual system."* All evaluation in this work was conducted experimentally via performance measurement under different availability conditions.

1.2. Motivation

Is a fast but fragile system good? Is an available but slow system good? These two questions demonstrate the importance of performance and availability in the current systems, specifically the computer clusters. Due to their correlation, i.e., the former commonly affects the latter and vice-versa, they compound an indivisible binomial for some kind of applications.

Generally, applications designed for parallel systems demand all available computing power and may not accept performance degradation. For example, in systems running under

time constraints, it is as critical to finish the application correctly as to accomplish it before a deadline (a situation that may invalidate the results of the execution). Below are typical examples of applications areas commonly executed in computer clusters.

- *Mission-critical applications*. These applications are crucial for the success of an enterprise. A failure in the application execution may result in a loss of money, serious operational disorder or other unrecoverable damage. These kinds of applications need to be executed in as little time as possible and without failures.

- *Fluid-flow simulation*. This consists to simulate the interaction of large three dimensional cells assemblage, e.g., weather and climate modeling, In weather prediction, for example, it is desirable to start the simulation as late as possible, in order to acquire the most recent data from sensors,. However, if the computation finishes after the expected time, the result data may be useless.

- *Natural behavior simulation*. A notoriously complex area, that makes computers simulate the real world and its interactions. Good examples are the forest fire simulation and individuals' behavior simulation. In the forest fire simulation when applied to fire contention, a delayed result can render the information useless once the fire line has reached the simulated position.

- *Medicine research*. Studies such as protein folding require massive computing power in order to predict the structure of the protein from a known sequence of the protein, being applied in many disease treatments. In this case, as in many others, any delay increases the cost, because the use of parallel machines is generally very expensive.

- *Astronomy.* Simulation of N bodies under the influence of physical forces, usually gravity. It is normally used in cosmology to study the process of galaxy cluster formation. As the number of bodies increases, the simulation is more complex and takes large periods of execution.

For these applications, correctly finishes and the time spent on executions become major issues when planning to perform tasks using parallel computer-based solutions. Therefore, it is reasonable to say that those applications are dependent on the system's *performability*.

Performability makes possible to analyze the effects of providing different protection levels in the performance of applications running on computer clusters. i.e., to analyze the overhead caused by a fault tolerance solution. Figure **1-1** contains a chart depicting the throughput of an application when different levels of protection provided by a fault tolerance

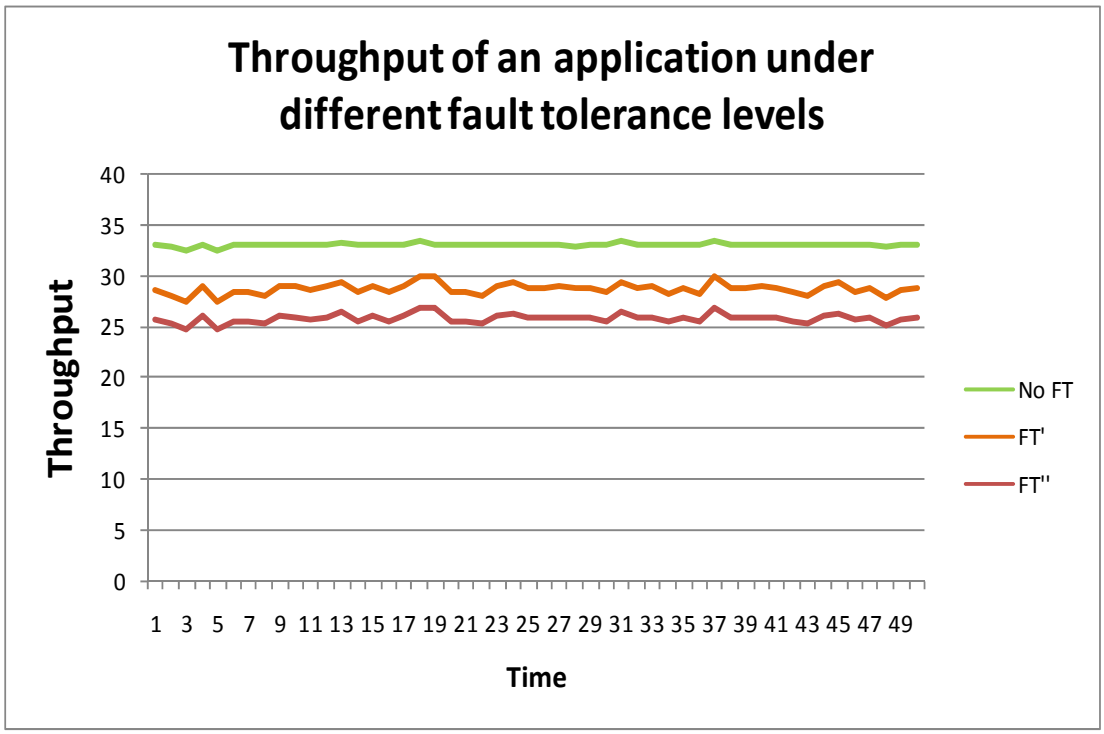


Figure 1-1: Throughput of an application under different fault tolerance levels.

solution is applied, namely “without fault tolerance” (NoFT), “fault tolerance A” (FT’) and “fault tolerance B” (FT’’), where $Availability_{FT''} > Availability_{FT'} > Availability_{NoFT}$. The chart illustrates the typical behavior of applications under a fault tolerance solution, when it causes a performance overhead ($Throughput_{FT''} < Throughput_{FT'} < Throughput_{NoFT}$) because of its activities, e.g., taking checkpoints or logging events. In order to increase the availability provided by a fault tolerance solution, it is usually necessary to aggregate more activities to the solution, e.g., replication of the redundant data (checkpoints and logs), or shorter checkpoint interval. However, these additional activities may affect the system performance even more.

When faults are taken into consideration, and these faults degrade the system’s performance, *performability* metrics can be applied to evaluate a system under faults presence. Figure 1-2 depicts a chart exemplifying the throughput of an application when single or concurrent faults occur against different degrees of availability (including no fault tolerance). This fault tolerance solution is characterized by keeping the system working but with the performance degraded. In this context, time constrained applications may not produce the expected results before their deadlines. In some cases, the degradation may reach unacceptable levels, leading to the need to perform a safe-stop and restart the entire system. Furthermore, the kind of fault uncovered by the availability degree may occur, interrupting the system, i.e., a correlated fault when the availability degree only protects the system from single faults. Preventive maintenance is a common approach to try fault avoidance. Preventive maintenance replaces components at the end of their lifetime, or fault-imminent components detected by sensors or based on historical information. The major issue in this approach is the need to stop an application running in the node containing the component to be replaced (or the own node in many cases). As illustrated in Figure 1-3, even in fault tolerant systems, this activity

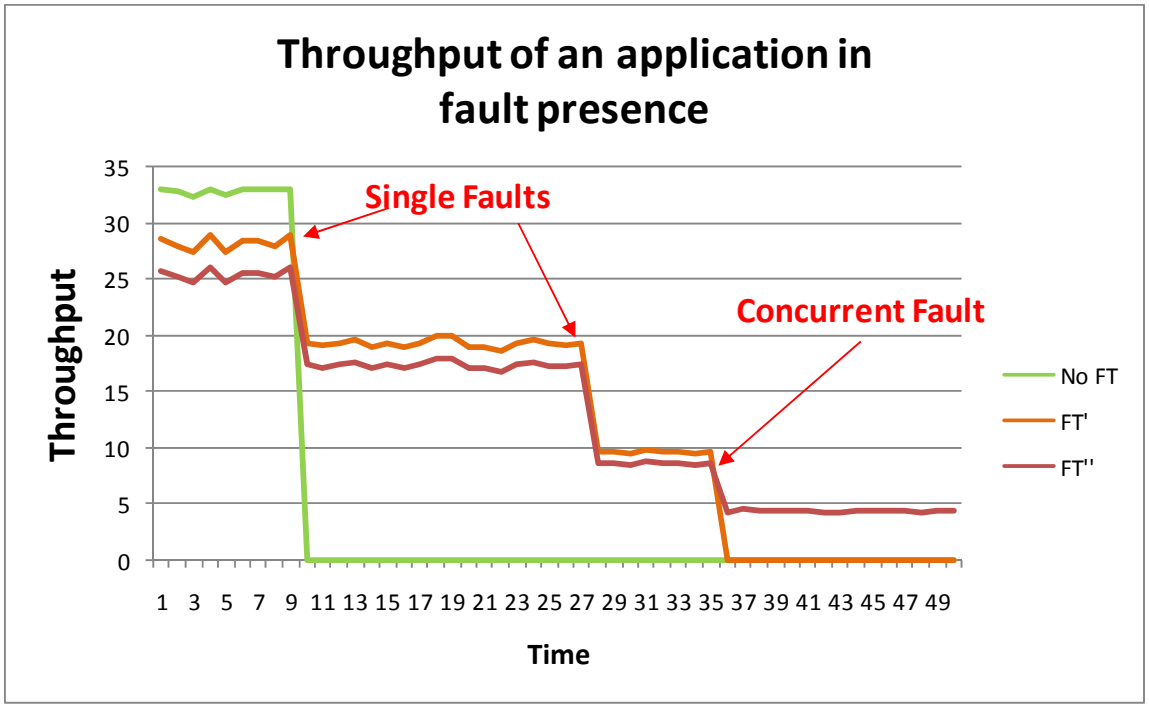


Figure 1-2: Throughput of an application in fault presence

usually demands a complete stop of a part of the computer system and the applications running on it, directly affecting the system's availability and consequently its *performability.*

All three factors - fault tolerance overhead, performance degradation, and maintenance stops - are issues that can affect the system's *performability*, and mean applications may not produce the planned results, i.e. at the expected time. These concerns justify the study of such factors and the research of solutions that may mitigate their effects on computer cluster systems.

Nagaraja et al. (NAGARAJA, K. et al., 2005) proposed a model that allow to quantify this metric when a fault load is injected in different layers of a three-tier computer cluster. In this work, the authors argue that unavailability periods are more relevant than availability periods for comparative performability analysis because two different availability values may be quite similar (perhaps differing by just a fractional order of magnitude, e.g. 99.9% and

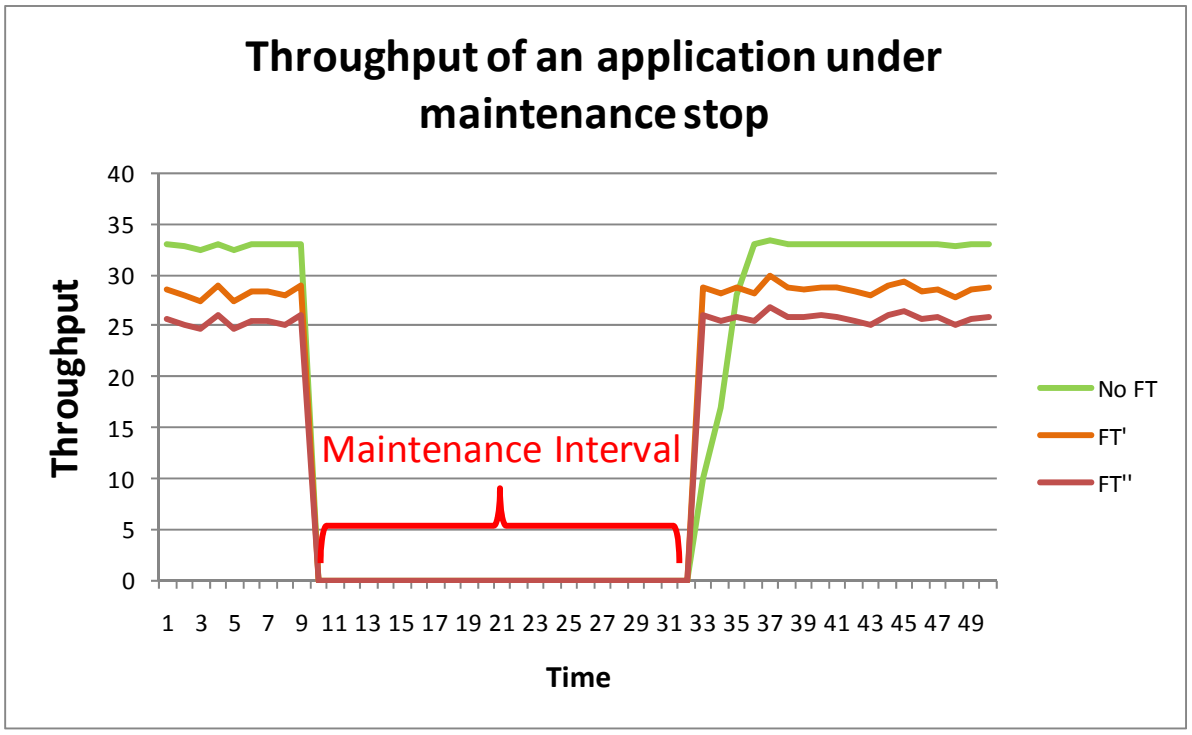


Figure 1-3: Throughput of an application under maintenance stop

99.99%), while the equivalent unavailability values differ by an order of magnitude.

1.3. Goals

Based on the fault tolerance RADIC (Redundant Array of Distributed Independent Fault Tolerance Controllers) architecture (DUARTE, A., 2007), factors influencing the computer cluster's *performability*, such as message log latency, performance degradation because of node losses or availability under concurrent correlated faults are studied and solutions are presented in order to improve *performability* in fault-free and post-recovery situations (after the occurrence of one or more faults).

In fault-free situations, the root causes of the performance overhead are identified and studied. Checkpointing activity is a common cause of the performance overhead in rollback-recovery solutions (OLINER, A.J. et al., 2005) and have been widely studied by the scientific community, resulting in some improvements (ELNOZAHY, E. N. and Plank, J. S., 2004),

(GAO, W. et al., 2005), (DALY, J. T., 2006), (AGARWAL, S. et al., 2004). Another known cause raised in this study is the increase of the message delivery latency caused by the pessimistic logging approach, which demands storing a copy of each message in a repository before continuing program execution. In case of using a higher degree of availability, the data replication of logs and checkpoints also has a strong influence on the performance overhead of the fault tolerance. Facing these factors, it is presented a solution reducing the overhead caused by log-based fault tolerance solutions such as RADIC and a method of imposing a low overhead when increasing the availability provided by RADIC, which directly improves the system's *performability*.

In post-recovery situations, the performance degradation effects of one or more faults in the system configuration after the RADIC recovery process is analyzed. The presented solution avoids configuration changes caused by the recovery process after a fault occurrence, which avoids performance degradation, and is able to restore a changed configuration, which re-establishes a process per node distribution, a factor that may influence the cluster's *performability*. Moreover, the mechanism also allows 'stopless' preventive maintenance to be performed and is completely integrated into the RADIC fault distributed controller. This works transparently and is configurable in order to adapt to the application and system requirements.

The solutions for failure-free situations improve a RADIC-enabled system's *performability* in two ways: a) by reducing the message delivery latency (in many systems, the message delivery latency is crucial to achieve a desired performance) and; b) by decreasing the system unavailability through low-overhead storing of n-replicas of the redundant data over several repositories (SANTOS, G. et al., 2009).

In contrast, after the recovery task, when the application process per node distribution may change, (affecting the system performance and consequently its *performability*), a dynamic redundancy (KOREN, I. and Krishna, C. M., 2007) was incorporated with functionality that enables RADIC, via spare nodes, to protect the system configuration from the changes that a recovery task may generate (SANTOS, G. et al., 2006), (SANTOS, G. et al., 2008).

When the original recovery process of RADIC changes a system configuration, it is proposed that a mechanism allows the re-establishment of the original process distribution. Such a mechanism permits the insertion of a replacement node during the program execution. This inserted node will take the recovered process, and restore the original process distribution.

Furthermore, a solution allowing maintenance tasks to be performed without needing to stop the entire application is also presented (SANTOS, G. et al., 2008). This solution inserts new or updated nodes during the program execution and uses a fault injector to schedule a fault in the node to be replaced just after the next checkpoint. This reduces recovery time by avoiding log processing during the recovery, and will forces the application processes in execution on this node to be migrated to the new node.

The major premise of these solutions is to keep RADIC features such as transparency, decentralization, flexibility and scalability as far as possible. Moreover, the solutions must also: a) impose a negligible overhead in relation to RADIC during failure-free executions. b) provide a quick recovery process when avoiding system configuration changes.

Several experiments were performed with the techniques presented in this work in order to validate their functionality and evaluate their employment in different scenarios.

$$Perfomability_{System} = Avg\_Thruput^{pw} \times \min\left(1, \frac{Target\_Unavail}{Avg\_Unavail}\right)^{uw} \quad (2)$$

In this work, the *performability* of computer clusters using a fault tolerance solution is quantitatively evaluated using the model referred in the previous section. This model takes into consideration the performance measurement of the system S under some situation versus the unavailability of this system in the same situation as represented by Equation (2).

The pipelined log and the process state n-replication were evaluated by a set of experiments involving a simple token pass application, which measured the message delivery latency and a matrix product program measuring the execution time. In each experiment, the overhead generated by these solutions were compared with a regular RADIC configuration and without fault tolerance.

The dynamic redundancy solution was evaluated by comparing the effects of recovery with and without available spare nodes. These experiments observed two measures: overall execution time, and throughput of an application. Different approaches for a matrix product algorithm were applied by using a static distributed Master/Worker and a SPMD approach implementing a Cannon algorithm and an N-Body particle simulation using a pipeline paradigm was executed.

1.4. Outline of this thesis

This thesis contains six chapters organized as follows. Chapter 2 presents state of the art research regarding performability and fault tolerance, and highlights the fault tolerance factors that may influence system's performability and how to measure them. Chapter 3 describes the RADIC fault tolerance architecture, the basis of this work's evaluation, and explains how it operates and how it affects the performability of a system. Chapter 4 details the

issues regarding fault-free performance overhead and performance degradation because of faults, and proposes solutions to improve the performability in each case. The experimental validation and evaluation of proposed solutions is presented in Chapter 5. Finally, Chapter 6 presents the thesis conclusion, summarizing its contributions and stating possible future works.

Chapter 2
Performability and Fault Tolerance

As explained in Chapter 1, this work focuses on the performability analysis of message-passing systems when a fault tolerance solution is applied. In order to clarify and improve the knowledge regarding this topic, this chapter discusses the state of the art of performability and fault tolerance, and highlights its importance, how to measure it and the factors that may influence this class of metrics in a fault tolerant computer cluster.

2.1. The Performability concept

The performance of computer systems has been subject of several studies for a long time (SABETTA, A. and Koziolek, H., 2008), resulting in many forms of evaluation based on techniques that take into consideration factors such as computing power, memory amount, communication structure, and workload. These techniques can be classified in three different groups: analytical modeling, simulation and measurement (KOZIOLEK, H., 2008). However, these studies often assume that a computer`s configuration is always available and remains unchanged during the entire evaluation, which often is untrue because of the probability of faults that can change the initial configuration.

On the other hand, availability has also been target of research, many works have presented different approaches for evaluating system availability (SONG, H. et al., 2006), (SUN, H. et al., 2003), (PIEDAD, F. and Hawkins, M., 2001), (GRAY, J. and Siewiorek, D. P., 1991). Others have presented solutions for increasing the system availability (BOUTEILLER, A. et al., 2006), (SKJELLUM, Y. S., 2004), (FAGG, G. E. and Dongarra, J. J., 2000), (AGBARIA, A. and Friedman, R., 1999), (RAO, S. et al., 1999). Generally, the availability

evaluation focuses on the probability of a fault occurrence and its recovery time, and rarely takes into consideration performance issues, such as the overhead caused by the additional computational resources applied to tolerate faults, time to react to a fault, or the performance degradation caused by the changes in the computer's configuration because of the faults.

The need to evaluate performance considering the system availability (or dependability in a broad sense) led to the definition of a new class of metrics that took the system performance when faults occur. into consideration This issue was the focus of Meyer's work (MEYER, J. F., 1980), which presented the definition of performability as a measure able to allow a unified evaluation of performance and reliability. Later (MEYER, J.F., 1992), Meyer published a performability retrospective presenting a more generic performability definition as a class of metrics that allows a unified evaluation of a system's performance and dependability. Dependability is a term covering many system attributes such as reliability, availability, safety or security (EUSGELD, I. and Freiling, F., 2008). For the purpose of this work, availability is the dependability attribute taken into consideration to improve performability.

Therefore, performability allows the evaluation of a computer cluster in a more realistic, complete and accurate way, since it takes into consideration factors such as faults and performance degradation. In this work, performability is also used to evaluate the impact of having to tolerate such faults, widening the appliance of the performability concept.

2.2. Evaluating performability

Reasonable questions regarding the performability are how is it possible to measure it? And which metrics must be used to evaluate its system? Today`s literature (EUSGELD, I. et al., 2008), (MEYER, J. F., 1995) presents a number of performability metrics and models that may be used to analytically evaluate the behavior of computer systems under the pres-

ence of faults, and which can be applied to predict system performance. On the other side, performability may be evaluated in a more pragmatic way, being realized experimentally via indirect measurements of a system (MEYER, J. F., 1995), such as in the analysis presented by Nagaraja (NAGARAJA, K. et al., 2005). This work focuses in the latter case, and experimentally measures system performance indexes (such as throughput, execution time, and overhead) under different levels of fault tolerance (independent faults, concurrent correlated faults, dynamic redundancy, and preventive maintenance). The metrics and how to obtain the performability components are detailed below.

2.2.1. How to measure performability in computer clusters

Performability can be initially evaluated by measuring availability and performance separately and then, applying a model to join these two metrics. Concepts regarding availability and performance evaluation are presented below and later, the model chosen to evaluate performability along this work will be demonstrated

Availability

Availability is one of the major requirements when using parallel computers. Any user of the applications exemplified in Chapter 1 expects to have the system available during the entire execution of its work. There are different classifications of availability (LIE, C. H. et al., 1977):

1. *Instantaneous (or Point) Availability*. The probability that a system will be available at the random instant T.
2. *Average Up-Time Availability (or Mean Availability)*. The fraction of a specified time interval that the system is available.

3. *Steady State Availability*. Instantaneous availability when the time approaches infinity. It is the lower base in the "bath-tube" curve abstraction.
4. *Inherent Availability*. Steady state availability, but considering the recovering downtime
5. *Achieved Availability*. Similar to the previous one, but also includes the maintenance downtimes.
6. *Operational Availability*. The ratio between uptime and total time after a period of time.

This work will use inherent availability, since it considers recovering activity. The Equation (3) represents mathematically inherent availability. According to this equation, inherent availability is given by the relationship between Mean Time Between Failures (MTBF) and Mean Time to Recover (MTTR) as follows.

$$A = \frac{MTBF}{MTBF + MTTR} \tag{3}$$

The MTBF is derived from the failure rate (λ) as can be seen in Equation (4) and may be related to a single component or an entire system. For a system, it is also called the System Mean Time Between Failures (SMTBF) and is calculated according to Equation (5), which allows to deduce that a system with many components will be more susceptible to faults. In cases of systems with different component' MTBF, the average value is used.

$$MTBF = \frac{1}{\lambda} \tag{4}$$

$$SMTBF = \frac{\overline{MTBF}}{N} \tag{5}$$

The MTTR is the average time spent returning the system to an operational state, and is dependent on the system's management structure. If the system is unmanaged the MTTR

can reaches its highest values, i.e., the reparation of a single node can take more than one week. In well-managed systems (with ready spare components and a support staff), this value may decrease to a few hours. In high-availability systems (using automatic and transparent fault tolerance), this value reaches as little as a few minutes.

Usually, availability is classified according to the percentage of time in an operational state. Such a percentage is also known as “the nines classification”. TABLE 2-1 summarizes the current system classification according to availability class.

From Equation (3) it is possible to deduce that there are two ways of increasing the availability of a system: either by increasing the reliability of its components or by decreasing the time for repair. To increase the components reliability generally means using highly expensive equipment, which sometimes becomes unfeasible to implement. Therefore, fault tolerance plays its role by reducing the MTTR. Indeed, the only way to reach a theoretical 100% availability is by the MTTR equaling zero, since a component with infinite MTBF is currently unfeasible.

TABLE 2-1: Availability classes classification (GRAY, J. and Siewiorek, D. P., 1991)

System Type	Unavailability (minutes/year)	Availability (percent)	Availability Class
Unmanaged	50,000	90	1
Managed	5,000	99	2
Well-managed	500	99.9	3
Fault-tolerant	50	99.99	4
High-availability	5	99.999	5
Very-high-availability	.5	99.9999	6
Ultra-availability	.05	99.99999	7

Performance

There are three different techniques for evaluating the performance of a system (KOZIOLEK, H., 2008): analytical modeling, simulation and measurement.

In the **analytical modeling approach**, performance models are constructed using stochastic Petri Nets, queuing networks or some other stochastic process. Measured or estimated values are used as input parameters, and the expected performance is calculated. The main advantage of this method is being able to predict quickly the system performance at low cost. However, this approach may offer results with low precision and some models can be complex when trying to represent the real world.

Simulation also uses models to predict the performance of a system. Such models represent each activity of the system affecting performance. The accuracy of simulation is better than the analytical approach, but requires greater effort. Usually, simulation software is used to implement the model. The main advantage is ease of changing system characteristics and evaluating the resultant performance.

Measurement is the approach used in this work, since it provides better accuracy and the necessary real implementation is accessible. The metric used for the measurement depends of the system's characteristics. Some examples are **throughput** as presented in (NAGARAJA, K. et al., 2005), **task completion time** (HAVERKORT, B. R. et al., 2001) or **response time** (SABETTA, A. and Koziolek, H., 2008). The major concern when using different metrics for performance is the meaning of the measured values i.e., for throughput measures, higher values are better while for task completion time, lower values are better. A possible workaround is to know how many tasks were executed during an application, and to divide such a value by the time the application took to accomplish its work. This results in

some kind of throughput. In this work, this workaround is applied whenever a comparison between different kinds of metrics is necessary.

2.3. Performability related factors

Several factors may influence the performability of a system. In this thesis, the evaluation is restricted to faults and fault tolerance. The theoretical concepts regarding these factors are explained and discussed below.

2.3.1. Faults

One of the most relevant factors influencing system performability is fault occurrence, so it is necessary to define what fault means. Generally, the terms fault, error and failure are mentioned interchangeably. By definition, failure is the undesirable behavior of a system (the system does not produce the expected results, for example a software abnormal ending). An error is the generating event which leads to a failure, unless it applies corrective actions (for example a programming error leads to an abnormal ending except when then the error is caught and treated). Finally, a fault is a system defect with the potential to generate errors. Thus, a fault may cause an error, which may cause a failure.

A fault's effects can be analyzed from different points of view. In a computing system formed by interdependent components, the occurrence of a fault in any of its components leads to an error and consequently to an entire system failure. These kinds of systems are also called fail-stop systems. On the other side if the components of a computing system are independent, a fault occurrence in one component will cause a failure of this component exclusively and the system will remain operational with the possibility of some performance degradation. The former kinds of systems may be turned into the latter if the faulty component can be replaced and a fault tolerance scheme applied.

In cases of independent components or when fault tolerance is applied, the system may present different levels of performance according to the fault distribution and the fault tolerance scheme adopted. In cases of interdependent components the system presents a binary behavior: it is up when no faults occur or is completely down when a fault occurs. In the first case, performability may be affected by performance degradation or availability reduction, i.e., if a system can tolerate just one fault, after this fault the system is unprotected and turns into a fail-stop system. In the last case is impracticable to evaluate performability, however, this case is useful for comparison purposes

.

2.3.2. Fault Tolerance

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

The fault tolerance may influence system performability in two ways. The first way is by increasing availability. When fault tolerance is applied in a fail-stop system, despite faults occurring, the perceived availability will be greater than before. The resultant availability will depend on the chosen approach for the fault tolerance. If it is using a n-redundancy of hardware, the availability will depend on how many redundant devices there are, or if data redundancy it is chosen, availability will depend on how the failed component is recovered and the state saving frequency. A study of fault tolerance in message-passing systems, which is the kind of systems considered in this work, follows.

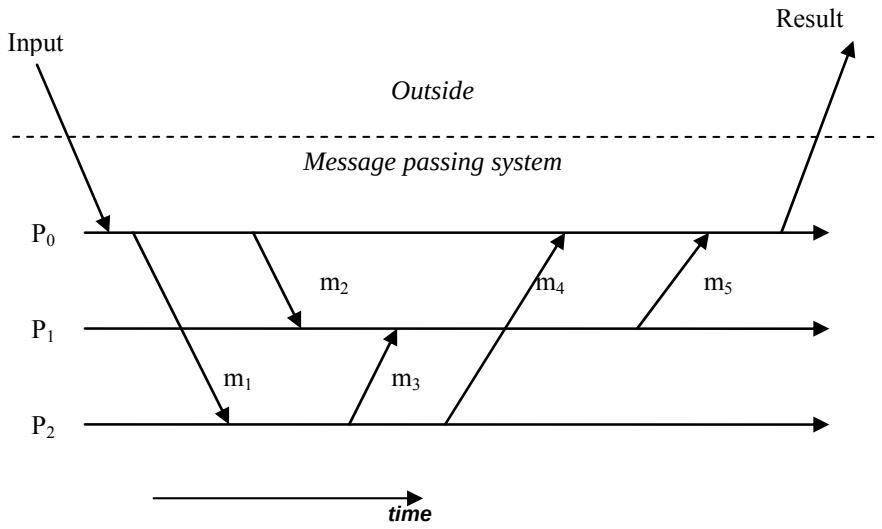


Figure 2-1:A message passing with three processes interchanging messages.

2.3.3. Fault Tolerance in Message-Passing Systems

Message-passing is a common technique used in parallel computers to provide communication between concurrent processes. This technique makes the following assumptions:

- The processes only have access to their own local memory;
- All communication between the processes comprises sent and received messages; and
- The data interchange requires cooperative actions in each process, meaning that a sent message needs a corresponding receive in the other process.

With these simple assumptions, message-passing is widely used for parallel computing because it fits well in cluster of workstations or supercomputers interconnected by a network. Figure 2-1 exemplifies the functioning of a simple message-passing system with three processes (P_0, P_1 and P_2) sending and receiving messages (diagonal arrows labeled from m_1 to m_5) through the timeline (horizontal arrows). Such a system receives an input from outside, and starts processing this input using a message-passing mechanism and provide a result to outside.

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

There are many techniques developed for increasing overall reliability and providing high availability for message-passing distributed systems including data or hardware replication protocols, self-stabilizing protocols and rollback-recovery protocols (KOREN, I. and Krishna, C. M., 2007). Rollback-recovery is widely studied and commonly used to provide fault tolerance for message-passing systems, while data replication usually improves fault tolerance at the system level.

2.3.4. Rollback-recovery

Rollback-recovery is a protocol or technique for providing fault tolerance based on returning the program execution to a point just before the fault occurrence, and in some ways, retrying the computation. According to Shooman (SHOOMAN, M. L., 2002) there are four basic types of rollback-recovery techniques:

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

Journaling – This periodically stores all inputs to the system. In the case of a fault, the processing may be repeated automatically. This technique is a usual feature in most word processors and some operating systems.

Retry – This technique is more complex and supposes that the fault is transient and in a subsequent moment, the system can operate normally. It performs the action repeatedly for a maximum number of attempts or until a correct result is achieved. Disk controllers are a good example of retry.

Checkpoint – This technique is an improvement on the reboot technique. In this approach, the system state is saved periodically, so the application or the system only needs to return to the most recent checkpoint before the fault.

The checkpoint approach becomes more suitable for parallel systems because of the characteristics of the applications running in these systems, which usually execute over a long period. Performing checkpoint is a more difficult task in distributed systems compared with centralized ones (KALAISELVI, S. and Rajaraman, V., 2000) because distributed systems are compounded by a set of independent processors with individual lines of execution. Furthermore, there is no global synchronized clock between them to allow starting a checkpoint at same time, and save the global state of the parallel application.

2.3.4.1. Basic concepts

Before continuing, important concepts involving the rollback-recovery in distributed systems should be introduced. These concepts will be useful for understanding the influence of fault tolerance on the system's performability and how the proposed solutions work.

Checkpoint

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

Stable storage

The use of checkpoints to perform rollback-recovery generally requires that a system state must be available after the failure. In order to provide this feature the fault tolerance techniques suppose the existence of a stable storage, which survives any failures in the system. Although a stable storage is usually confused with physical disk storage, it is just an abstract concept (ELNOZAHY, E. N. et al., 2002) and can be implemented in different ways:

- It may be a disk array using RAID, which tolerates certain number of non-transient failures;
- If using a distributed system, a stable storage can be performed by the memory of a neighbor node; or
- If it only needs to tolerate transient faults, a stable storage can be implemented using a disk in the local machine.

Consistent system state

The major goal of a rollback-recovery protocol is to return the system to working operation. Rollback-recovery is simple to implement in a single process application, but becomes a hard task in a computer cluster, with many processes executing in parallel. In parallel applications using message-passing, the state of the system comprises the state of each process running in different nodes and communicating between them. Therefore, taking a checkpoint of a process individually may not represent a snapshot of the overall system.

Hence, a consistent system state can be defined as one in 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 accused of the message sending too. During a failure-free execution, any global state taken is a consistent system state.

Domino effect

The *domino effect* (KOREN, I. and Krishna, C. M., 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 the communication with other processes, forcing these processes to roll back to checkpoint before this communication. This situation may continue to happen until reach the initial of the execution. This event is exemplified by the situation depicted in Figure 2-2, which shows an execution in which processes take their checkpoints (represented by blue circles) without coordinating with each other.

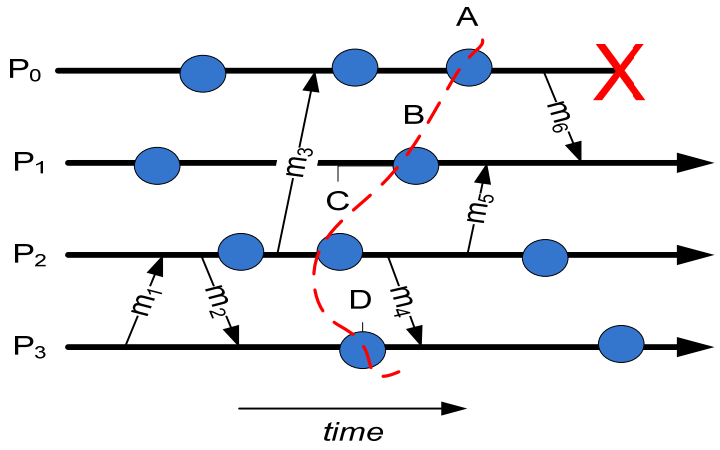


Figure 2-2: Domino effect

It is considered that processes start with an initial checkpoint. Suppose that process P_0 fails and rolls back to checkpoint A. The rollback of P_0 invalidates the sending of message m_6, and so P_1 must roll back to checkpoint B in order to "invalidate" the receipt of message m_6. Thus, the invalidation of message m_6 propagates the rollback of process P_0 to the process P_1, which in turn invalidates message m_5 and forces P_2 to roll back as well. Because of the rollback of process P_2, process P_3 must also rollback to invalidate the reception of message m_4. These cascaded rollbacks can continue and may eventually lead to a *domino effect*, which forces the system to roll-back to the beginning of the computation, despite all saved check-points.

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

In-transit messages

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

Logging protocols

Log-based rollback-recovery is a strategy used to avoid the domino effect caused by uncoordinated checkpoints. Logging protocols are a set of protocols which take message logs as well as checkpoints. Such protocols are based on the *piecewise deterministic* (PWD) assumption (STROM, R. and Yemini, S., 1985). Under this assumption, the rollback-recovery protocol can identify all the nondeterministic events executed by each process. For each nondeterministic event, the protocol logs a determinant containing all necessary information to replay the event 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 had 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 consistent with the input and output interactions that occurred up until the fault.

2.3.4.2. Checkpoint-based protocols

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

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 most convenient (for instance, when the process's state is small) and does not care about the checkpoints of the other processes. Zambonelli (ZAMBONELLI, F., 1998) evaluates several uncoordinated checkpoint strategies.

The uncoordinated strategy simplifies the checkpoint mechanism of the rollback-recovery protocol because it provides independence for each process and manages its checkpoint without negotiation with the other processes. However, such independence of each process comes at a cost:

- There is the possibility of a domino effect and all its consequences;
- A process can take a useless checkpoint since it cannot guarantee by itself that a checkpoint is part of a consistent global state. These checkpoints will overhead the system but will not contribute to advance the recovery line;

- It is necessary a garbage collection algorithm to free the space used by checkpoints that are useless; and.
- It is necessary a global coordination to compute the recovery line, which can be very expensive in applications with frequent output commit.

Coordinated checkpointing

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

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

Although straightforward, this scheme can yield a large overhead. An alternative approach is to use a non-blocking checkpoint scheme such as the proposals by (CHANDY, K. M. and Lamport, L., 1985) and (ELNOZAHY, E. N. and Zwaenepoel, W., 1992). However, non-blocking schemes must prevent the processes from receiving application messages that make the checkpoint inconsistent.

The scalability of coordinated checkpointing is weak because all processes must participate in every checkpoint (MALONEY, A. and Goscinski, A., 2009) and transmit their checkpoints to a stable storage that generally is centralized, an activity which may cause a communication bottleneck.

Communication-induced checkpointing (CIC)

The communication-induced checkpointing protocols do not require checkpoints to be coordinated and avoid the domino effect. There are two kinds of checkpoints for each process: local checkpoints that occur independently and forced checkpoints that must occur 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 will never contribute to the recovery of the system from failures) although they do consume resources and cause performance overhead.

As opposed to coordinated checkpointing, CIC protocols exchange no 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, L. et al., 1999) For CIC protocols, one can prove that a checkpoint is useless if it is part of a *Z-cycle*.

Two types of CIC protocols exist: indexed-based coordination protocols and model-based checkpointing protocols. It has been shown that both are fundamentally equivalent, (HELARY, J. M. et al., 1997) although offer some differences in practice (ALVISI, L. et al., 1999).

2.3.4.3. Log-based protocols

These protocols require 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 because there may be missed or orphan messages related to the recovered process (ELNOZAHY, E.N. and Zwaenepoel, W., 1994). During the process's recovery, the logged messages will be properly recovered from the message log, meaning the process can resume normal operation and the system will return to a consistent state (JALOTE, P., 1994).

Log-based protocols consider a message-passing based application to be a sequence of deterministic state intervals, each starting with the execution of a nondeterministic event (JALOTE, P., 1994). Each nondeterministic event relates to a unique *determinant*. In message-passing systems, the typical nondeterministic event that occurs to a process is the receipt of a message from another process (*message logging* protocol is the other name for these protocols.) Sending a message, however, is a deterministic event. For example, in Figure 2-1, the execution of process P_3 is a sequence of three deterministic intervals. The first one is the process' creation and the other two start with the receipt of messages m_2 and m_4. The initial state of the process P_3 is the unique determinant for sending m_1.

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

Log-based protocols guarantee that upon recovery of all failed processes the system contains no orphan processes. A process is orphan when it does not fail and its state depends on the execution of a nondeterministic event whose determinant cannot be recovered from a stable storage or from the volatile memory of a surviving process (ELNOZAHY, E. N. et al., 2002). There are three classes of log-based protocols: *pessimistic*, *optimistic* and *causal*.

Pessimistic log-based protocols

These protocols assume a pessimistic behavior, supposing that a failure may occur after any nondeterministic event in the computation. In their most simple form, pessimistic protocols log the determinant of each received message before the message influences 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, E. N. et al., 2002). Such a condition ensures that orphan processes never exist in systems using pessimistic log-based protocols.

Processes also take periodic uncoordinated 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 recovery procedure, the process uses the logged determinants to recreate the pre-failure execution, without needing any synchronization between processes. The checkpointing interval influences directly in the overhead imposed by fault tolerance, creating a dilemma: if checkpoints are taken in short periods, it causes greater overhead during a failure-free execution, but the recovery process will be less *expensive*. Furthermore, the checkpointing interval may also be limited by the message log

storage size, i.e, if there are many messages and the log size is reaching its storage limit, a checkpoint must be taken in order to perform a garbage collection.

Synchronous logging enables the observable state of each process to be always recoverable. This property has to four advantages to balance the high computational overhead penalty (ELNOZAHY, E. N. et al., 2002):

- Recovery is simple because the effects of a failure only influences the processes that fail;
- Garbage collection is simple because the process can discard older checkpoints and determinants of received messages that are before the most recent checkpoint;
- Upon a failure, the failed process restarts from its most recent checkpoint which limits the extent of lost computation; and
- There is no need for 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 that the message was delivered. Such an enlargement may be relevant if the application is communication bounded. In order to reduce the overhead caused by synchronous logging, the fault tolerance system may apply a *Sender Based Message Logging* model that stores the log in the volatile memory of the message sender, supposing it is a reliable device. In this case, the recovery process is more complex and needs to involve each machine that has communicated with the failed process.

Optimistic log-based protocols

In contrast, optimistic log-based protocols suppose that faults occur rarely, which relaxes the event log, but it allows the orphan processes appearing caused by failures in order to reduce the fault-free performance overhead. However, the possibility of appearing orphans processes makes recovery process, garbage collection and output commit more complex (JALOTE, P., 1994). In optimistic protocols as in pessimistic protocols, every process takes checkpoints and logs message asynchronously (ALVISI, L. and Marzullo, K., 1998). Furthermore, a volatile log maintains each determinant while the application's processes continue their execution. There is no concern if the log is in the stable storage or the volatile memory. The protocol assumes that logging to the stable storage will complete before a failure occurs (hence its optimism).

If a process fails, the determinants in its volatile log will be lost, and the state intervals started by the nondeterministic events corresponding to these determinants are unrecoverable. Furthermore, if the failed process sent a message during any of the state intervals that too 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 (MALONEY, A. and Goscinski, A., 2009), (JALOTE, P., 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 which can complicate the garbage collection policy.

The recovery mechanism in optimistic protocols can be either *synchronous* or *asynchronous* (ELNOZAHY, E. N. et al., 2002). Each one is explained bellow.

Synchronous recovery

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

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

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

Asynchronous recovery

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

Asynchronous recovery can produce a situation called exponential rollbacks. Exponential rollbacks occur when a process rolls back an exponential number of times because of

a single failure. The asynchronous protocol eliminates exponential rollbacks by either distinguishing failure announcements from rollback announcements or piggybacking the original rollback announcement from the failed process on every subsequent rollback announcement that it broadcasts.

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, is either in a stable storage or available locally to that process (MALONEY, A. and Goscinski, A., 2009). 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 unstable determinants in its volatile log on the messages it sends to other processes. On receiving a message, a process first adds any piggybacked determinant to its volatile determinant log and then delivers the message to the application.

2.3.5. Data replication

Data replication, applied in computer clusters, consists in to keep identical copies of relevant data on two or more nodes (KOREN, I. and Krishna, C. M., 2007). Despite the improvement of fault tolerance, such a technique introduces new problems: consistency and replica management (JALOTE, P., 1994). Consistency means that all copies on the nodes must have the same data, and for that to occur, a data replication scheme must implement a consistency control algorithm, which in turn implements a replica control method to ensure

that the operations performed in the data will be performed on multiple copies of such a data. The major concern about in the consistency control algorithm is failure. Two kinds of failure must be taken into consideration: node failures and communication failures. Node failures avoid the access to the data, while network link failures generate network partitioning. For the purpose of this work only node failures are taken into consideration.

There are two approaches for masking node failures,: optimistic and pessimistic. Optimistic approaches are suited to network link failures. In these cases the replica control method assumes that the operations performed in different partitions will not conflict. If inconsistency arises, the replica control method will try to resolve it using strategies such as version vectors or logging the writing and reading operations. Pessimistic approaches avoid occurrence of inconsistency by controlling access to data. Hence it will be applied in both aforementioned failures cases. Common pessimistic approaches are active replication, voting and primary site.

Active replication

In the active replication approach, all replicas are available for reading and writing operations from any source, therefore the replica control method must ensure that copies are always synchronized. A common method for providing such synchronization is atomic broadcast, i.e., all operations must be performed in all copies before the system can continue the processing. Such broadcast usually needs a scheme for order and an agreement that may lead to undesirable overhead. Furthermore, as different requests may be performed on different replicas, the concurrent request must be controlled in order to avoid inconsistencies. Typically, a two-phase locking protocol is used. When a request for an operation is made on a

replica, it first performs a lock operation, avoiding the reception of a new request, and when the operation finishes the replica is unlocked.

Voting

Voting is a technique that allows writing operations do not have to be performed in all replicas at once, rather, a majority group of replicas is elected to perform the writing operation. In each writing operation, a timestamp or a version number is added. During the read operations there is no need for all replicas to be up-to-date. Analogously, a majority group of replicas is chosen for the read operation and a request for votes is sent. The replicas then reply with the data and the timestamp or version number, and the requester uses the data from the replica with the highest version number or latest timestamp. Since read and write groups intersect, it ensures that at least one replica is up-to-date.

If the number of replicas is too large, reading and writing operations may take a long time to be performed. One solution for this situation is a variation of the original scheme based on a hierarchical voting organization. In such an approach, a set of nodes is organized as an m-level tree. As seen in Figure 2-3, the copies are stored at the leaves of the tree, and virtual nodes are added at higher levels until they reach the top level (level 0 or root). The organization is made in such a way that each node on the same level has the same number of children. During the reading and writing operations, a recursive algorithm assembles a majority group with the leaves of the tree. The voting for the latest information is made on each level, and only depends on the leaves associated with the nodes on that level.

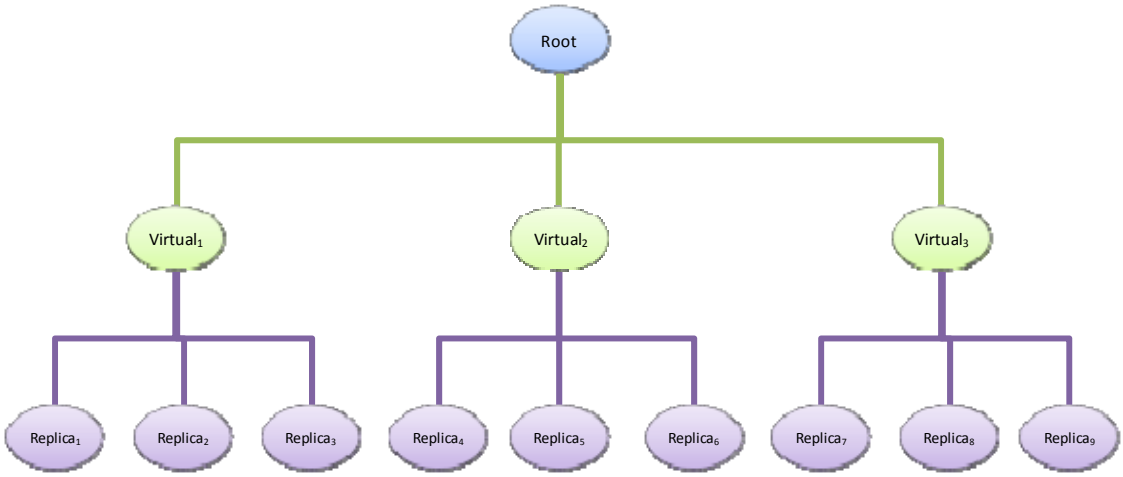


Figure 2-3: A tree for hierarchical voting with m=3

Primary site approach

In the primary site approach, also known as the primary-backup approach, there is only one active node and k+1 nodes designated as backups. All writing and reading operations are made to the primary node and this node forwards the request to the backup nodes and waits an acknowledgment. Hence, the nodes are organized in a logical linear organization. This arrangement is important to ensure the processing order: first the primary, then each backup.

If a failure occurs in the primary site, one of the backup nodes is elected as primary, in a simple approach, the next node on the logical linear organization will be chosen as the primary. If a failure occurs during replication, when the primary site is waiting for the acknowledgement, action is necessary in order to keep the consistency between the replicas. One simple action is the rollback of the last operation in the backup nodes. If one of the backups fails during a replication, it never returns an acknowledgment leading the primary to wait indefinitely. A timeout feature must be implemented, to remove such a node from the replication set.

2.3.6. Current researches

Fault tolerance becomes a major issue in the high performance computing area. Hence, many works has been developed in order to provide fault tolerance for parallel systems by taking into consideration factors that may influence the performability such as performance overhead. Some of the current research in this area is detailed below.

Chtepen et al in (CHTEPEN, M. et al., 2009) propose an adaptive solution combining data replication and checkpointing in order to improve the resource utilization efficiency in fault tolerant distributed systems. This solution dynamically switches between performing checkpointing and data replication according to some parameters such as available CPUs, system load or number of active replicas. Although this solution is designed to provide efficient resource utilization, it can also reduce the overhead of fault tolerance adapting it in function of the resources stability.

FT-Pro (LI, Y. and Lan, Z., 2006) is a fault tolerance solution that combines roll-back-recovery and failure prediction to take action at each decision point. Using this approach, the solution aims to retain the system performance, avoiding excessive checkpoints, and consequently improves the system's performability. Three different preventive actions are currently supported: process migration, coordinated checkpoint using central checkpoint storages and no action. Each preventive action is selected dynamically in an adaptive way intending to reduce the overhead of fault tolerance. FT-Pro works an initially determined and static number of spare nodes.

The solution proposed in (IZOSIMOV, V. et al., 2006) uses a combination of checkpointing and active replication for distributed embedded systems. Such systems have similar characteristics to the computer clusters, such as the use of message-passing for communica-

tion between nodes. In that work, the authors target critical applications, providing fault tolerance and increasing availability without increasing the resource utilization. Differently of solutions presented in this work, this technique lower the overhead by reducing the number of checkpoints in some process and replicating it in an available node. The solution performs well when there are inter-process dependencies that causes some idle processing time, which may be used in active replication.

Intelligent Checkpoint Engine **(ICE)** proposes a reliability-aware checkpointing strategy to obtain a performability improvement by performing fewer checkpoints (LIU, Y. et al., 2005) . The main idea behind ICE is to determine a checkpoint interval sensitive to the system's failure rate, and place a checkpoint as close as possible to the next failure. This means there will be fewer checkpoints, which reduces the performance overhead caused by this activity. ICE is implemented over the HA-OSCAR architecture. The kernel of ICE is an optimal checkpoint function that takes into consideration a failure rate calculated with failure information received from HA-OSCAR cluster management. Such a function then makes a decision based on a stochastic reliability model defining a checkpoint interval.

The **Score-D** checkpoint mechanism is a fault tolerance solution used in the Score Cluster System Software that implements a distributed coordinated checkpoint system (KONDO, M. et al., 2003). In Score-D's checkpointing algorithm, each node parallelly stores its checkpoint data into the local disk. In addition, it saves redundant data in a neighbor node to ensure the reliability for non-transient failures. This redundancy is achieved through parity generation. In the recovery task, the system uses the parity data distributed over the nodes to reconstitute the checkpoint image and restart the process in a spare node allocated statically at the program start. A server is in charge of sending a heartbeat to each node in order to detect

failures. The initial solution has a clear bottleneck caused by disk writing, so Gao (GAO, W. et al., 2005) proposed an optimization using a hierarchical storage approach combined with diskless checkpointing for transient failures tolerance. According to their results, such optimization has improved the checkpointing performability.

MPICH-V2, an improvement on MPICH-V1, implements the sender-based pessimistic log (the computing node now keeps the message-log) and aims to reduce the performance overhead (BOUTEILLER, Aurélien et al., 2003). It is well suited for homogeneous network large-scale computing. Unlike its predecessor, it requires fewer stable components to provide a good performance in a cluster. MPICH-V2 replaced the channel memories concept by event loggers to ensure the correct replay of messages during recovery. It is formed by additional components: dispatcher, checkpoint servers, and computing/communicating nodes. The dispatcher is responsible for launching the entire runtime environment, and performs a fault detection task by monitoring the runtime execution. The architecture assumes neither central control nor global snapshots. The fault tolerance is based on an uncoordinated checkpoint protocol that uses centralized checkpoint servers to store communication context and computations independently.

MPICH-VCL is designed for extra low latency dependent applications (BOUTEILLER, Aurélien et al., 2003). It uses a coordinated checkpoint scheme based on the Chandy-Lamport algorithm (CHANDY, K. M. and Lamport, L., 1985) to eliminate overheads during fault-free execution. However, it requires restarting all nodes (even non-crashed ones) in the case of a single fault. Consequently, it is less fault resilient than message logging protocols, and is only suited for medium scale clusters.

LAM/MPI a component architecture called System Services Interface (SSI) that allows to checkpoint an MPI application using a coordinated checkpoint approach (SQUYRES, J. M. and Lumsdaine, A., 2003) (BURNS, G. et al., 1994). This feature is not automatic, and needs a back-end checkpoint system. In cases of failure, all applications nodes stop and a restart command is needed. LAM/MPI demands a faulty node replacement. This procedure is neither automatic, nor transparent.

MPICH-V1, the first implementation of MPICH-V, has a good application in large scale computing using heterogeneous networks (BOSILCA, G. et al., 2002). Its fault tolerant protocol uses uncoordinated checkpoint and remote pessimistic message logging. MPICH-V1 is well suited for desktop grids and global computing as it can support a very high rate of faults. As this solution requires a central stable storage, it requires a large bandwidth that becomes the major drawback for this implementation.

2.4. Discussions

This thesis addresses the evaluation of performability using fault tolerance in two situations: fault-free and under the presence of faults. Fault-free analysis is unusual since performability is commonly associated to the presence of faults. However if one considers that the use of a fault tolerance solution is due to the probability of faults occurring, the performance overhead caused by this solution must be considered in order to evaluate the system's performability.

In the literature there are many approaches for evaluating performability, some of them are analytical-based models, Haverkort et al in (HAVERKORT, B. R. et al., 2001) presents a variety of analytical performability models as follows:

The steady-state performability (SSP) is given by $SSP = \sum_{i \in S} \pi_i r_i$, while the transient or point performability (TP) is given by $TP(t) = \sum_{i \in S} p_i(t) r_i$. If the model considers state absorption, the mean reward to absorption (MRTA) is given by $MRTA = \int_0^\infty r_{X(s)}\, ds$ and the cumulative performability is defined as $P(t) = \int_0^t r_{X(t)}\, ds$. For all these models, i is a state $\in S$ (set of possible states), π_i is the steady-state probability of residing in state i, with $p_i(t)$ the probability of residing in i at time t. X is a continuous-time Markov chain and $r_{X(t)}$ is a Markov reward process.

On the other side, there are pragmatic ways to evaluate performability, usually based on measurements. Kondo et al in (KONDO, M. et al., 2003) define a performability model to evaluate their checkpointing technique as $Performability = \frac{CP\_itvl}{CP\_itvl+CP\_time} \times \frac{MTBF}{MTBF+MTTR}$, where CP_itvl is the checkpointing interval and CP_time is the time spent to take a checkpoint. Soares and Pereira in (SOARES, L. and Pereira, J., 2005) use simulation to evaluate middleware performability. They measure the number of committed transactions within a fixed delay and use this as a performability metric. Nagaraja et al in (NAGARAJA, K. et al., 2005) propose the performability model (Equation (6)) adopted in this thesis. This model assumes that unavailability rather than availability periods are most relevant for comparative performability analysis. This assumption is reasonable because two different availability values may be similar, differing by just a fractional order of magnitude (e.g. 99.9% and 99.99%) while the equivalent unavailability values differ by an order of magnitude. *pw* and *uw* are performance and unavailability weights used to adjust the model facing the application and user needs. Therefore, for some applications, such as mission-critical, the unavailability weight may be defined higher than to normal applications, emphasizing the involved risk

factor of these applications. In this work, *pw* and *uw* are adjusted according to each scenario and *Target_Unavail* has the value 0.001% (as desirable as a high-availability system).

$$Perfomability_{System} = Avg\_Thruput^{pw} \times \min\left(1, \frac{Target\_Unavail}{Unavail}\right)^{uw} \quad (6)$$

2.4.1. Considerations regarding fault tolerance

Each one of the checkpoint-based protocols may influence system's performability differently. The uncoordinated approach may impose lower overhead in failure-free executions, because there are no coordination costs and consequently no scalability issues. However in the presence of faults, the recovery process may lead to a re-execution of the entire application, resulting in larger recovery times and consequently reducing the perceived system availability.

The coordinated approach penalizes fault-free performance because the coordination process' duration increases proportionally with the number of application processes. However in the presence of failures, the recovery process is simpler, reducing the recovering duration to the time needed to roll back to the last checkpoint, i.e., the elapsed time since the last checkpoint until the failure moment. Such behavior reduces the unavailability period in comparison to uncoordinated approach.

The communication-induced protocol is more complex to evaluate, because the number of taken checkpoints is unpredictable due to the forced checkpoints, dependent on the application's communication behavior. Since the recovery line may be discovered during the running, the recovery duration is similar to the coordinated approach.

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 simplici-

ty of recovery and garbage collection schemes, as well as its potential for rolling back correct processes. All of these factors influence directly the system performability because of the performance overhead, which can be considered a kind of degradation, or the recovery duration, that implies possible unavailable time.

Pessimistic logging has some advantages under the presence of faults. As a failed process may recover independently of the other, the fault effects are contained and the individual recovery process may influence the system's unavailable time one way or another depending on how coupled the application is. Nevertheless this recovery time is limited by the elapsed time since the last checkpoint in the worse case, i.e., due tothe process' interdependencies the entire system must wait for the recovery of one process. The main drawback of this solution is the performance overhead in applications with high amounts of communication, because of the increased message delivery latency.

In contrast to the pessimistic option, optimistic logging reduces the performance overhead in fault-free executions by avoiding synchronously logging every determinant event in a stable storage. However it may have some performance overhead if it uses dependency tracking algorithms during the execution. As always, there is a tradeoff regarding the recovery process. In this case, it is penalized by the need to gather the events that may be distributed over several places, calculate the dependency between the processes or by the possible exponential rollbacks. Such a complex recovery may lead to large periods of unavailability.

Causal logging is complex to evaluate from a performability perspective. The performance overhead in fault-free execution is dependent on the amount of piggybacked information in each message, and this information, in turn depends on the communication pattern and amount between processes. During recovery, causal logging acts similarly to the optimistic

logging protocol. Therefore it may imply in larger recovery time, and resulting unavailable time, than the pessimistic logging.

The data replication techniques may be used in conjunction with rollback-recovery protocols to improve system's performability by increasing system's availability. For example, the use of the primary site approach in storing checkpoints and logs decrease the probability of losing redundant data due failures in the storage device. The major concern in this case is the performance overhead of performing the replication over the backup nodes. Indeed, some contributions of this work rely on this approach to improve the system availability taking into consideration the performance issues. The voting approach is less suitable with few backup nodes, i.e., if using two or three backup nodes, the performance and behavior of this approach is similar to the active replication. However, the active replication may incur in higher overhead due synchronization constraints and the need for a two-phase lock in all operations.

Chapter 3
Performability in the RADIC Architecture

This chapter discusses the characteristics and behavior of the architecture chosen as basis for this work and how they influence in the system performability. In his work, Duarte (DUARTE, A., 2007) introduces a new fault tolerance architecture called RADIC, an acronym for Redundant Array of Independent Fault Tolerance Controllers. Evaluating the performance and availability of a fault tolerance system is complex and challenging. Indeed, different measures of performability are more suitable for different kind of applications while different characteristics of a system may have more or less significance. In this work, evaluation is made not only from outside, but inside too, analyzing the RADIC factors that define and influence system's performability and their influence. Later, this analysis will be the basis of a proposal on alternative designs to improve the performability.

3.1. RADIC architecture model

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

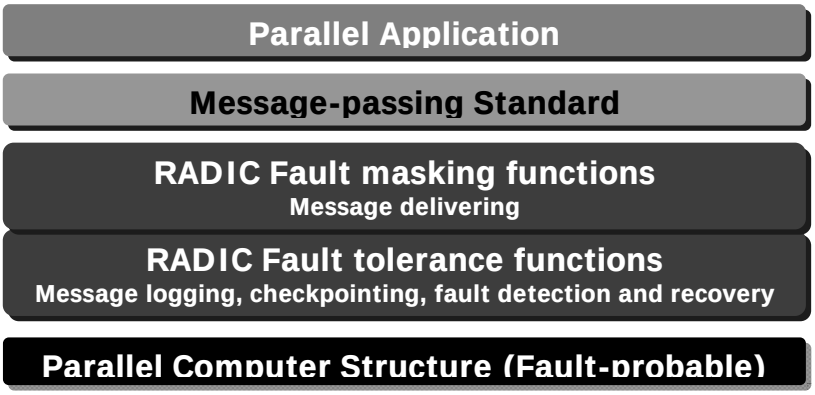


Figure 3-1: The RADIC layers in a parallel system

The core of the RADIC architecture is a fully distributed controller for fault tolerance that automatically handles faults in the cluster structure. Such a controller shares the parallel computers resources. The controller is also capable to handle its structure in order to survive to failures. The RADIC architecture is characterized by four key features as showed in TABLE 3-1. Some of these features directly or indirectly influence performability. The transparency reduces the possibility of fine tuning the application, which allows an optimized use of fault tolerance, e.g., taking checkpoints only when necessary. Decentralization reduces the storage overhead caused by central storage and distributes the redundant data saving activity among the nodes, but does suffer difficulties taking global decisions to improve performability such as load balancing or checkpointing protocol. Flexibility allows the adjustment of values or configuration to help improve performability.

TABLE 3-1: The key features of RADIC

Feature	How it is achieved
Transparency	– No change in the application code – No administrator intervention is required to manage the failure
Decentralization	– No central or fully dedicated resource is required. All nodes may be simultaneously used for computation and protection
Scalability	– The RADIC operation is unaffected by the number of nodes in the parallel computer
Flexibility	– Fault tolerance parameters may be adjusted according to application requirements – The fault-tolerant architecture may change for better adaptation to the parallel computer structure and the fault pattern

3.1.1. Fault model

There are several reasons why a computer cluster may fail (TREASTER, M., 2005). Therefore, a fault tolerance solution must define the scope of the faults it can handle. Such a definition is made in a higher level of abstraction called fault model (JALOTE, P., 1994). The fault model directly affects how availability is perceived and measured, because it excludes some kind of failures and the considered performability is also dependent on it. The RADIC assumed fault model, which defines the faults being considered is described below.

RADIC assumes that the message-passing system follows a fail-stop model. In this model, any node can fail at any time, resulting in a crash or halting the processes running on it. Such a model is commonly assumed in fault tolerance techniques (TREASTER, M., 2005) and allows for failures to be accurately detected (BIRMAN, K.P., 2005) through a timeout procedure. In RADIC, this is represented by the heartbeat/watchdog mechanism presented in section 0. The configuration of this procedure may affect the fault detection time and, consequently, the time to recovery and system availability. Therefore, RADIC detects a node has failed by the absence of an expected communication (the heartbeat or crash of an ongoing message transmission).

RADIC is technically designed to tolerate only permanent faults. However a RADIC implementation may also deal with transient network faults by retrying the communication a determined number of times.

RADIC may tolerate an undetermined number of faults until the cluster reaches the minimal configuration required (explained in section 3.5.2). In order to tolerate concurrent correlated faults, e.g., a node and its neighbor at same time, RADIC demands extra copies of redundant data.

RADIC relies on the underlying transport stack, and assumes that all communications are delivered correctly. Catastrophic faults, such as complete power supply failures, switches or link failures and fire are not covered by the RADIC fault model. Similarly, RADIC does not cover application internal faults (flawed software) or byzantine faults (data corruption, malicious behavior, or incorrect protocols).

3.2. RADIC functional elements

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

3.2.1. Protectors

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

The relationship between neighbor protectors exists because of the fault detection procedure. There is a heartbeat/watchdog mechanism between neighbor protectors. The protected node has to send a periodic life-sign called heartbeat to the protector running a watchdog. If this life-sign fails to arrive at the Watchdog within a certain period, the watchdog assumes a node failure and moves the controlled system into a fail-treatment state. By definition, the protector with the watchdog is the predecessor and the protector who sending the

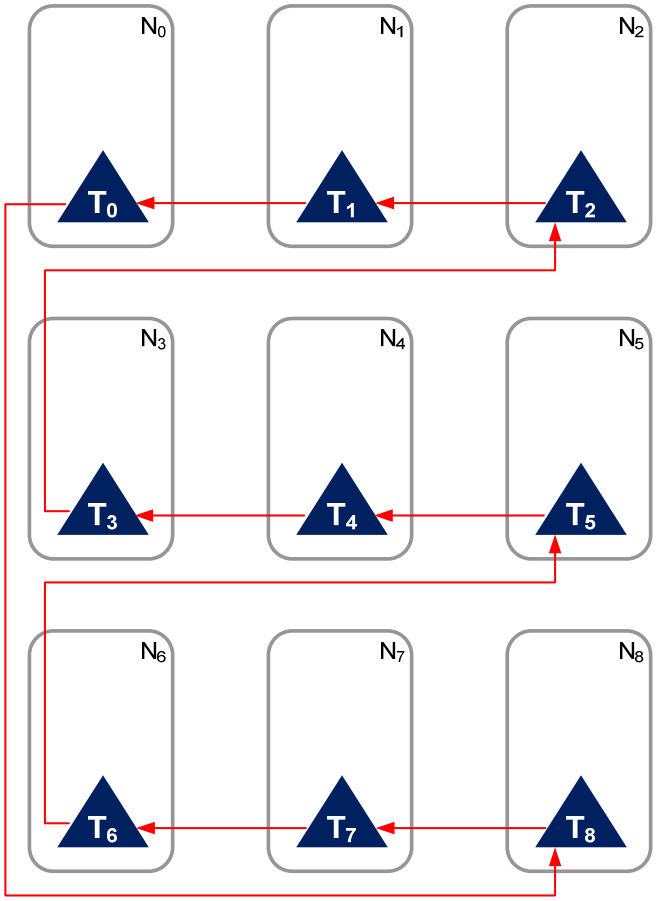


Figure 3-2: An example of Protectors (T_0-T_8) in a cluster with nine nodes. Green arrows indicate the predecessor←successor communication.

heartbeat is the successor. Thus, the protector running in the successor node protects the predecessor node.

The arrows in Figure 3-2 indicate the orientation of the heartbeat signals from the successor to the predecessor. Actually, each protector has a double identity because it acts simultaneously as a successor for a neighbor and as a predecessor for the other neighbor. For example, in Figure 3-2, the protector T_7 is the predecessor of the protector T_8 and the successor of the protector T_6. A protector only communicates with their immediate neighbors. In the same figure, the protector T_5 communicates only with T_4 and T_6. It never communicates with T_3, unless T_4 fails and T_3 becomes its new immediate neighbor.

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

- It stores checkpoints and message-logs from the application processes those are running in other node;
- It monitors its neighbors in order to detect node failures via a heartbeat/watchdog scheme;
- It reestablishes the monitoring mechanism with the following neighbor after a failure in one of its current neighbors, i.e., it reestablishes the protection chain; and
- It implements the recovery mechanism.

Those tasks are related to different phases of the RADIC operation, as described in TABLE 3-2

3.2.2. Observers

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

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

- It takes checkpoints and event logs of its application process and sends them to a protector running in another node;

TABLE 3-2: Phases of RADIC operation performed by protectors

Phase	Functionalities
Protection	To store checkpoints and event log
Detection	To perform the heartbeat/watchdog scheme
Recovery	To re-spawn processes
Reconfiguration	To re-establish the heartbeat/watchdog scheme

TABLE 3-3: Phases of RADIC operation performed by observers

Phase	Functionalities
Protection	To find a protector To take and send checkpoints To log and send events To mask a fault by searching for the faulty process location
Detection	To detect communication failures
Recovery	To process event logs
Reconfiguration	To update the radictable

- It detects communication failures when communicating with other processes or with its protector;
- In the recovering phase, it manages the messages from the message log of its application process and establishes a new protector; and
- It maintains a mapping table, called a *radictable*, indicating the location of all application processes and their respective protectors and updates this table in order to mask faults.

Similar to the protectors, those tasks are related to different phases of the RADIC operation, as described in TABLE 3-3

3.2.3. The RADIC controller for fault tolerance

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

Each observer has an arrow connecting it to a protector running in other node, to which it sends checkpoints and message logs of its application process. This protector is the predecessor of the local protector. Therefore, by asking to the local protector who is the predecessor protector, an observer can always know who its protector is. Each protector main-

tains a list of the observers it is protecting, and the observers running locally on its node. This list is called the observers' list.

The RADIC controller uses the receiver-based pessimistic log rollback-recovery protocol to handle the faults and satisfy the scalability requirement. As explained in the previous chapter, this protocol is the only one in which the recovery mechanism does not demand synchronization between the in-recovering process and the processes unaffected by the fault.

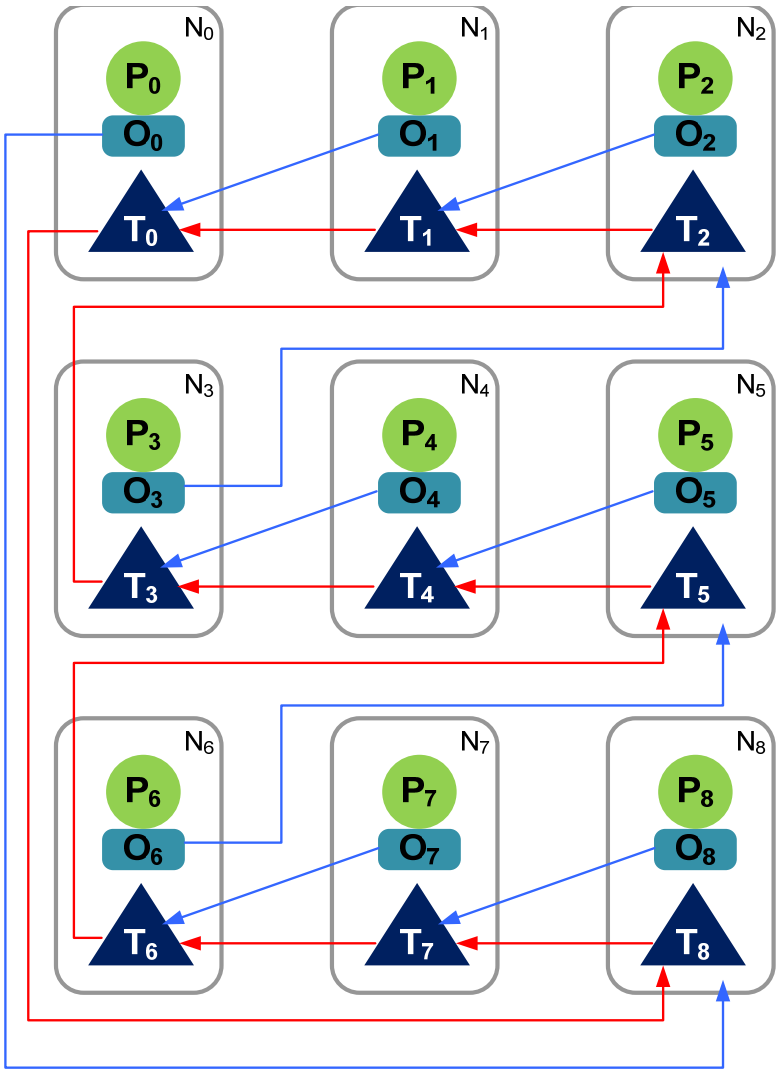


Figure 3-3: A cluster using the RADIC architecture. P_0-P_8 are application process. O_0-O_8 are observers and T_0-T_8 are protectors. O→T arrows represent the relationship between observers and protector and T→T arrows the relationship between protectors.

Such a feature avoids the scalability suffers with the operation of the fault tolerance mechanism.

Besides the fault tolerance activities, observers are responsible for managing the message-passing mechanism. This activity rests on a mapping table containing all information required to correctly deliver a message between two processes. Protectors do not participate directly in the message-passing mechanism; they only store the message log.

3.3. RADIC operation

As shown, the RADIC distributed controller concurrently executes a set of activities related to the fault tolerance. Besides these fault tolerance activities, the controller also implements the message-passing mechanism for the application processes. How these mechanism and tasks contribute to the RADIC operation is explained below.

3.3.1. Message-passing mechanism

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

To obtain the address of a destination process, an observer uses its routing table (the *radictable*). This identifies the destination process inside the application level by the identifying the destination process inside the communication level TABLE 3-4 represents a typical *radictable*.

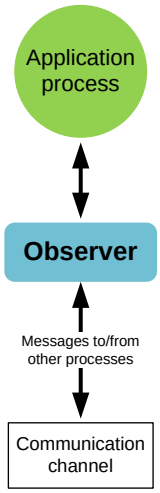


Figure 3-4: The message-passing mechanism in RADIC.

3.3.2. State saving task

In this task, protectors and observers collaborate to save snapshots of the parallel application's state. This task is responsible for the majority of network and storage resources consumption by the fault tolerance mechanism, as well as the performance overhead in fault-free executions.

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

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

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

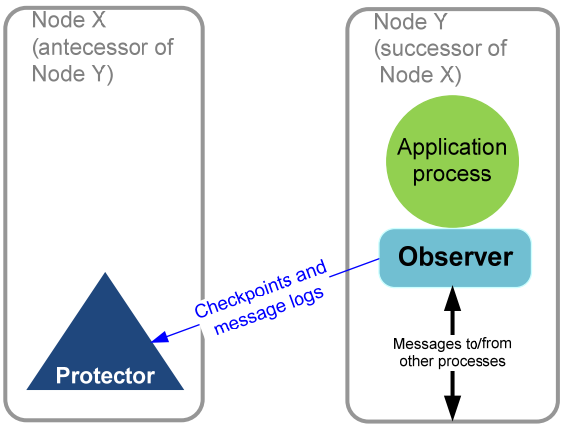


Figure 3-5: Relationship between an observer and its protector.

Additionally, message logging interferes in the message delivery latency, because under the pessimistic logging approach, a process only considers a message delivered after the message is stored in the message log. Furthermore, as more data (from the message logs) are transiting on the network, consuming more network bandwidth, the occurrence of packet collision on the physical medium is more likely, resulting in larger transmission times.

3.3.2.1. Checkpoints

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

According to an implementation, checkpointing may be an atomic procedure and a process becomes unavailable to communicate while a checkpoint procedure is in progress. This behavior demands that the fault detection mechanism differentiates a communication failure caused by a real failure from one caused by a checkpoint procedure. This differentiation is explained in section 3.3.3.

Protectors operate like a distributed reliable storage. Reliability is achieved by the checkpoints and message logs of a process being stored in a different node. Therefore, if a

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

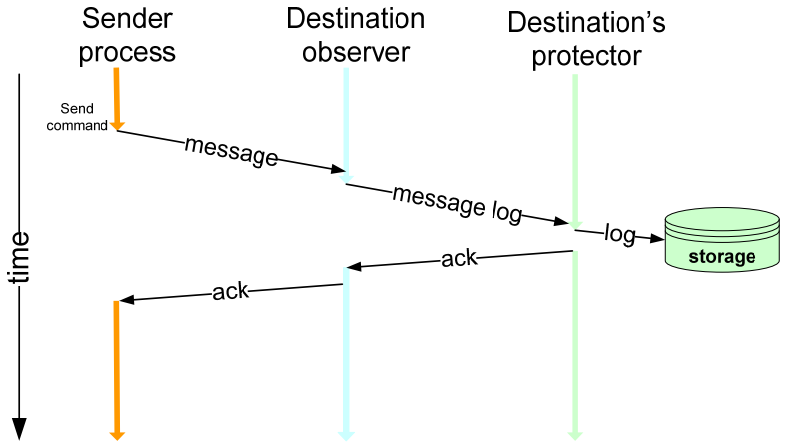
Thanks to the uncoordinated checkpoint mechanism of the pessimistic message-logging rollback-recovery protocol used by RADIC, each observer may establish an individual checkpoint policy for its application process. Such a policy may be time-driven or event-driven. The RADIC architecture allows the implementation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

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

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

Other approach is to provide an adaptive system, using both the time-driven and event-driven policies. For example, the time-driven policy is the default, however, if the storage space for logs is exhausted, this event may trigger the checkpointing procedure to perform the garbage collection.

When an observer takes a checkpoint of its process, this checkpoint represents all computational work undertaken by the process until that moment. The observer sends this computational work to the protector. As the process continues its work, the state saved in the protector becomes obsolete. To make possible the reconstruction of the process' state in case



of failure, the observer also logs in to its protector all messages its process has received since its last checkpoint. Therefore, the protector always has all information required to recover a process in case of a failure. The information is, though, always older than the current process' state.

3.3.2.2. Message logs

Each observer must log all messages received by its application process because the pessimistic log-based rollback-recovery protocol. As explained in the previous chapter, using message logs together with checkpointing improves the fault tolerance mechanism by avoiding the domino effect and reducing the amount of checkpoints the system must maintain.

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

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

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

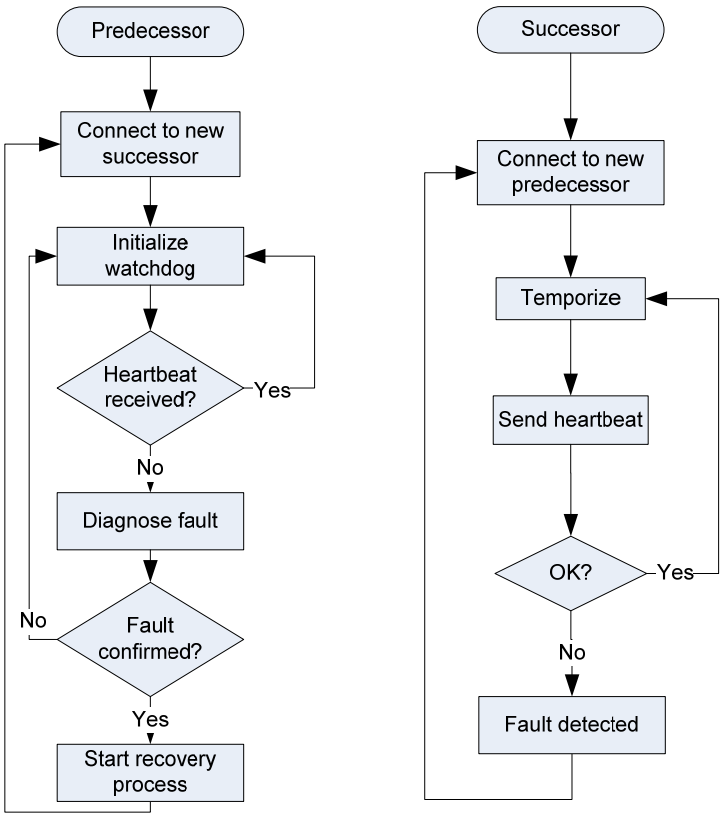


Figure 3-7: Protector algorithms for predecessor and successor tasks

3.3.2.3. Garbage collection: fault tolerance and resources

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

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

3.3.3. Failure detection task

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.

3.3.3.1. How protectors detect failures

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

There is a heartbeat/watchdog mechanism between two neighbors. The predecessor is the watchdog element and the successor the heartbeat element. Figure 3-7 represents the operational flow of each protector element.

A successor regularly sends heartbeats to a predecessor. The heartbeat/watchdog cycle determines how quickly a protector will detect a failure in its neighbor, i.e., the response time of the failure detection scheme. Short cycles reduce the response time, improving the system's MTTR, but also increase the interference over the communication channel. Figure 3-8 depicts three protectors and the heartbeat/watchdog mechanism between them. This picture shows the predecessors running the watchdog routine and waiting for a heartbeat sent by its neighbor.

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

A predecessor, in turn, waits for a new successor after to detect a failure in its current successor. Furthermore, the predecessor also starts the recovering procedure, in order to recover the faulty processes that were running in the successor node.

3.3.3.2. How the observers detect failures

Each observer relates to two classes of remote elements: its protector and the other application processes. An observer detects failures either when communication with other application processes fails or when the communication with its protector fails. However, considering an observer only communicates with its protector when performs a checkpoint or message-log, an additional mechanism must exist to guarantee an observer will quickly perceive that its protector has failed.

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

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

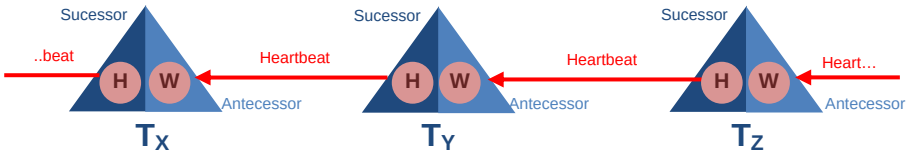


Figure 3-8: Three protectors (T_X, T_Y and T_Z) and their relationship for detecting failures. Successors send heartbeats to predecessors.

3.3.3.3. How the observers confirm a failure

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

This section explains how observers tackle the first problem. How the observer tackle the second situation will be explained in the description of the Fault Masking Phase.

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

In order to avoid this fake failure detection, before a sender observer assumes a communication failure with a destination process, the sender observer contacts the destination's protector and asks about the destination's status. To allow that each observer knows the location of the protector of the other process, the *radictable* now includes the address of the destination's protector, as shown in TABLE 3-5.

TABLE 3-5, shows that the protector in Node 8 protects the processes in Node 0, the

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

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

protector in Node 0 protects processes in Node 1 and so forth.

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

The radictable and the search algorithm

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

To overcome this problem, each observer uses a search algorithm to calculate the address of failed component. This algorithm relies on the determinism of the protection scheme. Each observer knows that the protector of a failed component is the predecessor of this component. Since a predecessor is always the previous element in the *radictable*, whenever the observer needs to find an observer or its protector it simply looks at the previous line in its *radictable*, and finds the address of the respective node. The observer repeats this procedure until it finds the process or protector it is looking for.

3.3.3.4. Recovery task

As previously explained, in normal operation protectors monitor computer's nodes and observers take checkpoints and message logs of the distributed application processes. Together, protectors and observers function as a distributed controller for fault tolerance.

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

3.3.3.5. *Reestablishing the RADIC structure after failures*

The protectors and observers implicated in the failure will take simultaneous atomic

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

actions to reestablish the integrity of the RADIC controller's structure. TABLE 3-6 enlists the atomic activities of each element.

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

3.3.3.1. Recovering failed application processes

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

3.3.3.2. Recovery side-effect

After recovering, the recovered process runs in the same node as its former protector. This means the computational load increases in such a node, because it now contains its original application processes plus the recovered processes. Therefore, the original load balancing

TABLE 3-6: Recovery activities performed by each element implicated in a failure.

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

of the system changes.

Performance degradation

The aforementioned system configuration change may lead to a graceful performance degradation according to the running application's characteristics. This occurs because two or more process share the computing power of a node. Moreover, after recovering, the memory usage in the node hosting the recovered process raises leading to disk swap in some cases. Such an occurrence is one of the major issues relating to performability decreases in such systems.

RADIC makes possible the implementation of several strategies to overcome the load balance problem after process recovery. One possible strategy is to implement a heuristic for load balance that could search for a node with lesser computational load. Therefore, instead of recovering the faulty process in its own node, a protector could send the checkpoint and log of the faulty processes to be recovered by a protector in a node with less computational load. Such a strategy will clearly increase the system's MTTR because of the searching for and transferring redundant data, affecting its availability. Despite the performance benefits obtained in remaining execution may justifying such efforts, there is no guarantee that a node with less computational load will be quickly found. In a situation where all nodes have the same load, this procedure can take much time.

3.3.4. Fault masking task

Fault masking is an observers' attribution. Observers ensure the processes continue to correctly communicate through the message-passing mechanism, i.e., the observers create a virtual machine in which failures do not affect the message-passing mechanism.

TABLE 3-7: The *radictable* of an observer in the cluster in Figure 3-3.

Process identification	Address	Protector (predecessor address)	Logical clock for sent messages	Logical clock for received messages
0	Node 0	Node 8	2	3
1	Node 1	Node 0	0	0
2	Node 2	Node 1	0	0
3	Node 3	Node 2	1	1
...	...	...	...	...

To perform this task each observer manages all messages sent and received by its process. An observer maintains, in its private *radictable*, the address of all logical processes or the parallel application associated with their respective protectors. Using the information in its *radictable*, each observer uses the search algorithm (see section 0) to locate the recovered processes.

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

TABLE 3-7 represents a typical *radictable* including the logical clocks. It shows that the observer owning this particular table has received three messages from the Process 0 and has sent two messages to this process. Similarly, the process has received one message and sent one message to Process 3.

3.3.4.1. Locating recovered processes

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

Two situations creating fake fault detection were described in the Fault Detection Phase section. The first situation occurs when an observer takes a checkpoint of its application process, making this process unavailable to communicate. The solution for this problem was described in the Fault Detection Phase. However, the description of the second situation and its solution follows below.

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

Figure 3-10 describes the algorithms used by an observer acting as sender or receiver. An observer uses the search algorithm only if the communication fails when it is sending a message to another process. If the failure occurs while the process is receiving a message, the

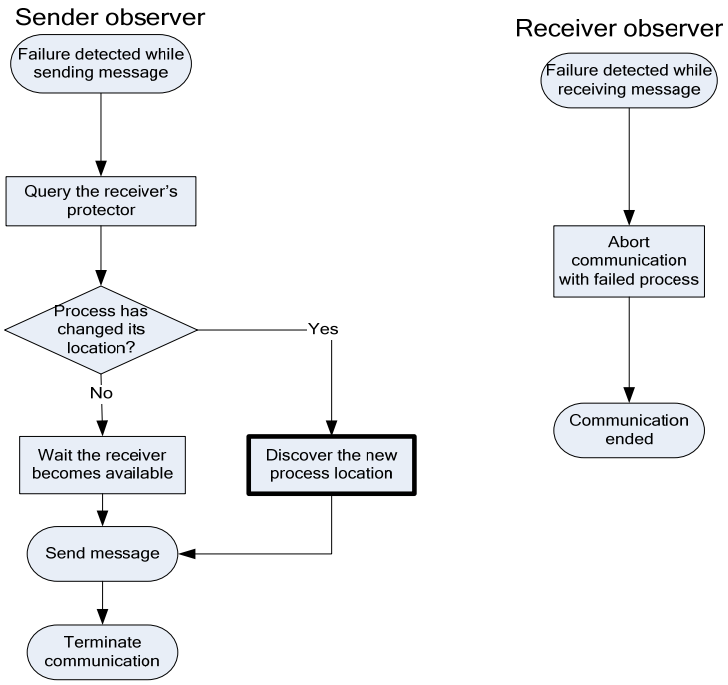


Figure 3-10: Fault detection algorithms for sender and receiver observers

TABLE 3-8: Part of the original *radictable* for the processes represented in Figure 3-9a.

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

observer simply aborts the communication because it knows that the faulty sender will restart the communication after it has recovered.

To clarify the behavior of a recovered process, Figure 3-9d represents the final configuration after a failure in one of these nodes. The process P_3 that was originally in the faulty node N_3 is now running in the node N_2. Therefore, all other processes have to discover the new location of P_3.

The new protector of P_3 is T_1, because P_3 currently runs in the same node as its original protector T_2. If an observer tries to communicate with the faulty process P_3, it will obtain a communication error and ask protector T_2 about the status of P_3. In this case, T_2 informs that it is not responsible for P_3 (because T_1 is now its current protector).

To identify the current protector of P_3, the sender observer uses its *radictable* to follow the protector chain. The sender observer knows that if T_2 is no longer protecting P_3, then the probable protector of P_3 is the predecessor of T_2 in the protector chain (because a faulty process normally recover in the predecessor neighbor node).

Therefore, the sender observer reads its *radictable* and works out the identity of the protector of the predecessor of protector T_2. In the previous example, the predecessor of protector T_2 is T_1. In the *radictable* the order of the protectors in the chain follows the same order as the table index. Therefore, the predecessor of a node is always the node in the previous line of the table, as shown in TABLE 3-8.

Now that the sender observer knows who the probable protector of the receiver process P_3 is, it makes contact and asks about the status of P_3. If the protector confirms the location of P_3, the sender observer updates its *radictable* and restarts the communication process. Otherwise, the sender observer continues to follow the protection chain, asking each following predecessor about P_3 until it finds where the process P_3 is.

In the previous example, the updated *radictable* of a process who tries to communicate with the recovered process P_3 has the information presented in TABLE 3-9. In this table, line three of the *radictable* (represented by bold font) represents the update location of process P_3 together with its new protector. As RADIC is completely distributed, other processes such as P_4 for example, remain unaware of the fault. When they try to communicate with the recovered process, they will search and discover the P_3 location, and updates their *radictable* as needed. No information has to be broadcasted to all processes.

This process is based on the determinism of RADIC when recovering, which guarantees that the recovered process will be in a node known by its protector. This heuristic will be changed when dynamic redundancy will be incorporated, because the spare node use may generate an indeterminism when locating a failed process, once such process may recovers in any spare node available.

TABLE 3-9: Part of the updated *radictable* of a process that has tried to communicate with P3 after it was recovered as shown in Figure 3-9b.

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

3.3.4.2. Managing messages of recovered processes

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

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

Observers discard repeated messages received from recovered processes. However, a recovered process starts in a different node to before the failure. Therefore, it is necessary to make the observers capable of discovering the recovered processes' locations.

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

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

3.4. RADIC functional parameters

The RADIC controller initially establishes two time parameters: the checkpoint interval and the watchdog/heartbeat cycle.

Choosing the optimal checkpoint interval is a difficult task. The interaction between the application and the checkpoints determines the enlargement of the application execution time. Using the interaction between the observers and parallel application processes, the RADIC controller allows the implementation of any checkpoint interval policy. Each observer can calculate the optimal checkpoint interval by using a heuristic based on local or distributed information. Furthermore, the observer may adjust the checkpoint interval during process execution. There is a history of studies proposing strategies for choosing checkpoint interval (DALY, J. T., 2006), (NAM, H. et al., 1997) , (YOUNG, J. W., 1974). In this work, the checkpoint interval is determined only for studying its impact in the performance overhead.

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

Setting the RADIC parameters to achieve the best performance of the fault tolerance scheme is strongly dependent on the application's behavior. The application's computation-to-communication pattern plays a significant role in the interference of the fault-tolerant ar-

chitecture on the parallel application's run time. For example, the amount and size of the messages directly define the interference from message-log protocols.

3.5. RADIC flexibility

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

The freedom to adjust fault tolerance parameters for each application process individually is one functional feature that contributes to the flexibility of the RADIC architecture. Additionally, two features play an important role for the flexibility of RADIC: the ability to define degrees of availability and the structural flexibility.

3.5.1. Concurrent failures degrees of availability

In RADIC, a recovery procedure is complete after the recovered process establishes a new protector, i.e., only after the recovered process has a new protector capable of recovering it. In other words, the recovery procedure is finished when the following steps have been completed:

1. The predecessor protector detects the node failure;
2. It confirms (diagnostically) the failure with the faulty node successor;
3. It re-establishes the heartbeat/watchdog;
4. It re-spawns the process using the stored checkpoint;

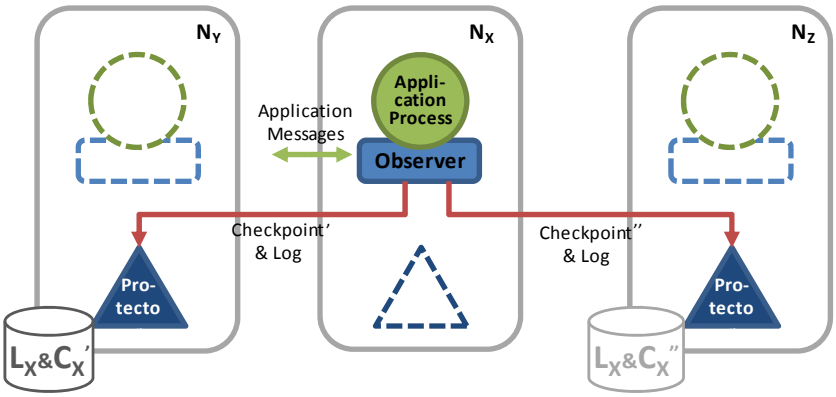


Figure 3-11: An observer using two protectors.

5. The recovered observer establishes a new protector by communicating with the local protector;
6. The recovered observer processes the log conjunctly with the application: and
7. Finally, the observer takes a checkpoint in the new protector in to store the state after log processing.

RADIC optimistically assumes that the protector recovering a failed process will never fail before recovery completion, i.e., using a simple configuration the RADIC controller supports several simultaneous non-correlated faults. Nevertheless, the RADIC architecture allows the construction of an *N-protector* scheme to manage such a situation.

In such a scheme, each observer transmits the process's checkpoints and message logs to *N* different protectors. In Figure 3-11, an observer is using two protectors, sending the checkpoint and log ($L_X\&C_X$') to a protector and a copy of them ($L_X\&C_X$'') to another protector. If the protector running on node N_Y fails while it is recovering a failed application process, the protector running on node N_Z would assume the recovering procedure.

In other example, in the cluster of Figure 3-9, if node N_2 fails before the recovery of P_3, the system will collapse. To solve this situation using an *N-protector* scheme, each ob-

server should store the checkpoints and message-logs of its process in two or more protectors. In Figure 3-9, using two protectors would mean that O_3 should store the checkpoints and message-logs of P_3 in T_2 and T_1. Therefore, T_1 will recover P_3 in case of a failure in T_2 while it is recovering process P_3. During the recovery process, some election policy must be applied to decide the protector who will recover the failed process.

The number of correlated concurrent faults the system must support defines the number of protectors needed by each process. Using such a configuration yields two main costs. The first is the replication of the fault tolerance information in the protectors, which reduces the total storage capacity of the cluster. The second is the data redundancy overhead, namely checkpoint storing and the message transmission latency. The latter suffers a significant increase because in the pessimistic message-log protocol, each observer must now log any received message (and each checkpoint) into N protectors, where N is the number of elements that can concurrently fail even if correlated.

The checkpointing overhead may be avoided by applying a round-robin scheme over the protectors. However this approach cannot be used to log messages since a protector responsible for initiating the recovery must have all data needed to perform the process. This would make the recovery process complex and time consuming. Moreover, such an approach also demands coordination during recovery order to determine the most recent checkpoint replica among the protectors.

Such a feature plays a major role in the system's performability by increasing the degree of availability but generating additional performance overhead.

3.5.2. Structural flexibility

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

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

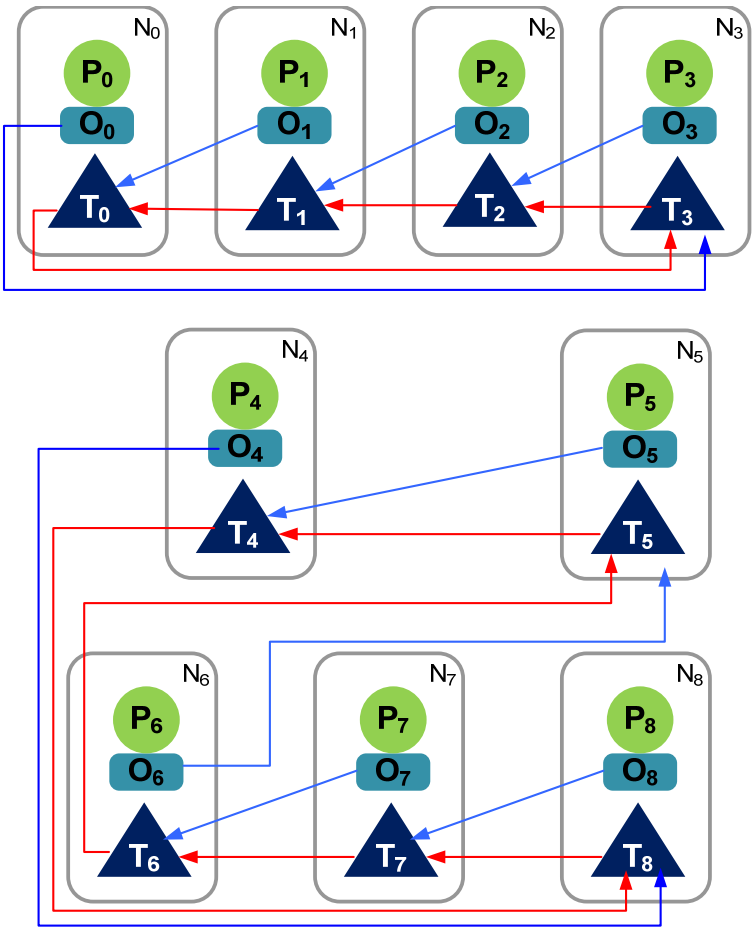


Figure 3-12: A cluster using two protectors' chain.

To implement this feature one column must be added to the *radictable.* This column indicates the protector's chain. An observer uses this additional information to search the protector of a faulty node inside each protectors' chain. The bold column in TABLE 3-10 exemplifies the chain information in a typical *radictable*.

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

$$MaxFaults = Number\_of\_Protectors - 2 \quad (7)$$

TABLE 3-10: The *radictable* of an observer for a cluster protected by two protectors' chains such as in Figure 3-12.

Process identification	Address	Protector (predecessor address)	Chain	Logical clock for sent messages	Logical clock for received messages
0	Node 0	Node 3	0	2	3
1	Node 1	Node 0	0	0	0
2	Node 2	Node 1	0	0	0
3	Node 3	Node 2	0	1	1
4	Node 4	Node 8	1	2	3
5	Node 5	Node 4	1	0	0
6	Node 6	Node 5	1	0	0
7	Node 7	Node 6	1	1	1
8	Node 8	Node 7	1	0	0

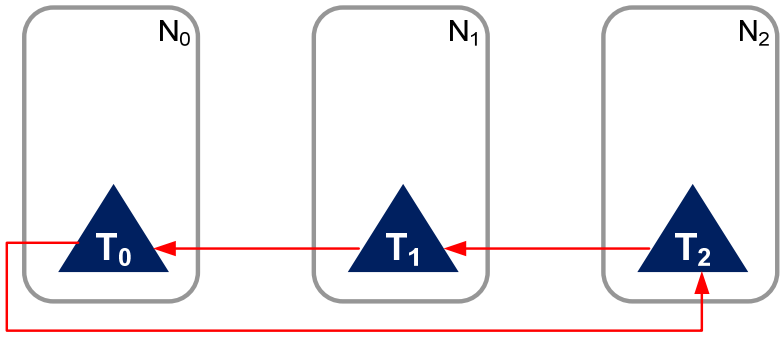


Figure 3-13: The minimum structure for a protectors' chain.

If a fault takes out a node of the chain, and a chain with two nodes is not capable of handling any fault, the minimum number of protectors in a chain defines the maximum number of faults such a chain can handle. Equation (7) expresses this relationship. The maximum number of faults that a protector chain can handle is equal to the number of protectors in the chain minus two (the number of neighbors of a protector).

3.6. The RADIC overhead

Many fault tolerance solutions rely on a centralized stable storage to ensure the survival of redundant data. This requirement can create a bottleneck when saving the state of the overall application (BOUTEILLER, A. et al., 2006), (ELNOZAHY, E. N. and Plank, J. S., 2004), (SANCHO, J. C. et al., 2004) For instance, an application with hundreds of processes, each one with several megabytes or even gigabytes of state size, which stores its checkpoints and logs all interchanged messages in a single point, will require massive bandwidths in network and disk access.

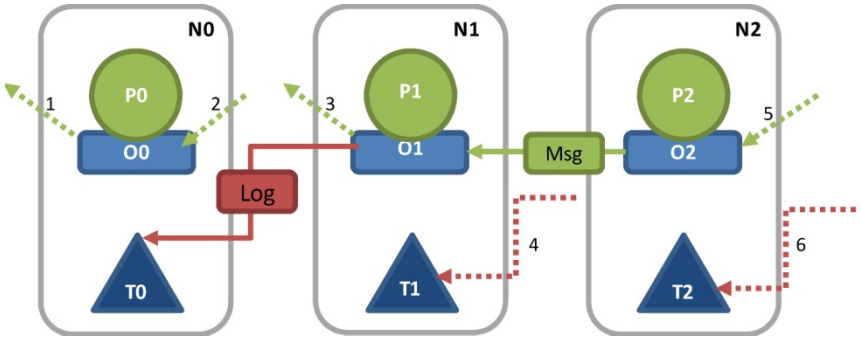


Figure 3-14: Concurrent communications during a message sending.

RADIC is different from common fault tolerance solutions because it implements a distributed stable storage over the protectors. Such a feature avoids the occurrence of bottlenecks when saving checkpoints and logs, despite demanding storage in each node. However, RADIC may generate some disturbances on the application's communication, and this can increase overheads. These disturbances are caused by concurrent communications during the message transmission as depicted in Figure 3-14 where the dotted arrows numbered from 1 to 6 represent communications that may occur while the message is sending and logging (or a checkpoint is being taken). For instance, the protector of receiver node (N1) may be receiving a checkpoint (dotted arrow #4) at same time that process P2 is sending a message.

These concurrent communications and their side-effects are dependent on factors such as the application's communication pattern, network topology and protectors' assignments.

If an application has a synchronized communication pattern (i.e., processes communicate almost at the same time), the probability of concurrent communication is greater. To better understand the influence of such a phenomenon in system's performability because of the performance overhead, evaluation experiments were conducted using synthetic programs generating synchronized and unsynchronized communication. The synthetic programs are based on a SPMD matrix product modified to force or avoid message sending synchronism.

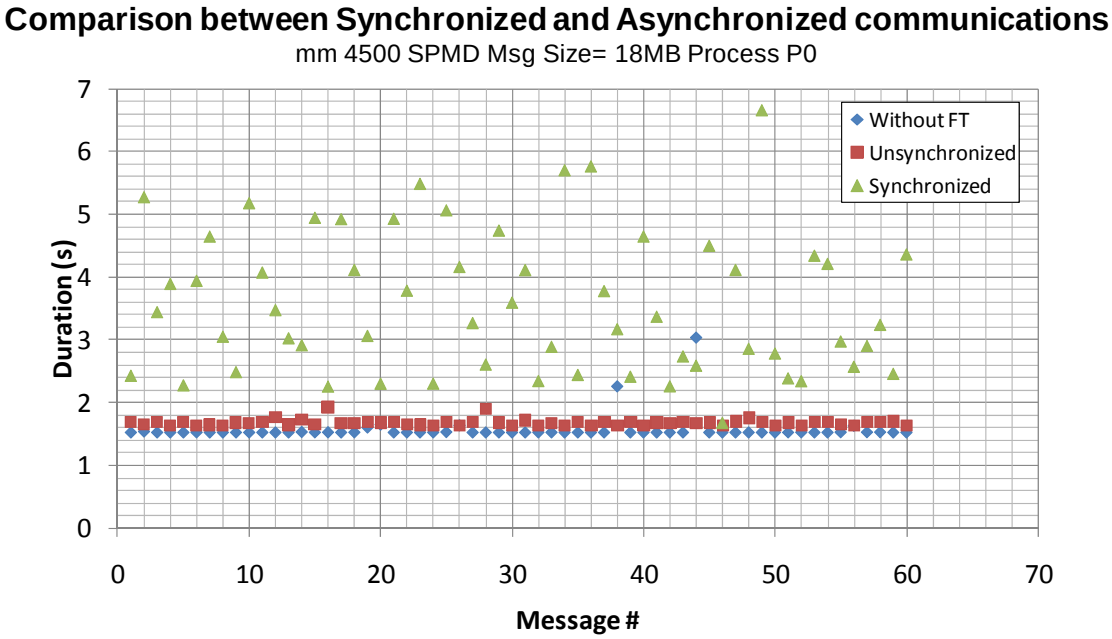


Figure 3-15: Influence of message sending synchronism in the overhead of message logging.

Figure 3-15 charts the result of these executions using a 4500×4500 matrix, generating 18MB messages. The chart shows the message sending durations of the process P0. It is possible to see the strong variation and time overhead when the communications are completely synchronized. However, if there is no synchronism in the communications, the overhead is low and stable.

This evaluation confirms the dependence between the RADIC overhead and the communication pattern of an application. Furthermore, as previously described, other factors may influence in this overhead, such as the network topology and routing techniques used in the computer cluster. If they provide alternate paths to message traffic, probability of these concurrent communications could be mitigated. In some cases, the protector-observer assignment may also reduce this probability by assigning to an observer the protector running in the most communicated node and avoiding disturbing the logging activity of other processes. This subject is still undiscovered and may be the target of future studies.

3.7. Estimating the availability provided by RADIC

To estimate the availability of a system is a task quite complex. The direct approach to achieve this number is using the operational availability, i.e., to measure how much time the system was operational during some period of time and to calculate such ratio. Such an approach has the inconvenient of only be possible after past the time, which may be useless in many cases.

As faults follow no specific model, one reasonable estimation technique, taking into consideration the fault model assumed by RADIC (see section 3.1.1) is inherent availability. This approach is based on probabilistic factors such as average time between failures (MTBF) or average time for recovery (MTTR).

It is clear that the availability provided by RADIC depends on the cluster structure where RADIC is applied, i.e., according to the MTBF of such a cluster since a single fault will cause an interruption. Moreover, such availability also depends of RADIC parameters such as checkpoint and heartbeat interval and the running application characteristics, which leads to the MTTR.

The MTTR remains complex to determine in an uncoordinated pessimistic log-based solution. In contrast to coordinated approaches, it does not obligate all process to roll back to the last checkpoint. Only the faulty process must to be roll backed, which means that the application can continue executing while the process recovers, except if it is a tightly coupled application, i.e., another process waiting for a message from the faulty node. In this case, the entire application (if there are correlated dependencies) may have to wait some time until the process finishes its recovery. Therefore in the best case (loosely coupled applications) the

time to recover would be near zero and in the worst case (tightly coupled applications) the time to recover would be near to the checkpointing interval value.

In cases of coordinated checkpointing, as faults are equally likely within the checkpointing interval and the entire system rollbacks to the last checkpoint, it is reasonable to assume that the MTTR is equal to one half of the checkpointing interval. In RADIC's case, as an uncoordinated approach with a pessimistic logging solution, the MTTR would be equal to that assumed for coordinated protocols at most (in case of tightly coupled applications), but be smaller in many cases considering the entire universe of loosely coupled applications. This work assumed the MTTR was one half of the checkpoint interval (the worst case).

An example of estimating the availability provided by RADIC follows. A cluster without a fault tolerance solution is composed of 100 nodes, each one with a MTBF of 8,760 hours (one year). The SMTBF is 87.6 hours (see Equation (5)). Considering the MTTR is two hours i.e., support staff replaces the faulty node in two hours, the inherent availability of this system calculated using Equation (1) would be 97.76786% (87.6/(87.6+2)), not considering the time spent by the application re-executing until it reaches the state immediately before the fault.

When using RADIC, the MTTR of the system reduces according to the checkpoint and heartbeat interval chosen (because of automatic recovery). To simplify it was considered only the checkpoint interval, supposing that a fault is immediately detected and the reconfiguration takes an unnoticeable amount of time. A checkpoint interval of two minutes (0.034 hours) means that, according to the exposed before, the MTTR is 1 minute (0.0167 hours). In this case, the system availability would rise to 99.9808% (87.6/(87.6+0.0167)). To maintain this availability value in the face of concurrent correlated faults, RADIC must be configured

to use as many protectors per observer as needed, resulting in a performance overhead increase.

As mentioned before, for the purpose of this work the unavailability metric was preferred. Therefore, the estimated unavailability value using RADIC would be 0.01902% and 2.23214% without RADIC. While RADIC provides an increase of 2.21% in system availability, unavailability was reduced by many orders of magnitude, reflecting better the importance of using a fault tolerance solution.

Chapter 4
Alternatives for Improving a Computer Cluster's Performability

The previous chapter explained about how RADIC can protect an application from faults, and provides high availability. It presented operational details about saving state, detecting faults and recovering a process. It also showed that RADIC, as any fault tolerant solution imposes a performance overhead in fault-free executions because of its fault tolerance activities. Furthermore it may degrade performance after faults occur because of the recovery process changing the original system configuration. These issues directly affect the system's performability, since the system performance may be compromised in order to provide some degree of availability.

Time overhead and resources consumption (such as storage space) usually limit the availability provided by rollback-recovery solutions such as RADIC. An approach to increasing availability is to make several replicas of checkpoints and logs. However this will lead to an increase in the fault tolerance overhead. Such an overhead may be an important concern in cases of mission-critical applications.

In fault cases, such a system degrades performance, i.e., the system remains operational but with some loss of performance caused by changes on the processes per node distribution. In this situation, it might be relevant the system re-configuration, restoring the original process distribution and recovering the original performance.

Furthermore, it was incorporated the functionality of fault-probable nodes replacement (because of factors such as the MTBF) before the fault occurrence. This approach goes

beyond fault tolerance, allowing preventive maintenance actions. It is possible to replace each node of a cluster without stopping an application running. Such a feature is relevant for long or continuous running applications and can be used in conjunction with fault prediction strategies. These strategies can improve performability by fault avoidance.

This chapter discusses how these issues affects system's performability and evaluates the root causes of the performance overhead and degradation. It also presents solutions for improving system's performability in the presence or not of faults by reducing the message logging overhead. As explained before, an important factor regarding such an overhead is the sequential approach of the message logging (it receives the message, then it logs it). Facing this issue, a parallelization of such a process, based on the pipelining technique is proposed.

The rest of this chapter is organized as follows: Fault-free issues are discussed and alternatives for reducing message logging overheads and increasing availability with low overheads are presented. Performance degradation caused by faults is then presented with alternatives for avoiding system changes and system restoration. The chapter concludes by discussing how to provide preventive maintenance without stops.

4.1. Fault-free issues

Any fault tolerance solution leads to a cost, such as financial (special or redundant devices are expensive), extra storage space, or time overhead. These costs are usually related to the degree of availability offered by these solutions, i.e., in hardware redundancy as many redundant devices are necessary as many faults must be tolerated. In rollback-recovery based fault tolerance solutions, the costs are usually associated with storage space and, mainly, with time or performance overhead. The following sections explain the proposed solutions for

improving performability in fault-free situations by reducing the performance overhead, and increasing availability without imposing large overheads.

This work considers the fault tolerance overhead as performance degradation, so it is possible to be evaluated under the performability concept. This approach is reasonable if it is taken into consideration that a fault tolerance solution is only necessary because of the likelihood of faults, meaning the overhead caused by such a solution is related to faults.

4.2. Reducing the message logging overhead

As explained before, RADIC, as a log-based fault tolerance solution has two major sources of overhead: checkpointing and logging. The checkpointing overhead and possible solutions have already been the subject of various research proposals including diskless checkpointing (PLANK, J. S. et al., 1998), incremental checkpointing (SANCHO, J. C. et al., 2004), (AGARWAL, S. et al., 2004), checkpoint size reduction by compiler assisted selection of variables (RODRÍGUEZ, G., 2008), and models for optimum checkpoint interval (PLANK, J. S. and Thomason, M. G., 2001), (DALY, J. T., 2006). This work focuses on the overhead caused by message logging.

As presented in (BOSILCA, G. et al., 2002) and (RAO, S. et al., 2000), the main cause of the pessimistic log overhead is the store-and-forward approach of regular implementations, where the logging process starts only after receiving the complete message, as depicted in Figure 4-1. In the first phase the destination observer receives the entire message and, in the second phase, the destination observer logs the entire message in its protector.

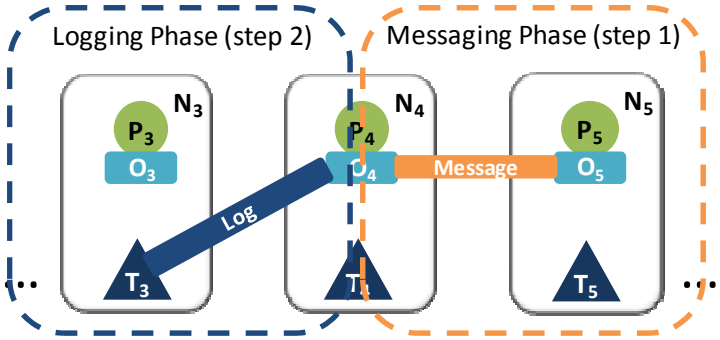


Figure 4-1: Phases of a RADIC receiver-based logging.

A simple mathematical model (NI, L. M. and McKinley, P. K., 1993) commonly applied in analyzing network routing techniques can approximately predict how this approach will perform. Suppose the message size is M, the bandwidth of the network is B, and the number of hops is n (fixed value of 2). The latency values are given by Equation (8). In practice, other factors may influence this equation according to implementation issues and network protocols.

$$Latency_{Store\&Foward} = \left(\frac{M}{B}\right) \times n \qquad (8)$$

It is possible to evaluate the overhead caused by regular logging by comparing the values presented in Figure 4-2. These values result from executing the NetPIPE (SNELL, Q. O. et al., 1996) network performance evaluator over a Gigabit Ethernet network. The TABLE 4-1 shows the numerical overheads of values presented in the previous chart. Overhead s reach a maximum of 89.7%, these value are slightly lower than the value obtained from the previous equation because of the aforementioned reasons[1].

[1] . This difference is because the RADIC prototype implementation, which uses a receiver message buffer that always accepts messages, which allows sending and receiving messages to be overlapped

TABLE 4-1: Numerical logging overhead comparison

	512 B	1 KiB	8 KiB	16 KiB	32 KiB	64 KiB	256 KiB	512 KiB	1MiB	8MiB	16MiB
Overhead	83.4%	85.0%	80.5%	72.9%	71.0%	72.3%	81.4%	84.1%	86.3%	89.3%	89.7%

Applications with a large number or size of communications may be severely affected by the effects of message logging, especially if they have no load balancing. Indeed some authors suggest this factor is the major drawback of log-based fault tolerance solutions (ELNOZAHY, E.N. and Zwaenepoel, W., 1994), (ALVISI, L. and Marzullo, K., 1998), (HUANG, Y. and Wang, Y., 1995). Figure 4-3 compares the execution times of a 9000×9000 matrix product SPMD program when performing or not message logging running with different number of nodes. In these executions checkpointing was not performed in order to solely evaluate the message logging overhead[2]. The increase in overheads according to the number of nodes is because of the inter-process dependencies. With four nodes, each process only

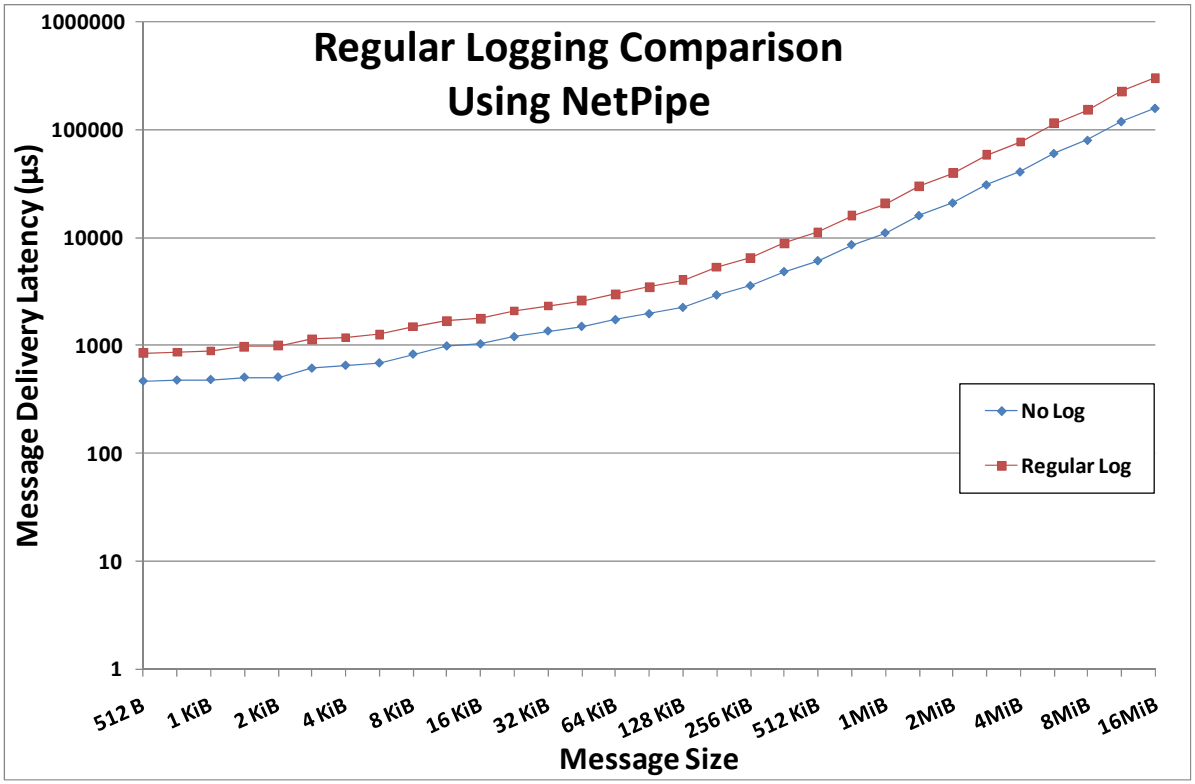


Figure 4-2: Message latency comparison using or not message logging.

[2] In practice, message logging is performed conjunctly with checkpointing to bound recovery time and storage space.

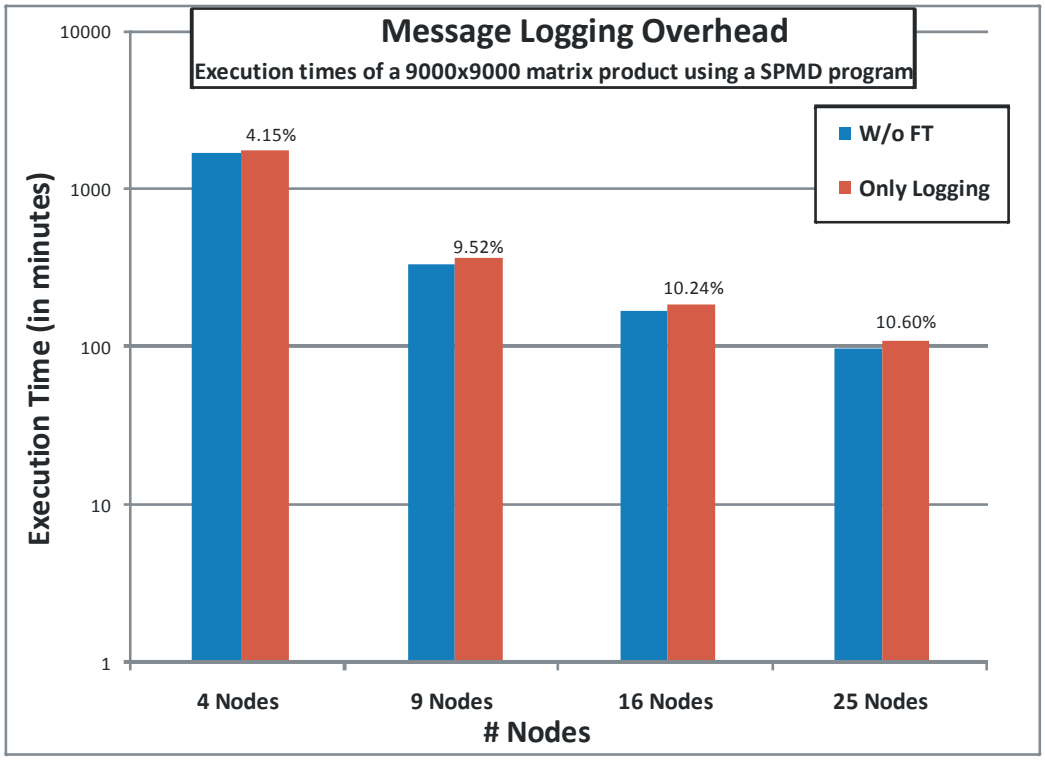


Figure 4-3: Execution time comparison between using or not message logging in a 9000x9000 matrix product over different number of nodes

depends on two other processes in order to advance each iteration. However, with nine nodes, each process depends directly on four other processes and indirectly on another four processes (Figure 4-4), causing a delay as one process is propagated to the others. Furthermore, the overhead is also dependent on network characteristics. In this case, a Gigabit Ethernet network interconnected by one network switch was used.

Such an overhead depends on application characteristics such as communication patterns, process interdependencies, message sizes and the underlying network. In this case, the program is based on the Cannon's algorithm, generating few inter-process point-to-point communications of a large size (an entire matrix block in each communication) and strong dependency (each computation depends on the results of two other processes' computation).

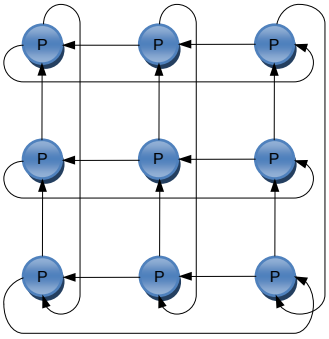


Figure 4-4: Message pattern of a matrix-multiplication using the Cannon's algorithm based on the SPMD paradigm.

As seen, message logging affects a system's performance and, consequently, its performability. In order to improve system performability the following solution allows a decrease in the message logging overhead without compromising the availability provided by the fault tolerance solution. The next section presents the details of this solution.

4.2.1. Pipelining the logging process

In order to improve the logging process, two factors have been taken advantage of: a wide range of actual networks provides full duplex communications (it can send and receive data at same time) and communication buffers at lower network layers exist (for instance TCP or network interface card (NIC) buffers). Therefore, using a technique similar to the wormhole (NI, L. M. and McKinley, P. K., 1993), generally used in network systems like routers or switches, the observer's message-passing mechanism and state-saving tasks were modified to establish a pipeline of the logging process by slicing the sending message into small pieces. The receiver observer then logs these packets as they arrive, before completing the message as depicted in Figure 4-5.

By using a full duplex transmission, the observer can simultaneously receive pieces and log them into its protector. Theoretically, such a communication should not affect message transmission because of the full-duplex network feature. However, in practice, it de-

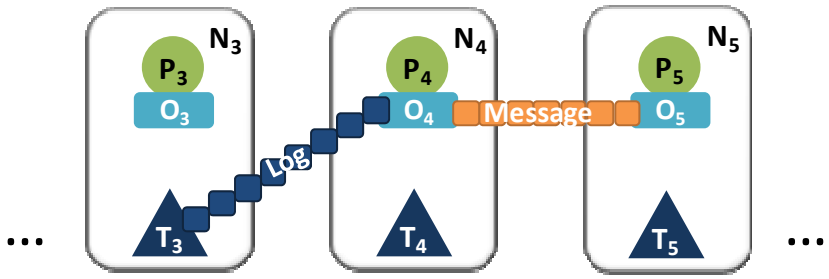


Figure 4-5: The pipelined log process.

pends on factors such as network protocol or topology, i.e. in TCP protocol, the receiver sends small acknowledgement packets to control the transmission.

When using reliable network protocols that ensure delivery, time spent processing the pipeline at each piece may be mitigated by underlying communication buffers. While the pipeline algorithm is processing one piece, more pieces may be received at lower network layers. In the same way, when the pipeline algorithm sends a piece, it only needs to queue it in the sending buffer without needing to wait for the send completion.

Supposing the same network mathematical models used in the section 4.2, using the Equation (9) in this case, usually applied for wormhole routing techniques, where P is the piece size, a pipeline transmission should introduce minimal overheads into the message delivery. Again, implementation issues may affect the expected values.

$$Latency_{Wormhole} = \left(\frac{M}{B}\right) + \left(\frac{P}{B}\right) \times n \qquad (9)$$

Piece size influence

Despite factors influencing the pipelined logging performance, such as the node's computing power, an important consideration is the piece size chosen to slice the message. An inappropriate piece size may generate considerable overhead in the pipeline process.

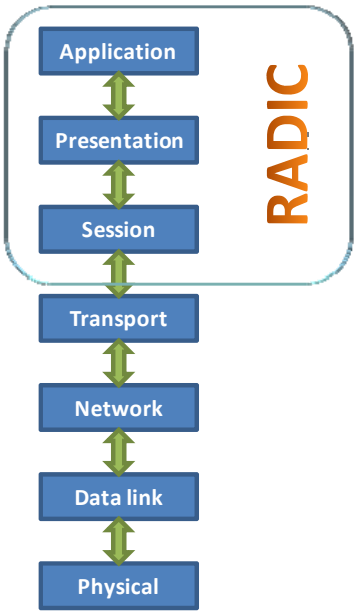


Figure 4-6: RADIC and the OSI layers.

Referring to the Open System Interconnection (OSI) reference model created by the International Organization for Standardization (ISO) that defines a layer-based standard for network protocols, RADIC, as an architecture for message passing, was designed to work over the three upper layers (application, presentation and session) as depicted in Figure 4-6. The lower layer (transport) is responsible for, among other operations, dividing the data from the upper layer into smaller units called TPDUs (Transport Protocol Data Unit). As the data passes by lower layers, the TPDUs are encapsulated into packets at the network layer, and the packets are encapsulated into frames at the data-link layer as depicted in Figure 4-7.

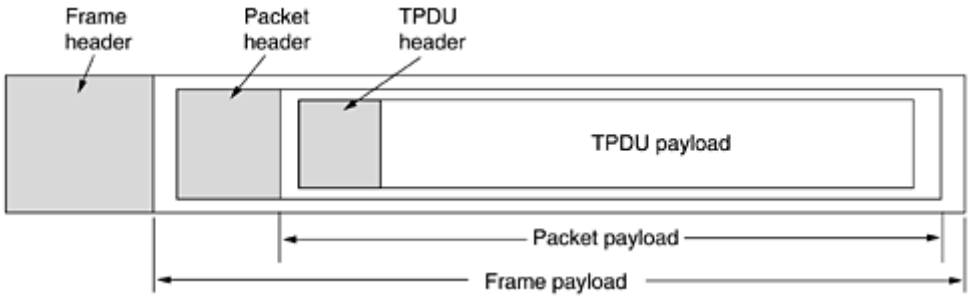


Figure 4-7: The encapsulation of data over the OSI layers.

TABLE 4-2: Default MTU sizes for different networks

Network	Default MTU (bytes)
PPP	296
X.25	576
IEEE 802.3	1,492
Ethernet	1,500
FDDI	4,352
4Mb Token Ring	4,464
Ethernet Jumbo Frames	1500-9,000
16Mb Token Ring	17,914
Hyperchannel	65,535

The frame payload at the network layer limits the TPDU payload and each network has a maximum transmission unit (MTU) limiting the frame size. Therefore, in order to avoid costs due to unnecessary extra information, the piece size used in pipelined logging must be defined according to the MTU of the underlying network (see TABLE 4-2). Indeed, it must fit into a TPDU payload (Equation (10)). Figure 4-8 presents three situations that may occur according to the piece size chosen:

- The piece is oversized, so it will be fragmented and another frame is used to transmit the exceeding data, clearly causing added costs because of the extra headers;
- In the case of an undersized piece, there will be unused space in each frame transmitted, as each frame carries all headers. This case will also generate additional overhead; or
- The desired situation, the piece plus the headers fits the payload of the MTU.

$$TPDU_{\text{payload}} = MTU - Size_{Frame\ Header} - Size_{Packet\ Header} - Size_{TPDU\ Header} \quad (10)$$

In order to keep the RADIC transparent, the pipelined logging implementation must be able to discover the underlying network MTU and adjust the piece size according with this payload.

Faults during the logging process

Any fault tolerant solution must take into consideration that faults can occur at any moment, including when performing the fault tolerance activities. Therefore, it is possible for a fault to occur during the pipelined logging process. If such a fault in the message sender or receiver occurs, the original RADIC recovery process is applied to bring the system back to a consistent state as described in previous chapter, and discard the ongoing log. However, if the fault occurs in the node running the protector receiving the pipelined log, the procedure is slightly different from the original one. In this case, the observer continues to receive the message and buffers it while a new protector is designed according the original RADIC recovery process. After the protector is established, the observer sends the entire buffered mes-

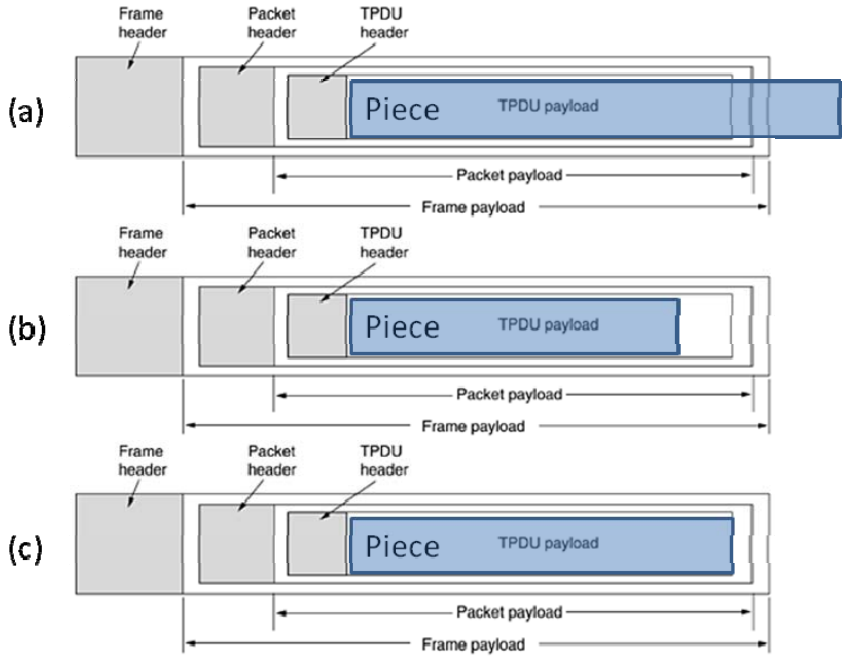


Figure 4-8: Three situations according the piece size: (a) Oversized, (b) Undersized and (c) Right-sized

sage pieces at once. If the message is still incomplete, the observer continues using the pipelined logging.

4.3. Protecting mission-critical processes

One application requiring total or degraded controlled fault tolerance is the mission-critical application. Mission-critical applications are ruled by time constraints (deadlines). They must perform a defined task before this deadline otherwise the task result is useless and must be discarded, representing a waste of time and computational resources. These missed deadlines are commonly caused by faults during the task execution time, as seen in Figure 4-9. The fault occurrence leads to a task restart and total re-execution, and consequently the deadline is missed. Risk is an important role for these applications. In this context risk is considered a function of the fault probability and the damage caused by such a fault. In mission-critical applications, the damage of a fault may be catastrophic such as a missed deadline. Therefore the use of a fault tolerance solution is indispensable.

A fault tolerance solution can certainly increase system availability to a certain level. However, this level is generally limited by factors such as resources, time overhead and cost.

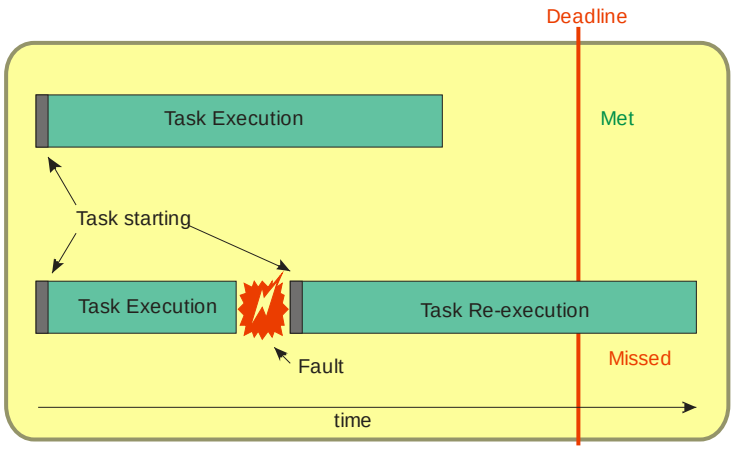


Figure 4-9: A mission-critical task missing a deadline due the fault occurrence

For instance, in the hardware redundancy approach, the availability level is derived from the number of redundant devices in use. That means as many concurrent faults must be tolerated, many redundant devices are needed. The factor limiting the availability in this case is the cost of replication. As explained before, the availability provided by rollback-recovery-based solution such as RADIC is limited by time overhead or by resources consumption (e.g., storage space).

In order to increase the availability provided by a rollback-recovery-based fault tolerance solution, common non-excluding approaches are to reduce the checkpoint interval or make several replicas of the checkpoints and logs (also called redundant data). Both approaches imply an increase of the fault tolerance overhead. The first decreases unavailability periods during the recovery process, which is especially significant in the coordinated checkpoint approach, because in this protocol, all processes must rollback to the last checkpoint, as opposed to uncoordinated protocols where only the faulty process must rollback. The second approach tolerates concurrent correlated faults, a situation when two or more faults occur concurrently and affects both the application's computing node and the redundant data repository.

The concurrent correlated faults are theoretically less probable, however, some studies (LIANG, Y. et al., 2005), (SAHOO, R. K. et al., 2003) have demonstrated that in real systems, faults are temporally and spatially correlated, which may increase the concurrent fault probability. Furthermore, depending how and where the stable storage is implemented, it may also increase the likelihood of these faults correlating to each other. The risk existent in this situation may be unacceptable for mission-critical applications.

In order to deal with these concurrent faults, the RADIC architecture is configured to use more than one protector. The number of correlated concurrent faults the system must support defines the number of protectors needed by each process. Using such a configuration yields two main costs. The first is the replication of the fault tolerance information in the protectors, which reduces the total storage capacity of the cluster. The second is the data redundancy overhead, namely checkpoint transmission and storing and the message transmission latency (because of the log replication). As deduced by Equation (11) the latter suffers a significant increase because in the pessimistic message-log protocol, each observer now must log any received message (and each checkpoint) into N protectors, where N is the number of elements that can concurrently fail even if correlated. The checkpointing overhead could be avoided applying a round-robin scheme over the protectors. However, that procedure makes the recovery process more complex and time consuming. Moreover, it also demands coordination during the recovery in order to determine the most recent checkpoint replica among the protectors. However this approach cannot be used for message logging since a protector responsible for initiate the recovery must have all data needed to perform the process.

The charts in Figure 4-10 and Figure 4-11 show the calculated overhead of message logging over two protectors[3] using different message sizes, and the calculated overhead of checkpointing over two protectors[4]. Such an overhead may be significantly high and its employment for mission-critical applications inappropriate because of their time constraints.

Therefore, the main challenge to improving the system performability by providing a

$$T_{total} = NxT_{data} \quad (11)$$

[3] The logging overhead was calculated by multiplying the overhead with only one protector by two.

[4] The checkpointing overhead was calculated by multiplying by two the time spent checkpointing over tbe protector.

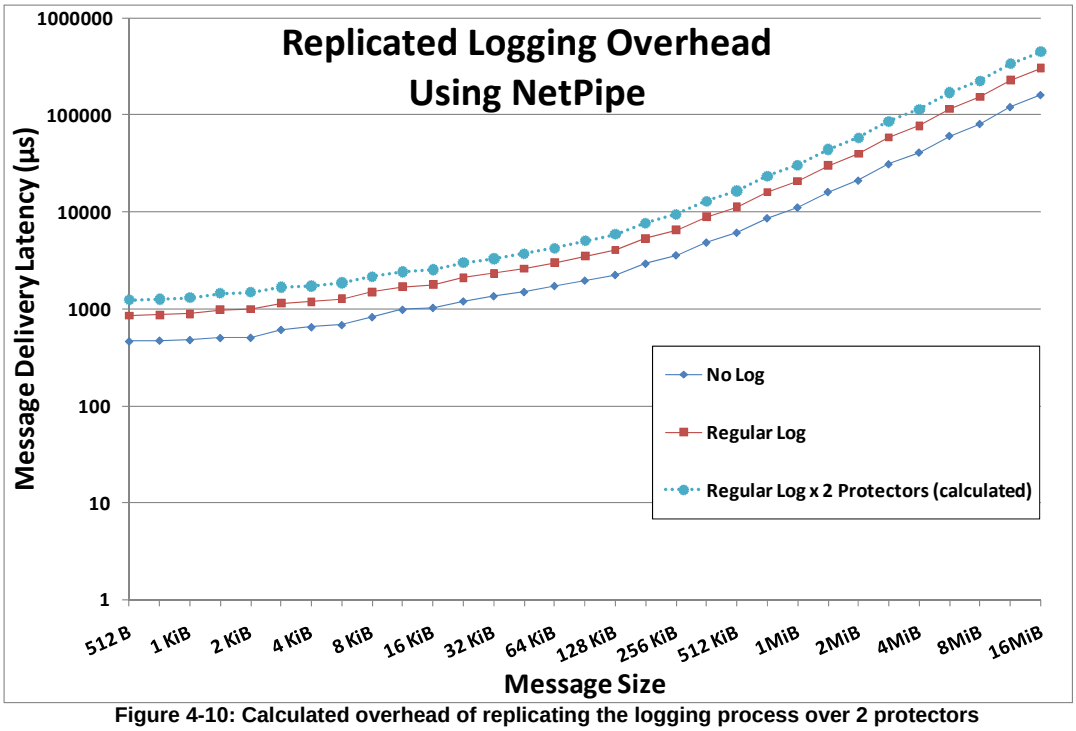


Figure 4-10: Calculated overhead of replicating the logging process over 2 protectors

higher degree of availability to mission-critical applications is to increase system availability without imposing a large overhead. The solution for accomplishing this objective now follows.

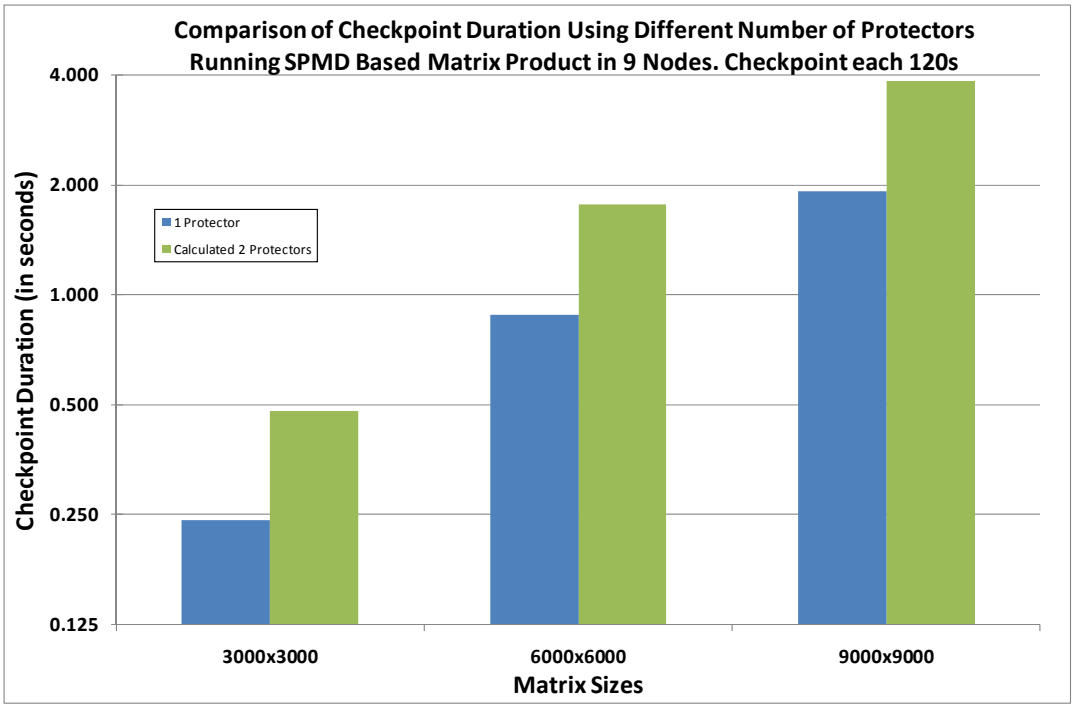


Figure 4-11: Calculated overhead of replicating the checkpointing process over 2 protectors

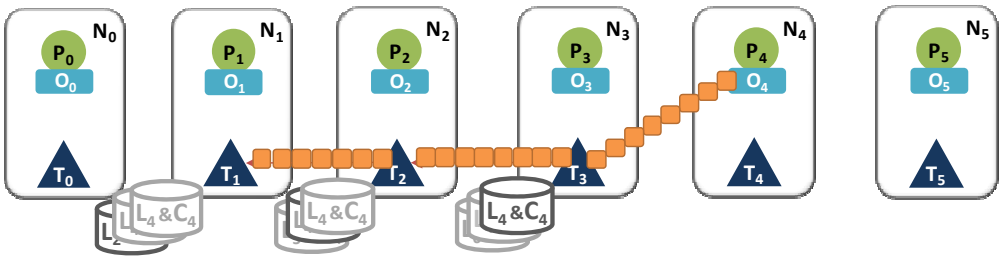


Figure 4-12: RADIC configuration using three protectors per observer and pipelining the redundant data replication.

4.3.1. Pipelined data replication

The primary site approach for data replication is a technique used to increase system availability and consists to replicate the data over N nodes, with one of these nodes designated the primary and the others designated backups. All data saving requests sent to the primary are forwarded to the backups. In RADIC, this approach means replicating redundant data through various protectors as explained in section 4.3. To overcome these issues, the data redundancy replication strategy of RADIC was modified. The process was parallelized by extending the pipeline approach presented in the section 4.2.1 for dealing with all redundant data over N protectors as shown in Figure 4-12 (the detection scheme is intentionally not depicted because it remains unchanged). In this approach the observer O_4 divides the data to be sent (checkpoint or message log) into small pieces and sends them to the first protector (T_3). Each protector then receives the first piece, and stores it in a local buffer (the dark grey disk drawings labeled L_4&C_4).It will forward all received pieces of this communication to the next protector, which stores each piece in its local buffer (the light grey disk drawings). The entire process finalizes when all involved protectors confirm the receiving of all pieces. Ideally, the total time to perform this operation is given by Equation (12). The penalization of performing replication is independent of the redundant data size. The following topics detail this solution.

$$T_{total} = T_{data} + NxT_{piece} \quad (12)$$

Changes in the protectors' operations

The protectors are the RADIC component most affected in this solution because of the need to deal with the pipelining process. In the original RADIC protocol, each protector only needs to store information on its two neighbors, the predecessor and the successor. But to implement the pipelining process, the protectors must now to store information regarding *N* predecessor protectors, where *N* is the number of replicas. For example in Figure 4-12 T_4 must now store information regarding T_1, T_2 and T_3. To accomplish that it creates a new structure called a predecessor's list (TABLE 4-3), which replaces the information on only one predecessor. This list is created at the start (Figure 4-13a). To keep this list updated, the protectors perform a message forwarding procedure to spread the changes in the list every time a fault occurs. This message is forwarded as many times as the number of replicas.

During the state saving activities, namely checkpointing and logging, when a protector starts to receive pieces of redundant data, information containing the number of replicas of this data to be stored is piggybacked on the first piece. If the number is greater than zero, the protector updates this data decreasing by one and starts to forward each received piece for its predecessor, storing a copy in a local buffer. Figure 4-13b clarifies this procedure.

TABLE 4-3: An example of *predecessor's list*

Predecessor identification	Address
0	Node 2
1	Node 1
2	Node 0
. .	. .

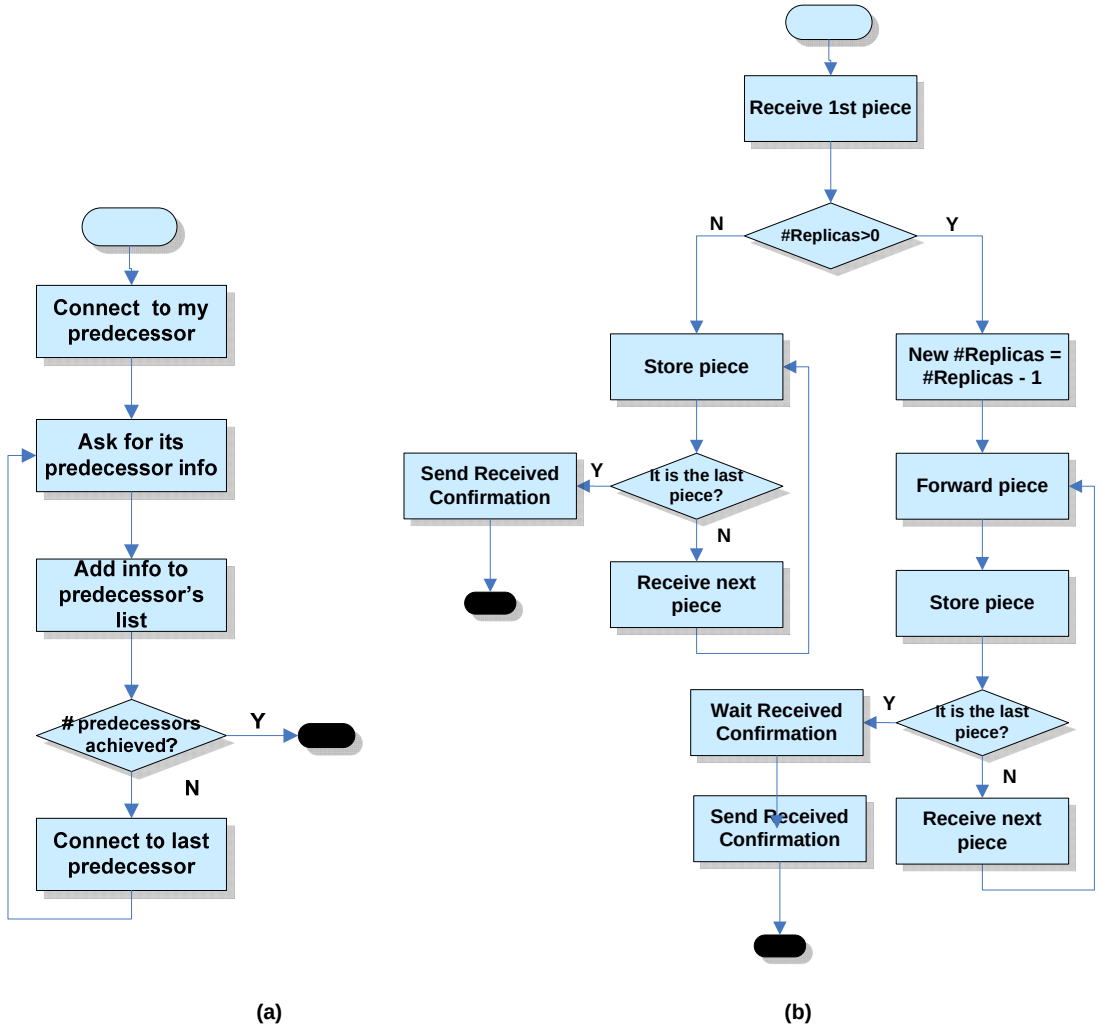


Figure 4-13: Flowcharts of (a) Predecessor's list creation and, (b) Redundant data forwarding

Each protector must also manage the redundant data replicas by knowing which stored replica belongs to which observer/application process. Hence, the original observer's list explained in the previous chapter is modified to include a field marking when an observer

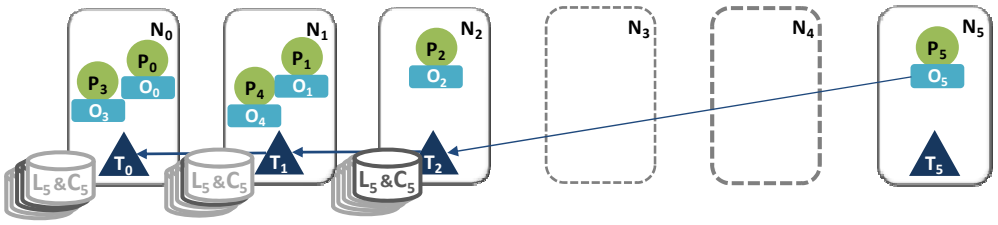


Figure 4-14: RADIC configuration using three protectors per observer after a concurrent correlated fault of 2 nodes

is just a replica. This change is useful at recovery time to decide which process to recover.

Moreover, additional changes were made in the recovery activity. Figure 4-14 depicts a RADIC configuration after the occurrence of concurrent correlated faults on nodes N_3 and N_4. In this situation, the protector T_2 has detected a fault in node N_3 and waits for a connection from other protector. Protector T_5 has a list with the predecessor protectors which have a replica of the redundant data, so it tries to connect with each one in sequence, until it reaches protector T_2. O_5 then establishes T_2 as its protector. Using the observer's list, protector T_2 decides which processes it must recover according to information received from T_5 regarding which nodes failed, i.e., as T_5 tried to connect each predecessor protector, it tells T_2 which nodes failed. T_2 must then apply a recovery policy to balance the process distribution over the nodes used in the recovery.

In Figure 4-14 a simple policy was used to recover to the last protector's node. In this situation, the remaining protectors store the redundant data of almost all processes (depicted as light and dark grey disks), which demands high storage space. The study of a more efficient policy, taking into consideration the node workload or available storage must be addressed in future works.

Finally, an important issue about the design of this solution was a fault occurrence in a node involved in the replication process. In this case, two scenarios can occur:

- The faulty node runs the first involved protector (the primary site); or
- The faulty node runs one of the protectors storing a replica.

In the former, RADIC acts according to its original protocol by finding a new protector and performing the recovery process. In the latter, the successor protector running in the

faulty node's successor continues to receive the data pieces and stores them in its buffer while the recovery is performed. After the recovery process has established a new predecessor protector, the successor then restarts the pipeline sending all data stored in its buffer.

Changes in the observers' operations

The observers are less affected by this approach than the protectors. The observers are now in charge of slicing the redundant data to be sent in similar way to that discussed in section 4.2.1. The observers may also decide how many replicas of the redundant data will be stored by piggybacking the desired number on the first piece. This feature allows greater flexibility when configuring RADIC, and permits different protection degrees for each application or even for each process. In a master-worker program, for example, it is possible to define more replicas to master than worker, or in a cluster with some fault probable nodes (because of their MTBF or something else) the number of replicas of the processes running on these nodes may be greater than the others.

4.4. Performance degradation because of faults

The RADIC architecture explained in the previous chapter is an example of a fault tolerant solution that uses only the active cluster's nodes to recover failed processes. As seen, the recovery process changes the system configuration, leaving the system with an unplanned process per node distribution.

Despite the high availability provided by RADIC, the aforementioned system configuration change left the system with one node less and leads to the presence of processes sharing a computing node. In this node, both processes will suffer a slowdown in their executions and a growth in memory usage that may lead to a disk swap. Moreover, these processes

access the same protector, and will compete to send the redundancy data. Supposing that a previous process distribution aiming to achieve a certain performance level according with the cluster characteristics was made, this condition becomes undesirable, especially if the application is unable to adapt itself to workload changes along the nodes.

Long running programs are highly susceptible to faults according to the system MTBF, meaning the likelihood of a fault constantly increases over time. In consequence, the probability of node losses and overloaded nodes gradually increases, which may lead to an impracticable situation. Figure 4-15 depicts a RADIC configuration with nine processes running in a cluster after the recovery of sequential faults in nodes N_5, N_4 and N_3, which means that the faults always occurred in an overloaded node. This figure demonstrates that in the node N_2 each process has a maximum of 25% of the available node's computing power. This can be a typical situation in clusters composed of thousands of nodes or running long programs.

Figure 4-15 also depicts other problem caused by successive recovered faults: All the processes running in the node N_2 are storing their checkpoints and logs in the same neighbor (N_1). Checkpoints are usually large in common scientific programs, and logs occur frequently in some kinds of applications, meaning this may cause an undesirable situation such as:

- The communication channel becomes a bottleneck due because of the intensive traffic between the nodes;
- A queuing of requests for checkpoint and log transmission in this protector may occur as each process is sending its checkpoint and log to the same protector; or

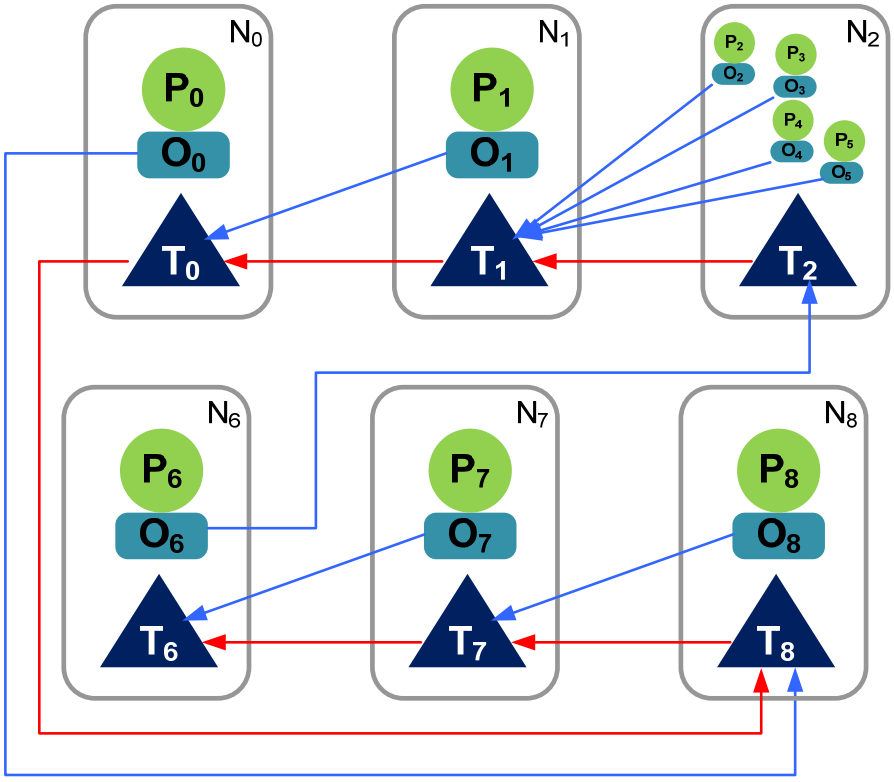


Figure 4-15: A RADIC cluster configuration after the recovery of sequential faults in nodes N_5, N_4 and N_3.

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

All these situations may slow down every process in the node and degrade system performance (Figure 4-16). This chart shows the results of execution of an N-body particle simulation program based on the example presented by Gropp et al (GROPP, W. et al., 1999, p.177), running in a 10-node circular pipeline, and observed during 50 minutes, taking checkpoints each 120 s . The two lines represent the throughput (in simulation steps) of the application: The first (with squares) represents a failure-free execution of the program, and the second (with circles) the execution with fault injection.

After each fault the performance, in this case the throughput measured in simulation steps per minute, degrades gracefully. The throughput starts at approximately 29 steps/m, and after each fault performance approximately decreases to a half, a third and a quarter sequentially, reflecting the two, three and four processes sharing the same node. Immediately after each fault a quick performance penalty because of the recovery process is visible.

This behavior is typically the subject of a performability study. In the aforementioned situation, RADIC could keep the application working, providing a constant degree of unavailability despite suffering performance degradation. Assuming the same method for estimating unavailability presented in section 3.7 (in this case the system MTBF refers initially to 10 nodes) a value of 0.00190% is found for the unavailability using RADIC and a value of

0.22779% when there is no fault tolerance solution. This value increases as faults occur because there are fewer nodes, increasing the system MTBF and consequently system

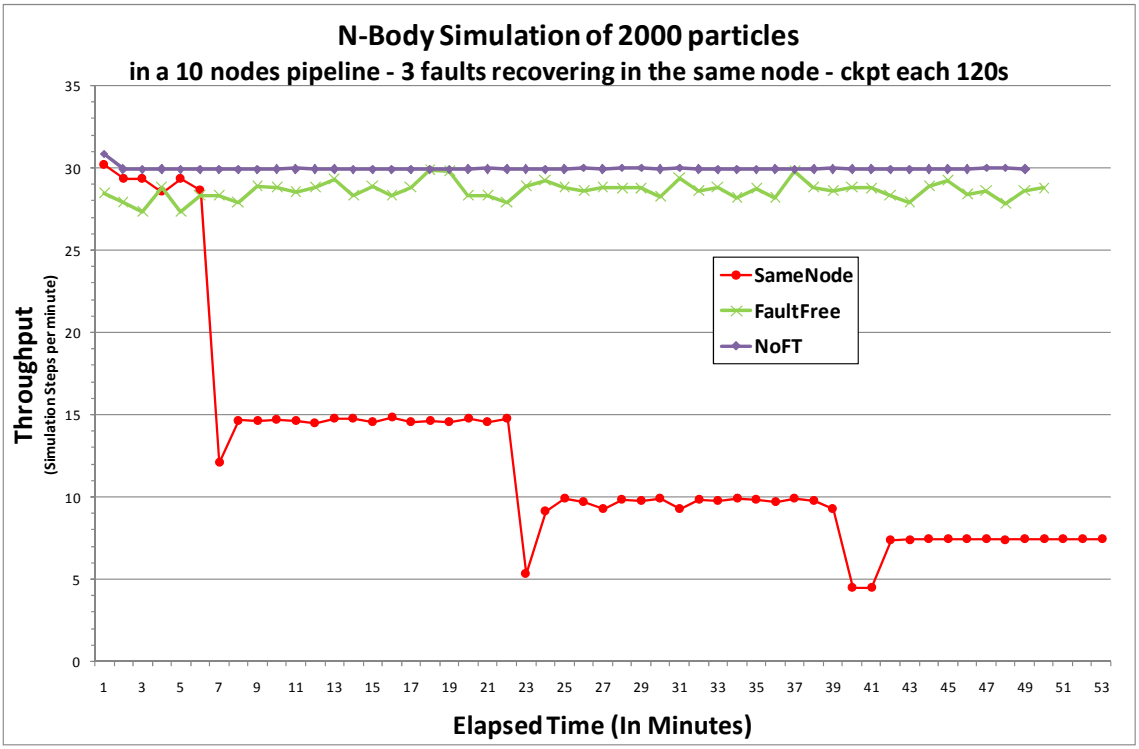


Figure 4-16: Result chart of an N-Body simulation after three faults recovered in the same node.

TABLE 4-4: Performability behavior of an N-body simulation after one, two and three faults recovered in the same node.

	Without fault tolerance	Fault-free	After one fault	After two faults	After three faults
Throughput (simulation steps/min)	**29.96**	**28.65**	**14.51**	**9.43**	**7.22**
Unavailability	**0.22779%**	**0.00190%**	**0.00171%**	**0.00152%**	**0.00133%**
Performability	**10.11761**	**25.19222**	**13.02734**	**8.66778**	**6.81544**

availability. The results in TABLE 4-4 are obtained by applying Equation (13): This equation is the same Equation (6) with *pw=1* and *uw=0.2*, which means that in this application the unavailability has an weight of 20% of the performance's weight (in mission-critical applications such a value may be equal, for example). Performability is strongly penalized by the unavailability when there is no fault tolerance, emphasizing the damage of a fault. Furthermore, it confirms performability decreases as faults occurs. After two faults, performability achieves values lower than without fault tolerance, suggesting a safe-stop of the application in order to re-establish the original process per node distribution (in case of available nodes).

Another evaluation takes into consideration task completion time. In this case, an important factor is the fault moment. Depending on the moment when the fault occurs, the disarrangement caused by the recovery process may affect the applicability of the obtained results. For example in a weather prediction program, that deals with many variables and has a well-defined deadline to produce its results, a large delay caused by performance degradation leads to the application producing obsolete results.

$$Perfomability_{System} = Avg\_Thruput \times \min\left(1, \frac{0.001\%}{Avg\_Unavail}\right)^{0.2} \quad (13)$$

An aggravation of this condition may occur in tightly coupled parallel systems where communication between processes is interdependent. If processes experience a slowdown in their execution they start to postpone their responses to other processes. These processes will then be held while they wait for a message from the slow nodes, propagating the slowdown by the entire cluster. Figure 4-17 shows the execution times of a SPMD implementation of a matrix multiplication using the Cannon's algorithm.

Each execution was performed using nine nodes of a cluster and one fault was injected at different moments (25%, 50% and 75% of the execution time). The tallest bars indicate more execution time. Therefore, having only one node sharing processes can cause considerable delays, even when the fault occurs close to the end of processing.

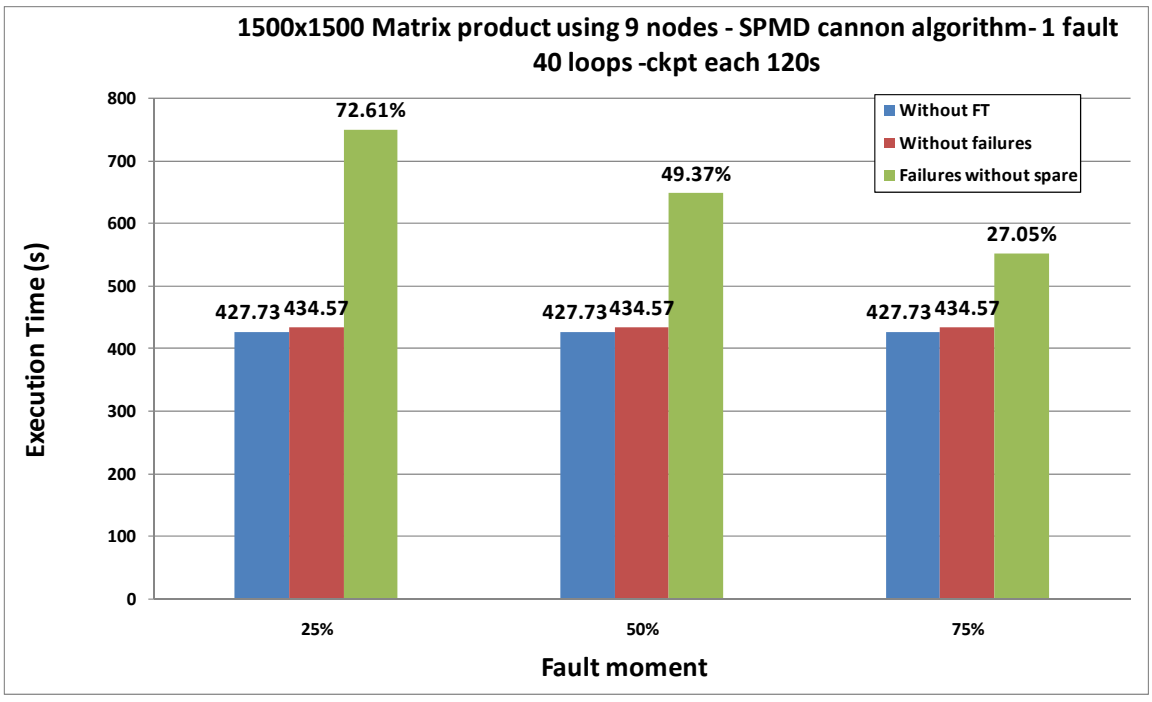


Figure 4-17: Execution times of a matrix product program implemented under the SPMD paradigm using a Cannon algorithm. Occurrence of one failure per execution at 25%, 50% and 75% of the execution time

TABLE 4-5: Performability behavior of an SPMD matrix product with faults occurring in different moments.

	Without fault tolerance	Fault-free	Fault at 25%	Fault at 50%	Fault at 75%
Throughput (elements/s)	5260.33	5175.98	2998.66	3465.91	4075.20
Unavailability	0.20506%	0.00171%	0.00157%	0.00162%	0.00166%
Performability	1814.001	4648.113	2740.116	3148.227	3680.271

The results[5] in TABLE 4-5 were obtained by repeating the same performability analysis performed with the N-body program. These values show a clear dependency on the fault moment in the performability, reflecting the elapsed time where a node shared its computing power. They also prove that there is one moment in the execution when before a fault occurs the measured performability will suggest the need to re-establish the distribution process per node.

The factors exposed and the results showed until now, demonstrate that the system configuration change caused by a recovery process using active nodes may produce unwanted system performability decrease because of performance degradation. This performance degradation may make impracticable the use of some applications that have time constrictions or demand all computing power possible such as mission-critical applications. Therefore, it is desirable that the fault tolerance solution avoids this phenomenon and more than ensuring the application completion, also protects the system configuration from the possible changes in order to assure the performability under the presence of faults.

[5] As discussed in the section 2.2, in order to achieve comparable performance values, 2250000 (a matrix of 1500x1500 elements) tasks were considered and divided by the elapsed time to perform the product in each situation presented in Figure 4-16. This resulted in the application throughput measured in elements/second. The unavailability values take into consideration the fraction of time when the application executed with one node less.

The previous examples are real cases using the RADIC architecture that confirm the behavior of throughput degradation presented in the introduction. Performability analysis provides a better understanding about the influence of fault tolerance solutions, and the resulting values can be used to determine when an application may or may not to take an action to restore its initial configuration.

Despite tolerating some kinds of faults correctly, preventive maintenance tasks in RADIC may lead to a system stop in order to replace the fault-probable nodes. These tasks may involve replacing many nodes at once. Therefore, rather than tolerating faults, avoiding preventive maintenance stops will improve performability. A mechanism allowing a hot swap of the fault-probable machines without needing to stop the running applications is, therefore, desirable. In a future, the integration with a fault prediction mechanism will allow fault avoidance.

4.5. Improving performability under the presence of faults

Section 4.4 explained the side effects of the recovery process in some fault tolerant solutions. This section discusses one solution for protecting the system from such side effects, i.e., the system configuration changes that a recovery may cause.

Under the performability concept, the RADIC architecture was modified to, beyond ensuring the correct finish of the applications, protect the system from performance degradation caused by fault recovery, preserve the planned process distribution (system configuration) and conciliate performance and availability.

For that, a new protection level in RADIC was designed (SANTOS, G. et al., 2008). Called resilient protection level, it provides a flexible dynamic redundancy feature protecting the system configuration from the possible changes imposed by a recovery process and the consequent loss of performability. The dynamic redundancy is based on presence of spare components ready to assume the work of failed ones. If these spares are active, but not working, they are called hot spares.

This new protection level introduces a fully transparent management of hot spare nodes in the RADIC architecture. The major challenge in the resilient protection level is to keep all the RADIC features and provide a mechanism for using and managing spare nodes in a fully distributed system. At this level, RADIC allows the restoration of the system configuration as the avoidance of active node losses by:

- Starting the application execution with a pre-allocated number of spare nodes (the spare nodes are used as faults occur until they reach zero); or
- Inserting new spares to replace the consumed ones. During the application execution spare nodes are consumed as needed, hence, this approach re-establishes the planned number of spares in the system. This approach is also useful for replacing failed nodes when there are no spares in the configuration.

Such a mechanism improves the RADIC performability without affecting its four major characteristics: transparency, decentralization, flexibility and scalability. Each one of these approaches is explained in detail below.

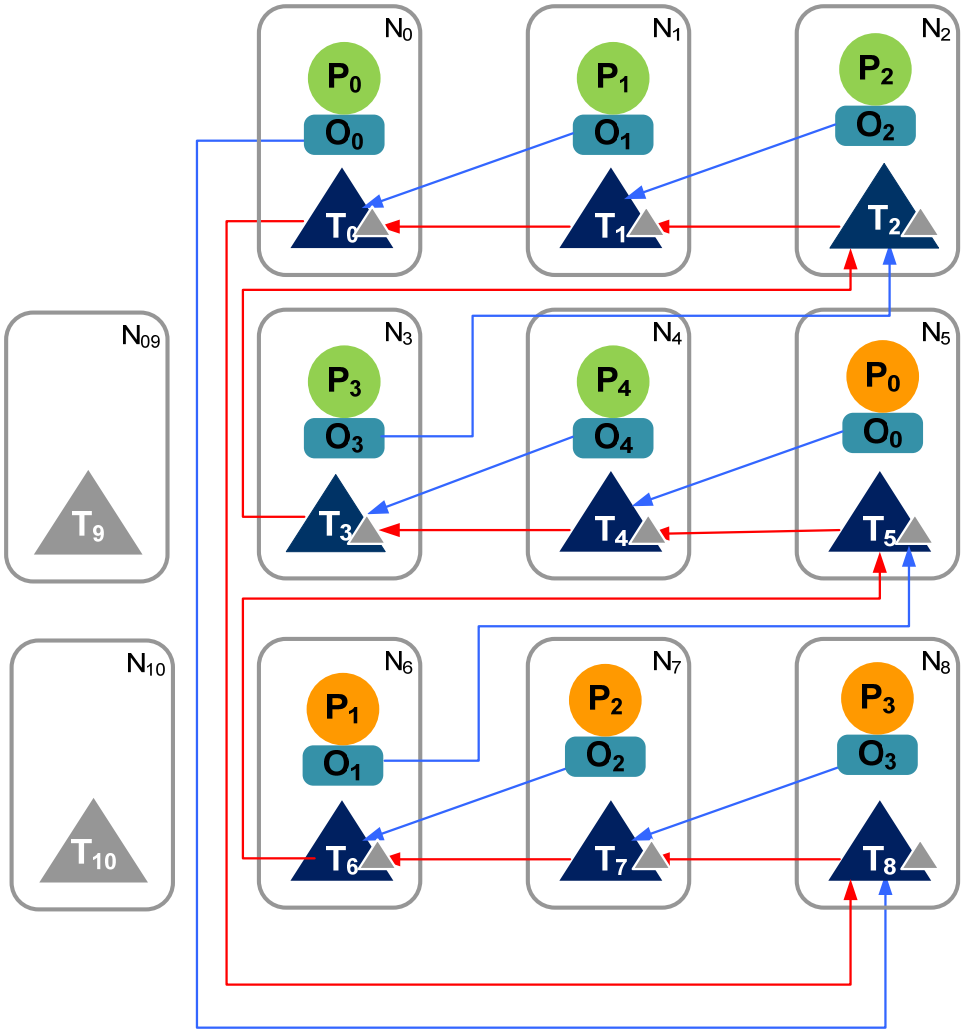


Figure 4-18: A cluster running two applications and using the resilient protection level with two spare nodes (N_9 and N_{10}).

4.5.1. Avoiding system changes

In this approach, the resilient protection level provides a mechanism that avoids the system configuration change mentioned in section 4.4 by providing a set of spare nodes to assume failed processes, instead of recovering in working nodes. A resilient protection level configuration can have any spare nodes as desired. Each spare node runs a protector process in a *spare mode*.

Such an approach aims to control the performance degradation generated by the original RADIC recovery process (henceforth called the basic protection level) once node loss is

avoided by replacing it with a spare node. The original flexibility is preserved by allowing as many spares as desired. This does not affect the scalability feature since the spares do not participate in fault tolerance activities except, of course, the recovery task. RADIC transparency is retained by a management scheme needing no administrator intervention and keeps all information regarding spares fully decentralized.

TABLE 4-6: A *sparetable* example of each protector in the cluster of Figure 4-18.

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

In this protection level, a spare protector does not perform the regular tasks described in section 3.2.1.It simply stays in a listening state waiting for a request. Figure 4-18 depicts a resilient level configuration using two spare nodes (N_9 and N_{10}). In this figure the spare node protectors are denoted in grey. These protectors do not participate in the detection scheme, avoiding a failure detection overhead caused by a workless node. Spares are available for any application running on the cluster, (in this case there are two applications) and each active protector carries the spare presence information, represented by a small grey triangle.

Each active protector maintains the information about the spares presence. This information is stored in a structure called a *sparetable*. TABLE 4-6 shows the *sparetable* structure. In the first column is the spare identification according to the same protector's identification. The second field is the physical address of the spare node. The third column indicates the number of observers (processes) running on this spare. This field is useful for indicating if the spare is still in an idle state, i.e., the number of observers is equal to zero.

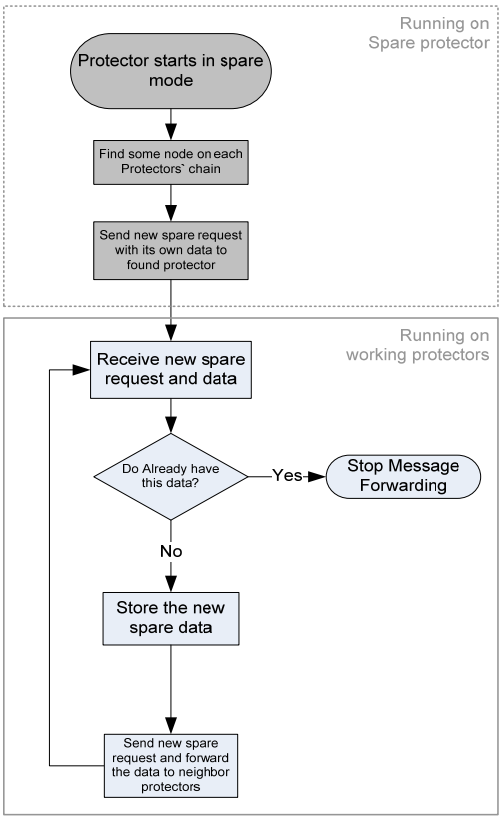


Figure 4-19: How a protector in spare mode announces itself to other protectors

4.5.1.1. How active protectors detect spare nodes

To keep RADIC as a fully distributed system, the spare nodes must spread their existence for all active nodes of the cluster. To achieve this, the protector, when starting in the spare mode, announces itself to the other protectors through a reliable broadcast based on the message forwarding technique (JALOTE, P., 1994, p.142).This technique was chosen because it does not affect the original RADIC scalability.

When running in spare mode, the protector searches an active protector running in the cluster and starts a communication protocol with him requesting its addition in its *sparetable*. The active protector receiving this request, searches whether the new spare data is already in its *sparetable*. If it is not, this protector adds the new spare data and forwards this request to its neighbors, passing the new spare information in sequence. Each protector performs the

same task until it receives an existing spare node data, which finishes the message forwarding process. Figure 4-19 clarifies this announcement procedure.

The procedure occurs before the application starts, while RADIC is mounting its *radictable* and just after the protectors started. Therefore, it is a latency caused by the initialization process, and is not considered overhead in the execution time. At the end of the spare nodes announcement, all the protectors have a spare list containing the data of all spares available. It is not critical for such a procedure to be performed atomically; it can be performed in parallel with the RADIC operation. If a fault occurs, RADIC recovers with its original protocol. When the announcement procedure reaches the recovered process, it will migrate to the new spare node inserted.

4.5.1.1. Recovering using spare nodes

In the resilient protection level, the original RADIC recovery task described in section 3.3.3.4 was modified to contemplate spare node use. Currently, when a protector detects a fault, it first searches for spare data in its *sparetable*. If there are idle spares, i.e., the number of observers reported in the *sparetable* remains equals to zero, it starts a spare use protocol. In this protocol, the active protector communicates with the protector running in the spare asking for its state, i.e. how many observers are running on its node. At this point, two situations may happen:

- If the spare answers that it already has processes running on its node, the protector then updates its *sparetable*, it searches for other spares and restarts the procedure. If the protector finds no idle spare, it executes the regular RADIC recovery task; or
- If the protector confirms the idle situation, it sends a request for its use.

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

- The protector (T_X) tells its predecessor (T_A) to wait for a connection from the spare (T_S) in order to be its new successor;
- Simultaneously, T_X commands T_S to connect to T_A and define it as its own predecessor;
- T_X instructs T_S to wait a connection from its future successor (T_X itself); and
- Finally, T_X connects to T_S, defining it as its predecessor.

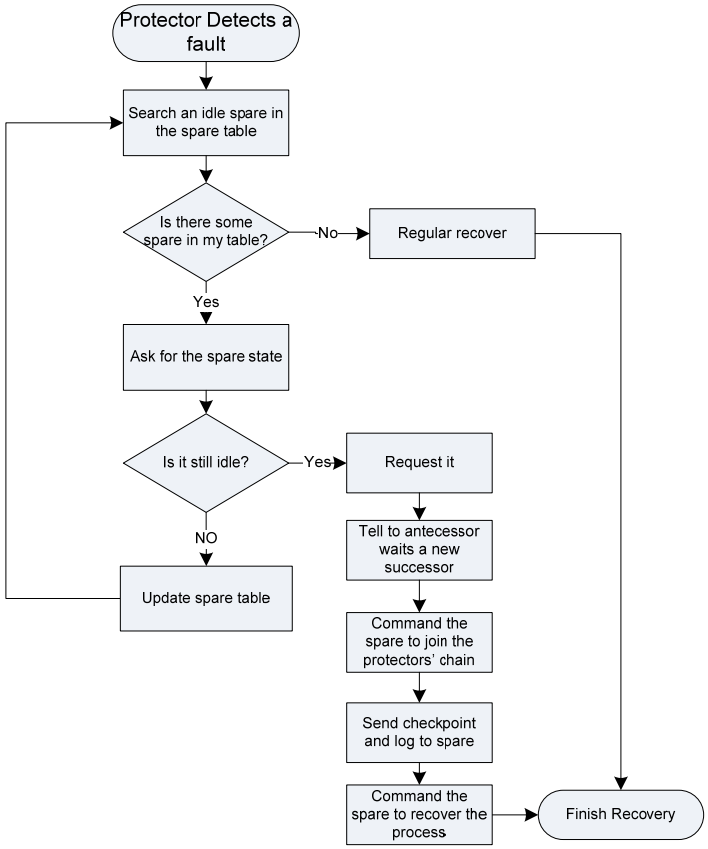


Figure 4-20: The recovery task using spare nodes (with the fault detected by the protectors)

Figure 4-21: Recovering tasks in a cluster using spare nodes: a) before fault; b) N_3 fails; c) the spare is connected; d) P_3 recovers in the spare

After finishing this step, the protector sends the failed process checkpoint and log to the spare, commanding it to start recovery of the failed process using the regular RADIC recovery process. Figure 4-20 clarifies this entire process. Figure 4-21 depicts the system configuration in four stages of the recovery task: a) Fault-free execution with the presence of spare nodes; b) a fault occurs in the node N_3; c) the protector T_2 starts the recovery by activating the spare N_9; and d) process recovered in the spare node

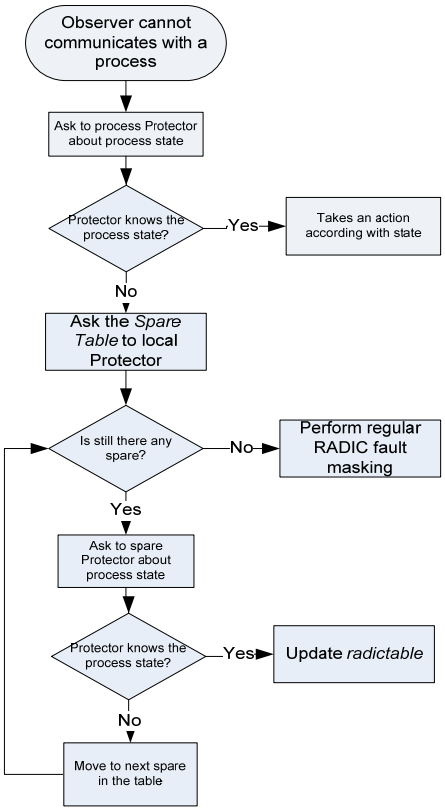


Figure 4-22: The new fault mask procedure

4.5.1.2. Changes in the fault-masking task

The original RADIC fault-masking task is based on a heuristic to determine where a faulty process will be running after the recovery. This heuristic was efficient in RADIC's basic protection level, because its recovery process is quite deterministic, i.e. the failed process always recovers in its protector's node. The resilient protection level inserts a small indeterminism in locating a failed process because it may have been recovered in any spare of the configuration.

To solve this indeterminism, a small change was implemented in RADIC's fault-masking task. This change consists of searching the *sparetable* for the faulty process after the observer failed to find the recovered process asking its original protector. However, as described earlier, only the protectors, not the observers, maintain the *sparetable* structure.

Hence, the communication between observers and the protector running in its node (the *local protector)* had to be increased. To execute the new fault-masking protocol, the observer asks the local protector for the *sparetable* and uses the information in this table to seek the recovered process in the spares. After the observer has found the recovered process, it updates its *radictable* with the new location, and does not need to perform this procedure again. Figure 4-22 contains the flowchart of this new procedure.

4.5.2. Restoring the system configuration

As explained in the previous item, the resilient protection level can maintain system performability by avoiding the system configuration change through the incorporation of transparent management of spares nodes. Such management allows it to request and use these spares without administrator intervention. Moreover, there is no centralized information about the existence of the spares, keeping faithful to the architecture's main principle of decentralization. The flexibility of this dynamic redundancy mechanism is its ability to start an application with a determined number of spares, or to include them dynamically during the application execution. This mechanism is explained below.

Long-term execution applications, such as 24 × 7 systems usually uses a fault tolerant solution based on redundancy to avoid degrading the system and retains its *performability*. If it is using dynamic redundancy provided by spare components, the system can retain the performance during a certain period. However, considering that the number of spares is fixed, they are going to request at each fault until it reaches zero. From this moment, the system starts to suffer some degradation after fault recoveries.

In the resilient protection level, such a happening can be avoided by restoring the initial system configuration. The procedure inserts new nodes to replace the used spare nodes.

Despite being performed at the start of an application, it may be executed at any moment during the program execution, without needing to stop the application. Using this procedure, failed nodes can return to the configuration as new spares after being repaired.

The same procedure can be used to replace faulty nodes. The announcement task was extended to permit it to request the spare at the announcement moment if some node of the configuration had already been overloaded, i.e. it has more processes executing than originally planned (according to the initial configuration parameters). This measure transfers the extra processes to the inserted node. If there are no overloaded nodes, this new node remains a new spare in the configuration.

Figure 4-23 shows the flow of this approach. Election policy may be applied to

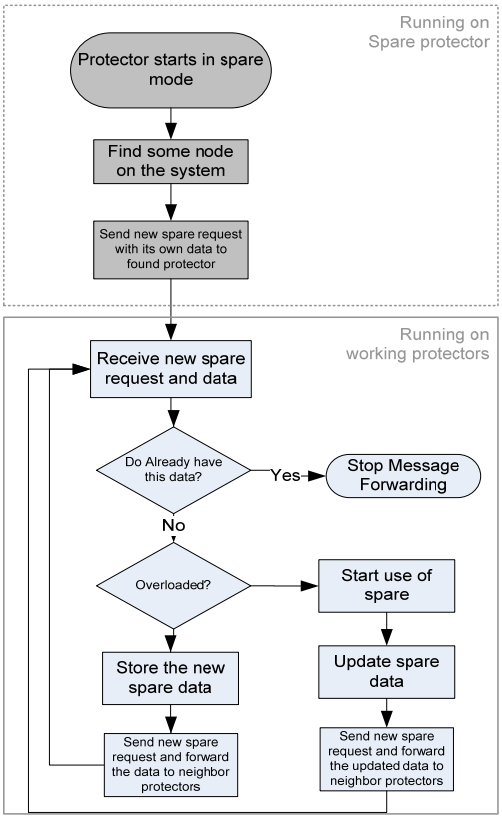


Figure 4-23: How a spare is used to replace a faulty node

choose what node will migrate their extra processes to the new node, i.e. the first node found or the most overloaded. After initiating the use of the new node, the protector updates the spare data informing that this spare is already in use and continues the spare spreading procedure. Thus all remaining protectors will already know about the existence of this node and which processes are running on it.

This approach has a limitation, which occurs if all original nodes have already been replaced. In this situation, the new spare does not how to discover a node of the application and needs extra information to be provided at the start of the procedure.

4.6. Providing a non-stop service

As discussed earlier, fault tolerance is a usual technique for avoiding interruptions during a program execution. Common approaches are hardware redundancy and data redundancy (generally checkpoint/restart). To reduce the probability of interruptions, the latter needs to provide transparent and automatic fault management. Another common technique is preventive maintenance, which usually consists of taking proactive action by periodically replacing fault-imminent or fault-probable components.

Unfortunately, hardware redundancy is expensive and inefficient in HPC, because of the need for as many redundant devices as the number of expected faults. Data redundancy, in turn, is relatively cheap. However, it may experience interruptions if the number of replacement nodes is exhausted or, in cases of using the same active resources, performance degradation reaches unacceptable levels. Preventive maintenance, despite avoiding fault occurrences, usually means interruptions when replacing components. However, some applications do not expect maintenance downtimes, meaning in the existence of a mechanism that allows these replacements without needing to stop the execution of such applications

The dynamic redundancy scheme present in the resilient protection level (explained in section 4.5) provides a scheme enabling the architecture to perform a scheduled hot replacement (without stopping the application execution) of a cluster node (SANTOS, G. et al., 2008). As such a level inserts the spare node after the application starts without requiring any stop in the program execution, all that is needed is to turn off the node to be replaced and their processes will automatically be migrated by recovering in the recently inserted node. As RADIC uses uncoordinated checkpointing, the entire application does not need to be stopped to perform these activities. However, highly coupled applications may present some slowdown during this process.

Such a mechanism is simple, and may be improved by implementing an automation feature that commands the machine to be replaced to automatically turn off, or take a checkpoint directly into the newly added node just before suicide. Other improvements may be including a fault prediction algorithm that chooses which of the machines to be replaced.

Chapter 5
Experimental Evaluation

5.1. Introduction

In the previous chapters, this thesis presented a study of performability issues in fault tolerance solutions, having RADIC as a study case. The goal is the providing of alternatives for improving the performability of computer clusters.

In Chapter 3, the RADIC architecture was analyzed and the factors influencing system performability were raised. These factors can be classified according two situations: with or without presence of faults, as listed below:

- Fault-free factors:
  - The overhead caused by fault tolerance activities (checkpointing and logging). In this case, the message logging overhead was identified as a major concerning, while the checkpointing overhead already was target of many researches;
  - The availability increasing, that allows the tolerance of concurrent correlated faults. This is a desirable feature for critical-mission applications and in RADIC it means to have many protectors per observer, leading to larger overheads; and
- Factors in the presence of faults:
  - After a fault the recovery process changes the system configuration leaving the system with a node less and processes sharing the computing capacity of a node in an unplanned manner. This may cause per-

formance degradation and consequently affecting the system performability.

- Preventive maintenance usually leads to an unavailable period of the node to be replaced, which leads to stopping the entire application running on this node.

Chapter 4 presented alternatives for improving performability addressing the raised issues with the solutions below:

- Reducing the message logging overhead by parallelizing this activity using the pipelining technique.
- Increasing the availability by applying the primary site approach for replicating the redundant data over the protectors and using the pipelining to obtain low overhead.
- Incorporating the resilient protection level based on dynamic redundancy. This protection level allows the insertion of spare or replacement nodes in any moment of the application execution. Such a solution transparently manages the request and use of spare nodes in a fully distributed way.
- The resilient protection level allows to perform preventive maintenance without need to stop the entire application by using a fault injection system. The fault is injected in the node to be replaced immediately after a checkpoint, and then the process running on it will migrate to a previous inserted spare node.

This Chapter presents a set of experiments that evaluate the effectiveness of the presented solutions. Different applications and benchmarks was used in various scenarios, representing the major parallel paradigms, evaluating in situations considered relevant and

comprising requirements such as scalability, flexibility, intensive computation or communication.

5.2. Experiment environment

Physical structure

To proceed with an experimental evaluation, two different clusters was used, named **Cluster A** and **Cluster B** as described below.

The **Cluster A** is formed by twelve computers with the following configuration: 1.9GHz Athlon-XP2600+ with 768 MB RAM and 40GB local disk. All nodes run Linux Kernel 2.6.17 and gcc v4.0.2 compiler. An Ethernet 100-baseTX switch interconnects all nodes. The network protocol is TCP/IP v4.

The **Cluster B** is formed by 32 computers with the following configuration: 2.8 G Hz Pentium 4HT, with 1GB of RAM and 80GB local disk. All nodes are run Linux Fedora Core 4 with kernel 2.6.11-1 and they are interconnected via a Gigabit Ethernet switch.

Prototype

All executions were performed using the RADICMPI prototype (DUARTE, A. et al., 2006). Currently, RADICMPI incorporates only basic MPI (SNIR, M. et al., 1998) functions, which includes MPI blocking and non-blocking peer-to-peer communications. RADICMPI also provides two important tools to perform experiments with fault tolerance: a fault injection mechanism and a debug log.

The generation of faults can be deterministic or probabilistic. In deterministic testing, the tester selects the fault patterns from the domain of possible faults. In probabilistic testing,

the tester selects the fault patterns according to the probabilistic distribution of the fault patterns in the domain of possible faults.

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

The mechanism is implemented at software level. This allows a rigorous control of the fault injection and makes easy the construction and operation of the fault injection mechanism. In practice, the fault injection code is part of the code of RADICMPI.

The RADICMPI debug log mechanism serves to help in the development of the RADICMPI software and to validate some procedures. The mechanism records the internal activities in a log database stored at the local disk of each node. TABLE 5-1 describes each field of the debug log database. The database has the same structure for protectors and observers

Benchmarks and Applications

To evaluate the message logging latency, NetPIPE (SNELL, Q. O. et al., 1996) was used and a simple token pass program. To perform the evaluation experiments, various kinds of parallel programs was applied: SPMD and master-worker matrix product, an N-body par-

TABLE 5-1: Fields of the debug log

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

ticle simulation using non-blocking functions in a pipeline approach, the Travelling Salesman Problem (TSP) program implemented in a master-worker fashion. Their descriptions are below.

NetPIPE Network Performance Evaluator

NetPIPE performs several latency round-trip measurements, increasing the message size after each set of measurements. For each message size, NetPIPE calculates how many messages have to be sent to reach a confident result and calculates the average round-trip time. Because of round-trip behavior, the measurement may be affected by the concurrent communications explained in section 3.6. For this reason, the message latency was also measured using a simple token pass program. This program sends one message each time through the running nodes, avoiding possible concurrency.

Matrix Product

The matrix product is a common operation used as kernel for many scientific applications and is also possible to apply different parallel paradigms over it. A master-worker and a SPMD algorithm were used, which makes easy the creation of different scenarios. Figure 5-1a shows the MW algorithm message pattern (1-to-N). The master process communicates with all the worker processes. Each worker only communicates with the master process. The SPMD version is based on the Cannon's algorithm and communicates in a mesh (Figure 5-1b). Each application process communicates with their neighbors, representing a tightly coupled application.

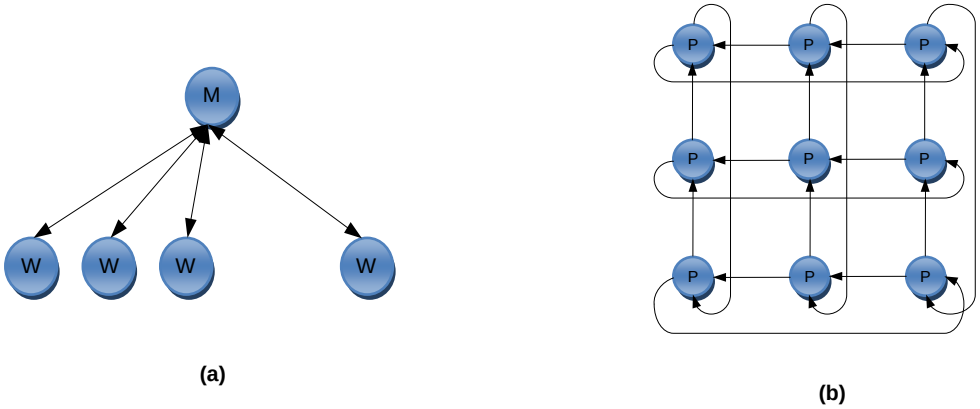


Figure 5-1: Message pattern of a matrix-multiplication using a) M/W paradigm and b) SPMD paradigm.

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

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

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

N-Body Particle Simulation

The N-Body program is based on the example presented by Gropp (GROPP, W. et al., 1999, p.117). This program performs a particle simulation, which calculates the attraction forces between them. It is implemented under the pipeline parallel paradigm and uses non-blocking MPI communication functions to overlap communication with computation. Figure 5-2 represents the flow of the actions performed by each process.

Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a combinatorial problem where the objective is to find the shortest path to a salesman needing to visit once and only once each city

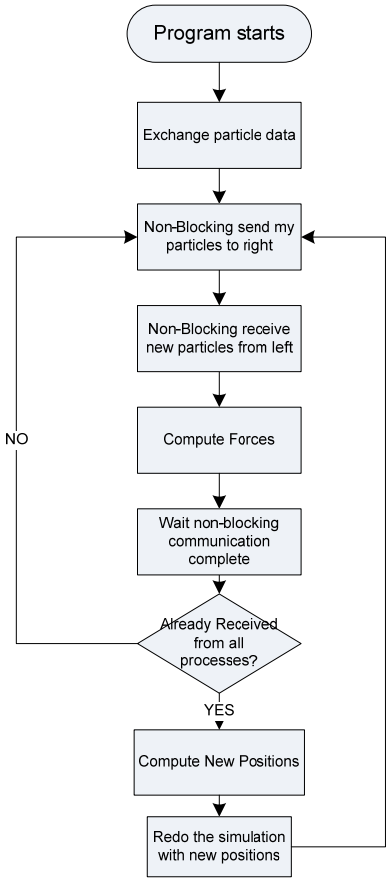


Figure 5-2: The N-Body particle simulation flow

from a set of cities, starting from a base city and returning to this city.

The algorithm used in this work is master-worker based, where the master defines a level L for dividing N cities in tasks that are sent to workers to calculate the permutation of N cities in L elements. Figure 5-3 shows a diagram of permutations for 5 cities and a division level of 2, in this case the Master calculates the permutations at level 0 and 1 and to generate 12 tasks to be sent to workers.

The TSP algorithm used in this work belongs to the class of exact search algorithms, and applies the branch-and-bound technique (GUTIN, G. and Punnen, A. P., 2007). This algorithm was chosen because of its importance, and because is computation intensive with small communications and process state size, which may not take benefit from the pipeline approach. The program uses the most direct solution, which tries all permutations (ordered combinations) and see which one is the cheapest (using brute force search). The running time for this approach lies within a polynomial factor of O(n!), the factorial of the number of cities.

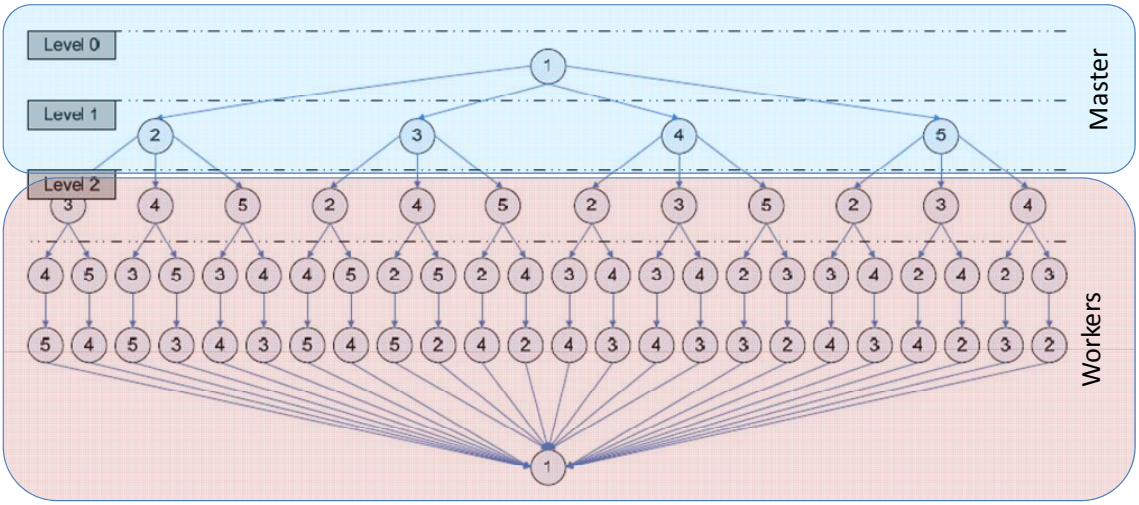


Figure 5-3: Possible permutations for TSP using 5 cities and division level of 2.

5.3. Experimental Results

The performability of presented solutions was evaluated through a series of experiments under different configurations comparing with not using the solution. These experiments comprise the use of aforementioned applications/benchmarks.

5.3.1. Evaluating pipelined logging

The pipelined logging was initially evaluated using the NetPIPE network performance evaluator. The program was executed in the **Cluster B** using a piece size fitting in the Gigabit Jumbo Frame MTU (9000 bytes) over 4 nodes (the communications are only between two nodes), measuring the message delivery latency and comparing the pipelined logging versus the traditional approach. Figure 5-4 shows the result of this execution. The vertical axis is represented using logarithmic scale because of the high latency variation over the different message sizes. The measured values are not continuous, they are plotted as lines for the ease of visualization. The line with diamonds represents the latency of message delivery without logging, which serves a comparison basis for other cases.

The latency of the pipelined log approach (line with triangles) is slightly greater than the regular logging approach (line with squares) when the message is small. This behavior occurs because in small messages there is no enough iterations to overcome the overhead caused by implementation issues such as internal controls or intrinsic message-passing overhead (extra headers and acknowledges).

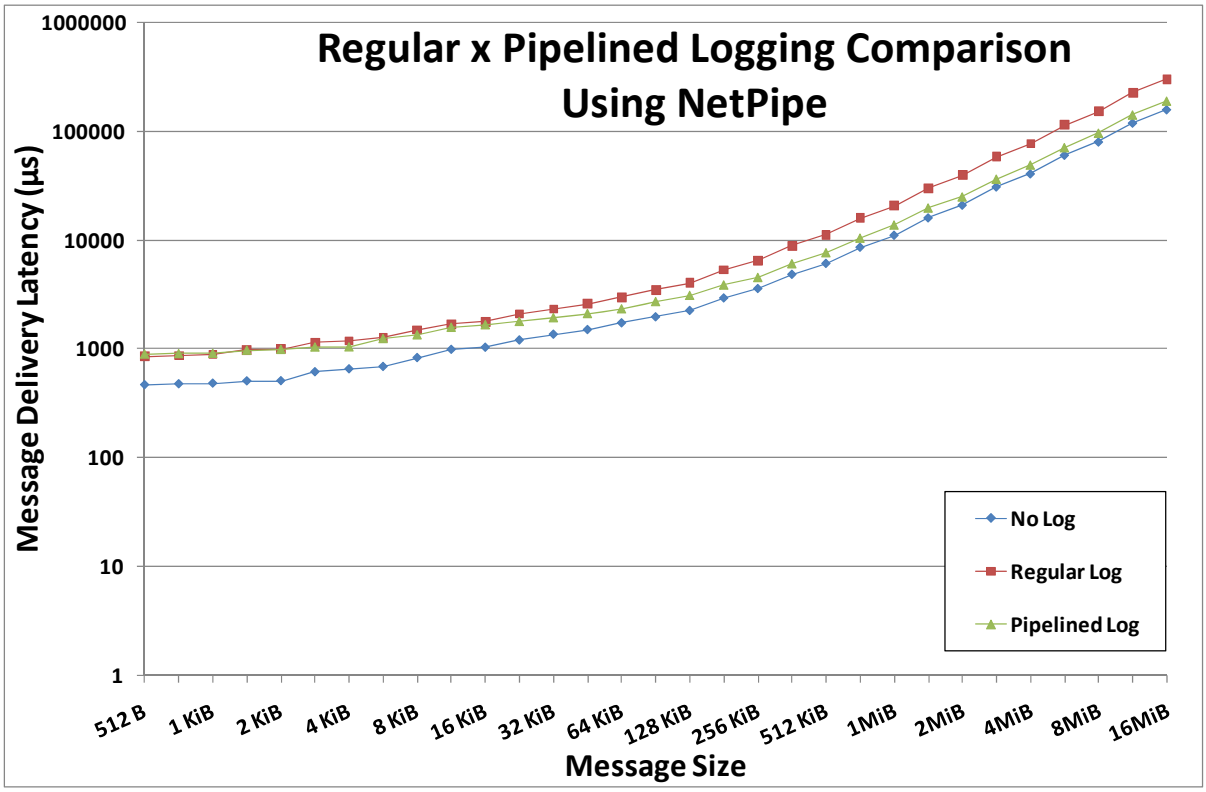


Figure 5-4: Message latency comparison between using or not pipelined message logging with NetPIPE

To provide a more accurate analysis, TABLE 5-2 presents the comparison of the overheads of the two approaches (pipelined x traditional logging) in relation to not use mes-

TABLE 5-2: Overhead comparison between using between using or not pipelined message logging with NetPIPE

Message Size	Regular Logging	Pipelined Logging
512 B	83.4%	92.7%
768 B	82.8%	90.7%
1 KiB	85.0%	89.3%
16 KiB	72.9%	62.0%
64 KiB	72.3%	35.2%
128 KiB	80.5%	38.3%
256 KiB	81.4%	26.7%
512 KiB	84.1%	25.7%
1MiB	86.3%	24.4%
2MiB	87.9%	19.3%
4MiB	88.5%	20.5%
8MiB	89.3%	20.3%
16MiB	89.7%	19.8%

sage logging. The pipelined log performance starts to get better as the size is greater than the 16 KiB approaching to the No Log latency in larges messages, meaning that the pipelining benefits are overcoming the aforementioned overhead.

The overhead of traditional logging is below of expected, which would be around 100%. This behavior occurs because of the RADICMPI reception buffer that is always accepting messages. As NetPIPE measures the round-trip message latency, part of sending and receiving messages are overlapped, resulting in lower round-trip latency. To evaluate the solution free of this behavior, it was evaluated using a token pass program, which sends a message to the next process in a circular fashion, thus, the overlapping is avoided because a process must to wait the token pass by all processes.

Figure 5-5 shows the comparison of the overheads of the two approaches in relation to

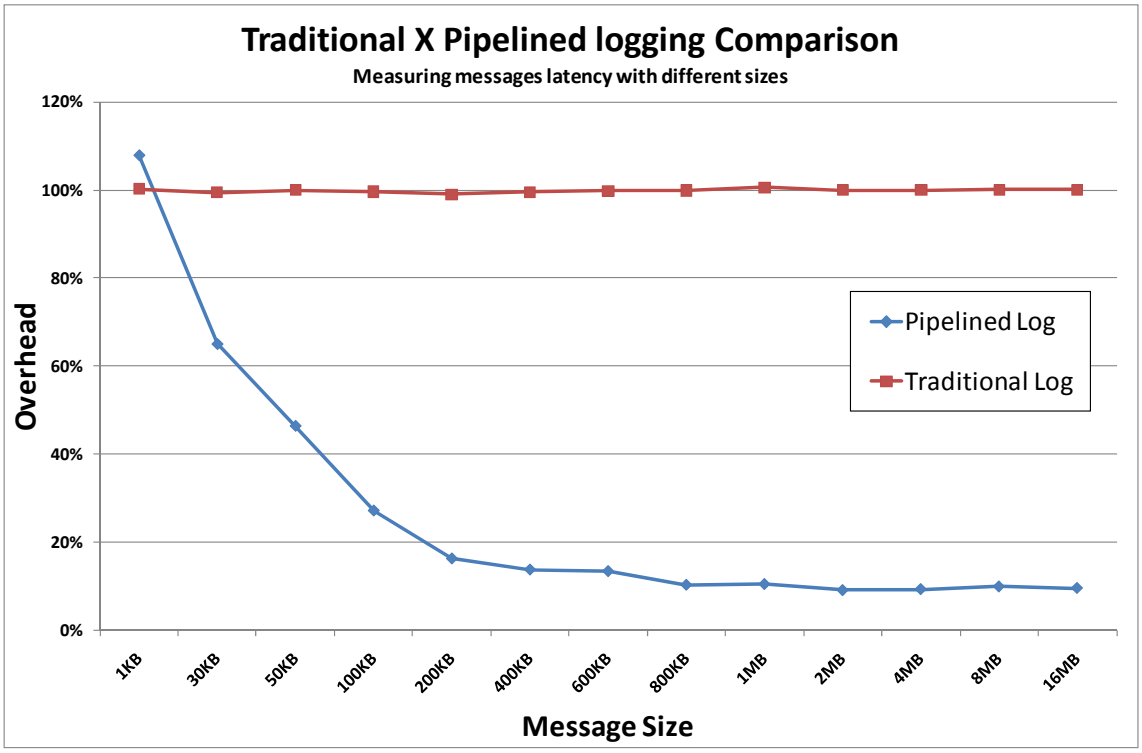


Figure 5-5: Overhead comparison between using or not pipelined message logging with a token pass program

not use message logging. In these results the overheads of traditional logging perform as expected at approximately 100% of a regular message latency, while the pipelined logging performs as low as 10% in large messages, but slightly worse (about 7%) in small messages. The following experiments about the message delivery latency will use this program because its results are not influenced by the RADICMPI reception buffer.

Evaluating according to the network type

The performance of pipelined logging was also evaluated in the two available networks (**Cluster A** and **Cluster B**) using a piece size of 1460 bytes (that fits in the Fast Ethernet MTU), such an experiment enforced the idea of an existent relation between the piece size and the underlying network. Figure 5-6 shows the result of these executions where the blue line with diamonds represents the latency of different message sizes in Fast-Ethernet network and the red line represents the latency of these messages in Gigabit-Ethernet network. The performance of Gigabit-Ethernet is getting worse as the message size increases (because of the undersized piece). The performance of Fast-Ethernet increases according with the message size and stabilizes at approximately 10% of overhead. Such an experiment also suggests that the piece size can be dynamically defined according to the message size because in small messages the Gigabit-Ethernet the chosen piece performed better (because of the inherent latency of this network), such an idea can be addressed in a future work.

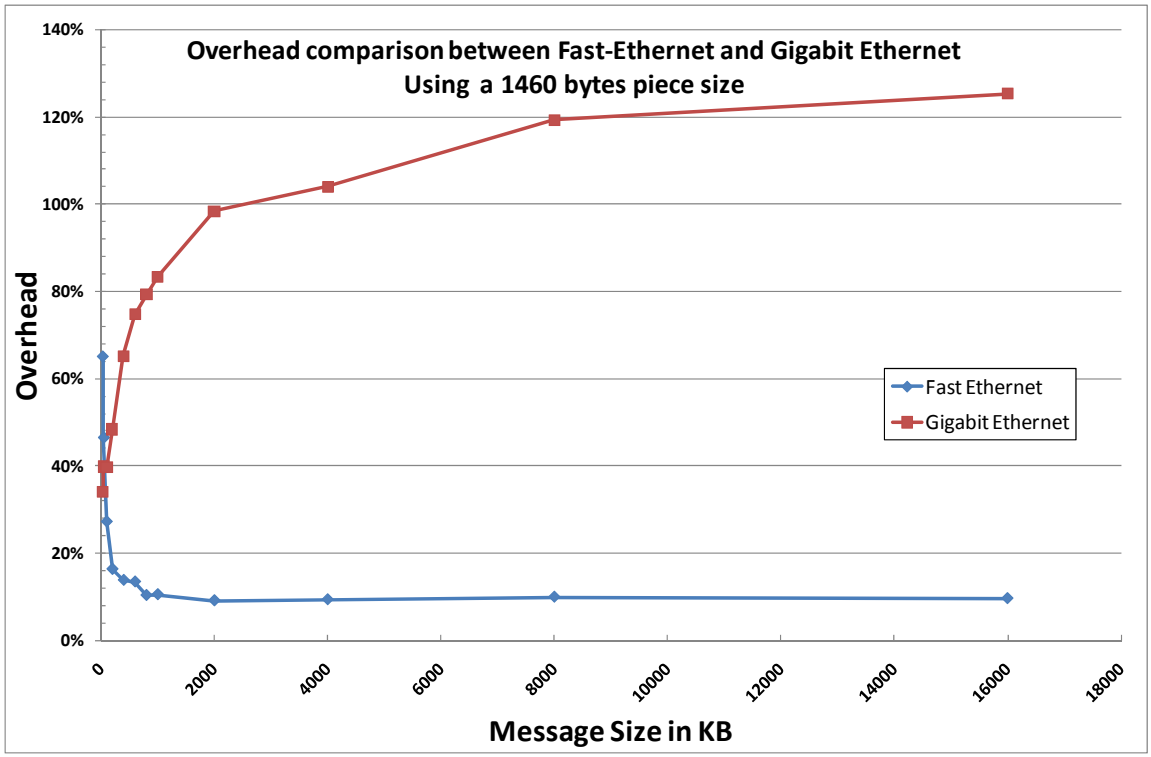


Figure 5-6: Pipelined logging overhead comparison between Fast-Ethernet and Gigabit-Ethernet networks using a 1460 bytes piece size

Evaluating according to piece size

An experiment varying the piece size in a same network was conducted to confirm the idea of influence of the piece size in pipelined logging performance. Figure 5-7 shows the behavior of message delivery latency using three piece sizes: 1460 bytes (fits in Fast-Ethernet), 4096 and 8192 (fits on Gigabit-Ethernet). The experiment was conducted over the network of Cluster B (Gigabit-Ethernet), and as expected when the piece size fits in the underlying network MTU, the pipeline logging performs better than with other values. Furthermore, as the message size increases, incorrect piece sizes worsen the pipeline logging performance.

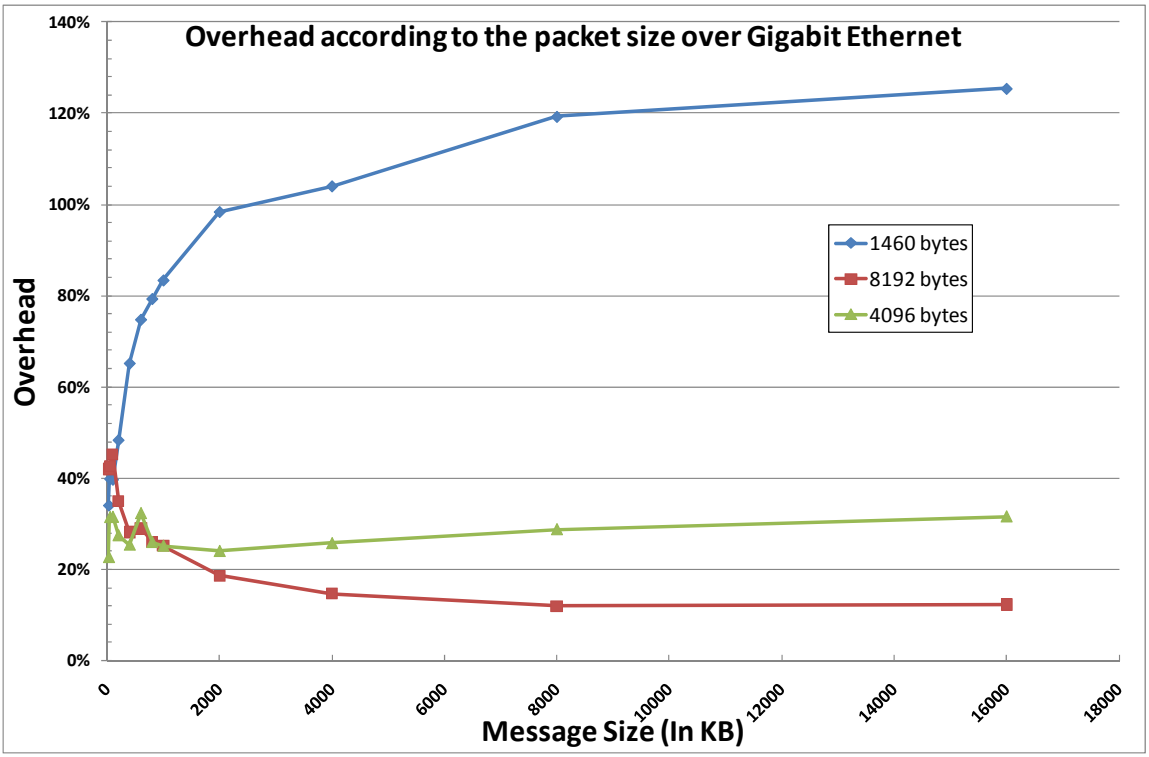


Figure 5-7: Pipelined logging overhead comparison between different piece sizes over a Gigabit-Ethernet network

Evaluating execution times

Until now, the effectiveness of the pipeline logging was evaluated individually, only measuring the message delivery latency. To assess the benefits in an application, the SPMD matrix product based on Cannon's algorithm was executed over different number of nodes (4, 9, 16 and 25 nodes, consequently the message size varies according to the nodes count) with a 9000×9000 matrix and using the Cluster B. In these executions, checkpointing was deactivated, performing only message logging. Figure 5-8 shows the measured elapsed times of these executions where the benefits of pipelined logging can be seen, reducing the overall performance overhead of fault tolerance in this application from 10.60% to 2.71% in the worst case and more than five times in the best case. Therefore, reducing the performance overhead, the pipelined logging improves the performability of this cluster running this application since the availability remains unaltered.

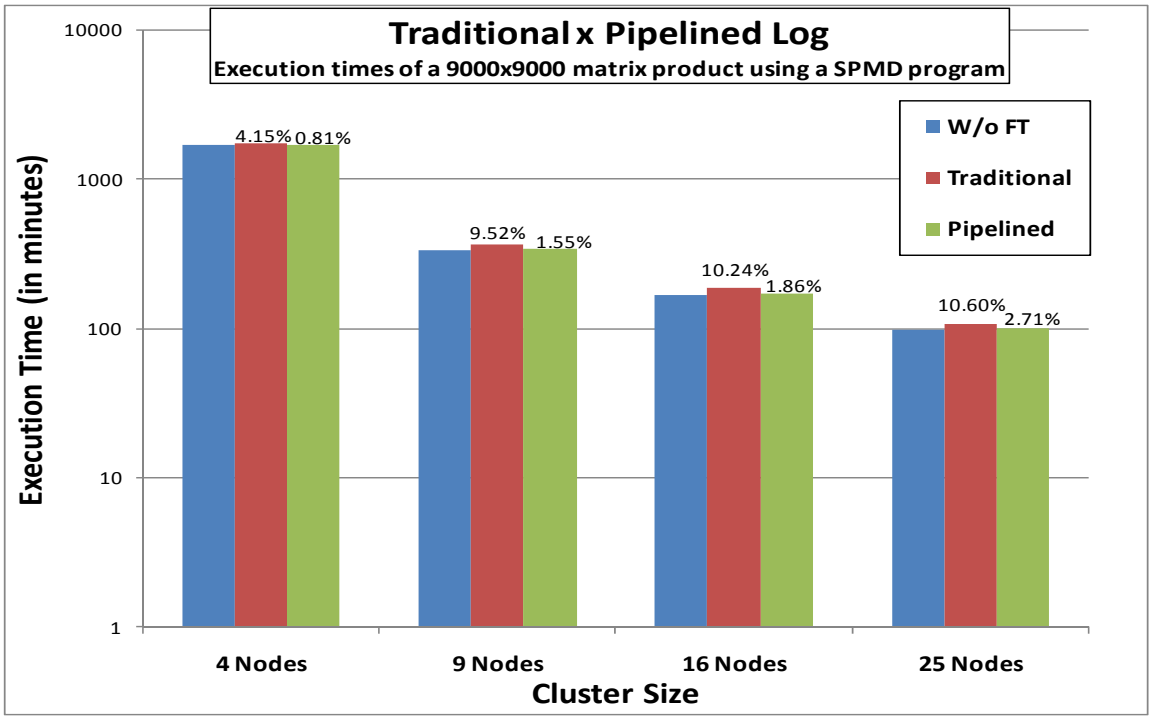


Figure 5-8: Execution time comparison of a 9000x9000 matrix product over different cluster sizes between not using log, and using pipelined or traditional message logging

Additionally, experiments with the Travelling Salesman program were also performed. This program is computation intensive with small messages, in the case of this experiment, which uses 15 cities and branches at level 2, the message sizes were a maximum of 120 bytes. The results of executions on each scenario are very similar because the program performs dynamic load balancing and messages are small, therefore the pipelined logging presents no improvement. However, the major cause of the overheads is checkpointing activity. The checkpoint sizes are 64 MB approximately and as the protectors implementation performs an asynchronous storage of the redundant data (they first receive all data, acknowledge it, and after stores at the disk), the overhead of fault tolerance is as low as 0.28% in the worst case using pipelined logging that is very near to 0.23% of using a regular approach.

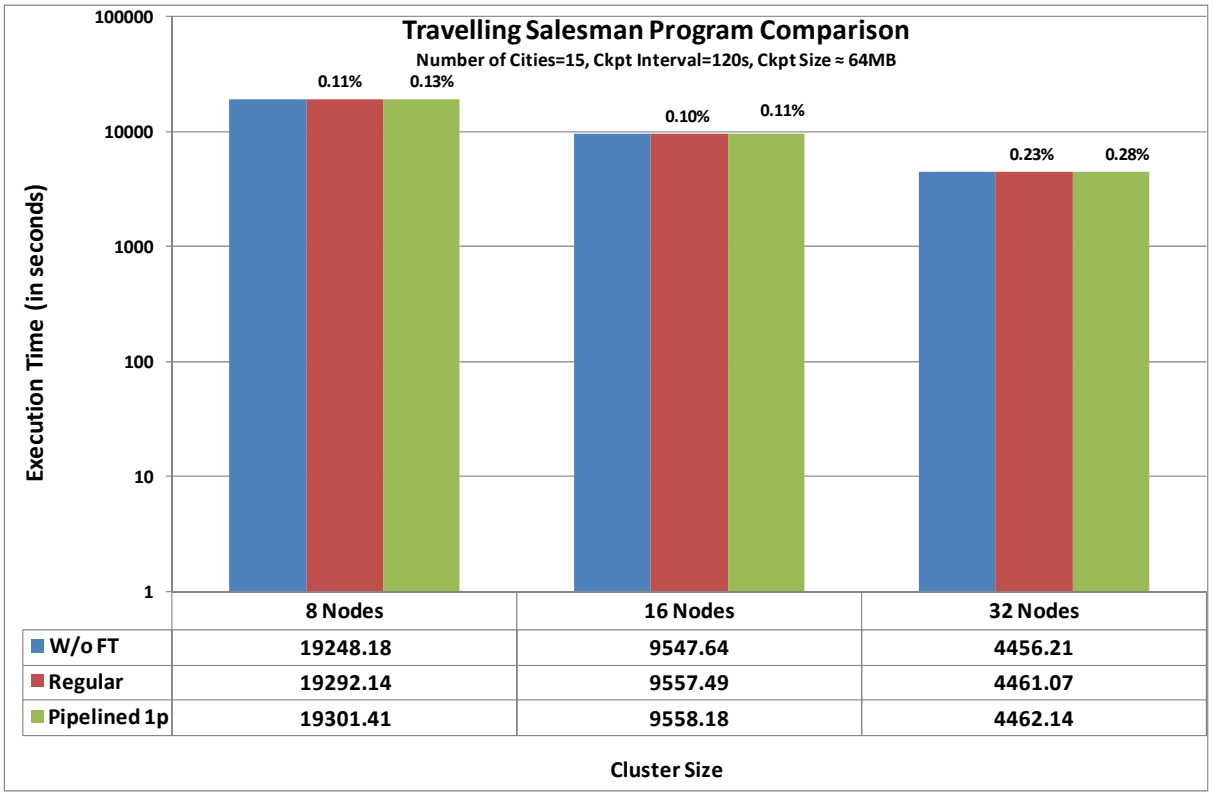


Figure 5-9: Execution time comparison using the travelling salesman program with 15 cities, comparing between not using log, and using pipelined or traditional message logging

5.3.2. Evaluating N-protectors data replication

Similarly, replication of redundant data over N-protectors was evaluated. The message delivery latency and the checkpointing duration were initially evaluated separately and, in sequence, using applications.

Message logging evaluation

For the message logging evaluation it was used the NetPIPE evaluator running in the cluster B. This experiment is very similar to the previous one presented at the pipelined logging evaluation, but including additional series about the number of protectors involved in the replication. Figure 5-10 presents the results of theses executions. The line with diamonds represents the latency of message delivery without logging, which serves a comparison basis for other cases. The line with circles represents the calculated latency of the regular logging

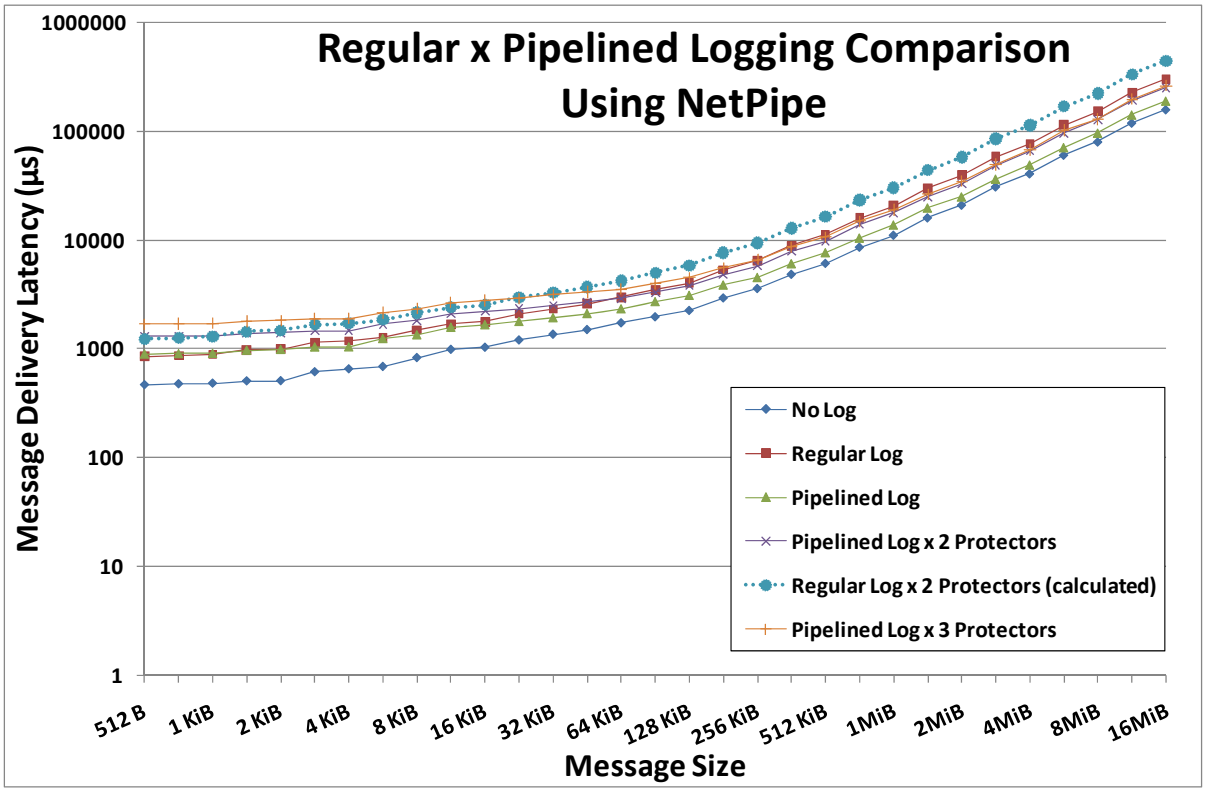


Figure 5-10: Message latency comparison using NetPipe and applying different number of protectors and message sizes

using two protectors. Such values were calculated applying two times the same overhead measured with only one protector. Similarly, the latency of using 2 protectors in the pipelined replication is greater in small messages, and it reaches values even better than the regular logging with only one protector when the message size is greater than 64 KiB. The logging latency values obtained with the pipelined replication using 3 protectors (line with crosses) figure to be slightly smaller than the regular log using only one protector when the message size overcomes 256 KiB getting better as the message size increases.

The TABLE 5-3 gives a numerical comparison of the overhead in some messages presented in the previous chart, allowing a better comprehension of these results.

Checkpointing evaluation

The effectiveness of the pipelined replication in the checkpointing activity was eva-

luated using the Cannon's matrix product with different matrix sizes over 9 nodes. It measured the average duration (in seconds) of sending and storing each checkpoint using the following matrix sizes: 3000x3000, 6000x6000 and 9000x9000 of double float precision elements (8 bytes), generating checkpoints sizes of approximately 24MB, 96MB and 216 MB respectively each 120s. The chart in Figure 5-11 shows the results of these executions grouped by checkpoint sizes. In this chart the vertical axis is represented using logarithmic scale because the values variation. The columns representing each scenario are in sequence:

- Regular checkpointing using 1 protector (R1P);
- Pipelined checkpointing using 1 protector (P1P);
- Calculated regular checkpointing using 2 protectors (R2P*);
- Pipelined checkpointing using 2 protectors (P2P);
- Pipelined checkpointing using 3 protectors (P3P);

TABLE 5-3: Numerical overhead comparison of message logging pipelined replication using NetPIPE

M. Size	RL	P1P	P2P	R2P	P3P
1 KiB	85.0%	89.3%	175.4%	170.0%	257.5%
2 KiB	96.8%	95.3%	178.5%	193.6%	259.8%
4 KiB	82.1%	60.1%	126.5%	164.3%	193.2%
8 KiB	80.5%	62.8%	120.6%	161.1%	183.8%
16 KiB	72.9%	62.0%	116.5%	145.7%	171.5%
32 KiB	71.0%	42.5%	85.3%	142.1%	131.1%
64 KiB	72.3%	35.2%	68.8%	144.6%	105.4%
128 KiB	80.5%	38.3%	69.9%	161.0%	100.6%
256 KiB	81.4%	26.7%	62.5%	162.8%	85.0%
512 KiB	84.1%	25.7%	60.2%	168.2%	74.5%
1MiB	86.3%	24.4%	61.2%	172.6%	70.9%
2MiB	87.9%	19.3%	56.1%	175.7%	64.8%
4MiB	88.5%	20.5%	61.1%	177.0%	65.5%
8MiB	89.3%	20.3%	59.6%	178.7%	61.0%
16MiB	89.7%	19.8%	57.9%	259.8%	65.1%

RL – Regular Log
P1P – Pipelined using 1 Protector
P2P – Pipelined using 2 Protectors
R2P – Regular using 2 Protectors (calculated)
P3P – Piepelined using 3 Protectors

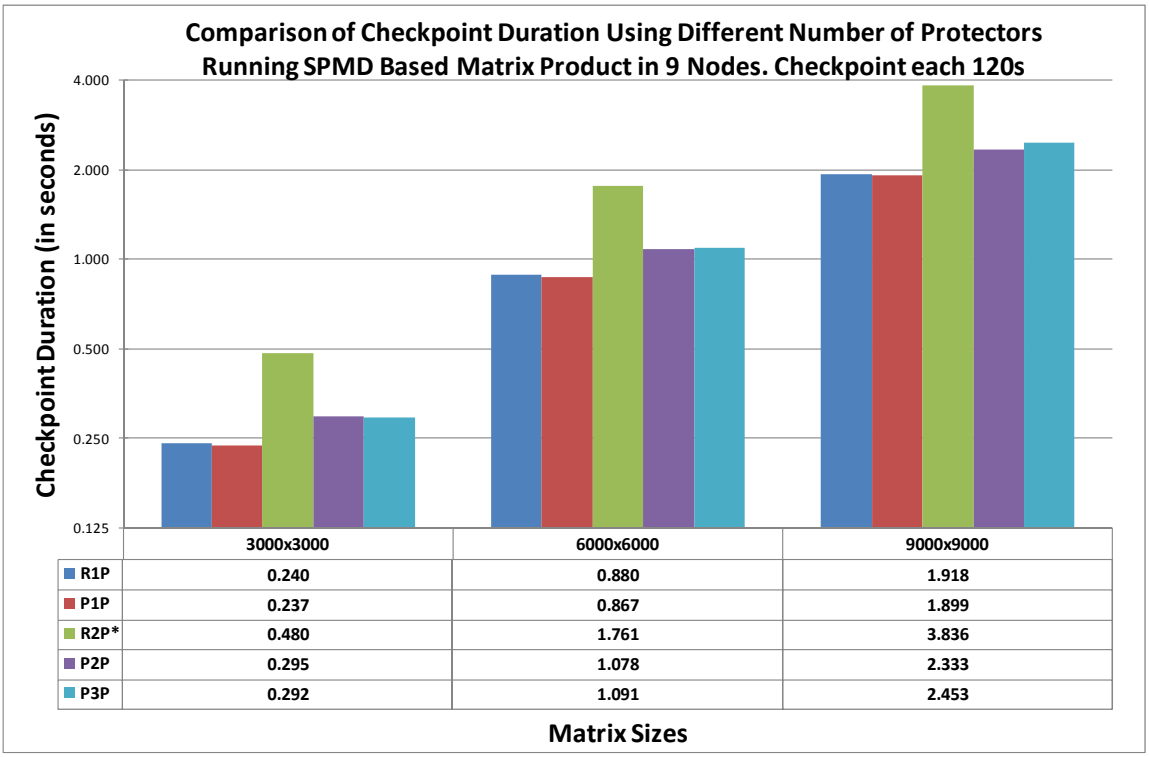


Figure 5-11. Checkpoint comparison using a SPMD matrix product program using different number of protectors and checkpoint sizes (according to the matrix size: 3000x3000, 6000x600 and 9000x9000).

Evaluating the execution times

Similarly, it was performed a set of experiments analysing the benefits of the solution in the execution time of set of scenarios presented in the checkpointing evaluation. In this case, it also performed executions without fault tolerance for each matrix size (W/o FT) in order to have a comparison basis. The computation and communication were repeated ten times in order to achieve enough time for checkpointing.

The results of these executions are presented in the chart of Figure 5-12. Due to the reduction of the message delivery latency and low overhead the redundant data replication, the execution times for a same matrix size are very similar despite the number of protectors, reaching at maximum 2.80% of overhead comparing the use of 3 protectors with using only 1 protector multiplying 9000x9000 matrixes. In the best case (3000x3000 matrixes), the pipe-

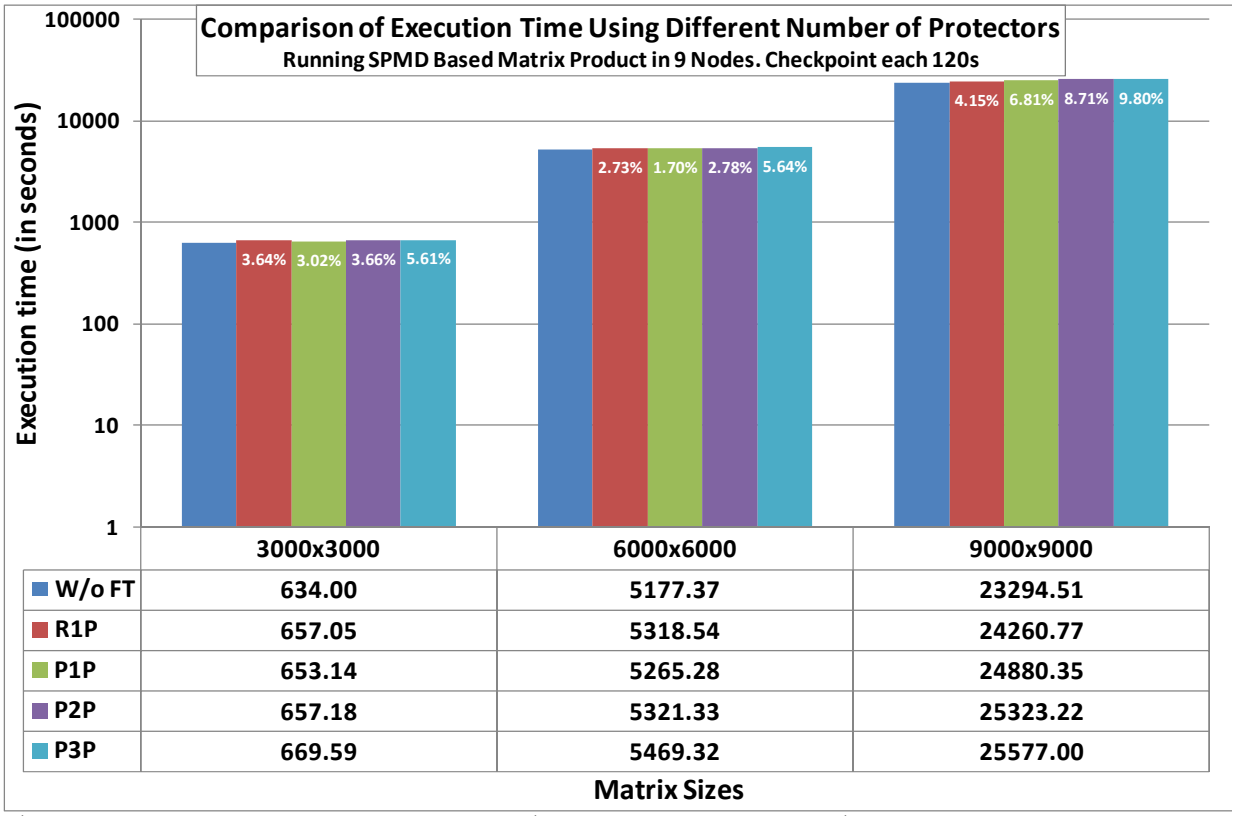


Figure 5-12. Execution time comparison using a SPMD matrix product program using different number of protectors and checkpoint sizes (according to the matrix size: 3000x3000, 6000x600 and 9000x9000).

lined data replication generated an overhead of only 0.68% comparing using 2 protectors with using 1 protector and 2.58% when using 3 protectors. In the 9000x9000 matrix executions, messages and checkpoints are larger, which leads to more network concurrency as explained before. This happening explains the worst performance of the P1P approach in comparison to the R1P.

It was also executed the same experiment with the Travelling Salesman program but increasing the number of protectors up to 3 protectors, measuring the execution time. As seen before such a program suffers low overhead of the fault tolerance solution due the small amount of communications, and the intrinsic load balancing. Therefore the results using different number of protectors are very similar for each situation as can be seen in the chart depicted in Figure 5-18. The increase in the number of protectors has generated minimal over-

head in the execution time, representing at maximum 0.344%. The difference of using 2 or 3 protectors is also unnoticeable.

These execution time experiments shows that the pipelined data replication increase the degree of availability by having many copies of the redundant data and it imposes a low overhead for the execution time of these applications. Therefore, the computer cluster performability has been improved for these applications, which may allow the execution of mission-critical applications in RADIC despite the occurrence of concurrent correlated faults.

The performability values presented in TABLE 5-4 and TABLE 5-5 are referred to the two previous experiments respectively. As the redundant data replication targets mission-critical applications, which have a high risk, i.e., one interruption during the execution may cause serious damage, the unavailability (*uw*) factor have the same value of the performance (*pw*) factor in the performability equation (=1). As can be seen, in all cases using 2 protectors with pipeline, the performability was increased including comparing with the pipeline.

5.3.2.1. Evaluating according to the fault moment

This experiment series evaluate the behavior of the applications according with the moment of the fault when using or not dynamic redundancy

TABLE 5-4: Performability behavior of a SPMD matrix product program using different number of protectors and checkpoint sizes (according to the matrix size: 3000x3000, 6000x600 and 9000x9000) using or not pipelined replication.

	3000x3000	6000x6000	9000x9000
Unavailability wo/Correlated Faults	**0.00171%**	**0.00171%**	**0.00171%**
Unavailability w/Correlated Fault of 2 nodes	**0.00178%**	**0.00178%**	**0.00178%**
Throughput Regular	**13697.49**	**6768.78**	**3338.72**
Performability Regular	**7677.03**	**3793.69**	**1871.25**
Throughput P1P	**13779.68**	**6837.25**	**3255.58**
Performability P1P	**7723.10**	**3832.07**	**1824.66**
throughput P2P	**13694.87**	**6765.23**	**3198.65**
Performability P2P	**7997.94**	**3950.96**	**1868.04**

TABLE 5-5: Performability behavior of the travelling salesman program with 15 cities, comparing different number of protectors using or not pipelined replication. Throughput in Million of routes/s

	8 Nodes	16 Nodes	32 Nodes
Unavailability wo/Correlated Faults	0.00152%	0.00304%	0.00609%
Unavailability w/Correlated Fault of 2 nodes	0.00159%	0.00317%	0.00634%
Throughput Regular	67.78	136.82	293.13
Performability Regular	42.74	43.14	46.21
Throughput P1P	67.75	136.81	293.06
Performability P1P	42.72	43.13	46.20
throughput P2P	67.70	136.78	292.96
Performability P2P	44.48	44.93	48.12

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

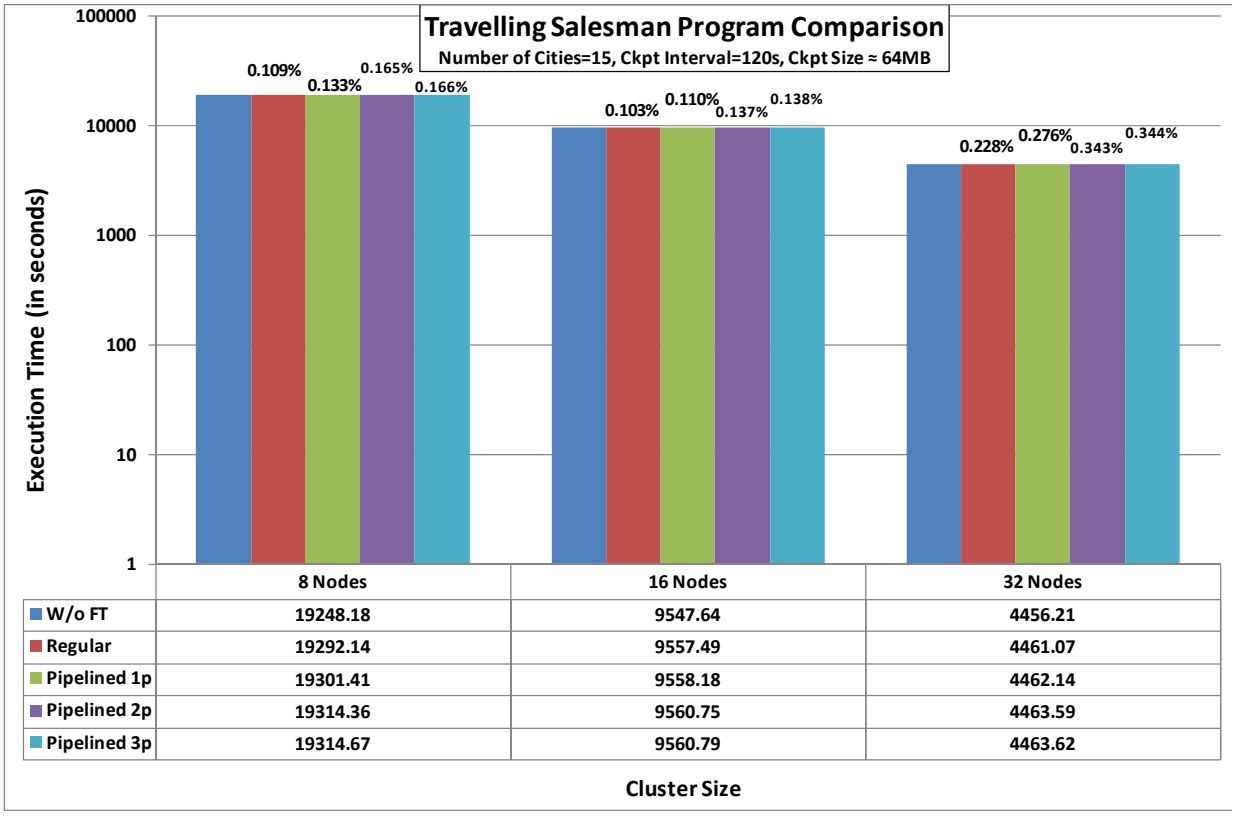


Figure 5-13. Execution time comparison using the travelling salesman program with 15 cities, comparing between without fault tolerance and different number of protectors using or not pipelined replication

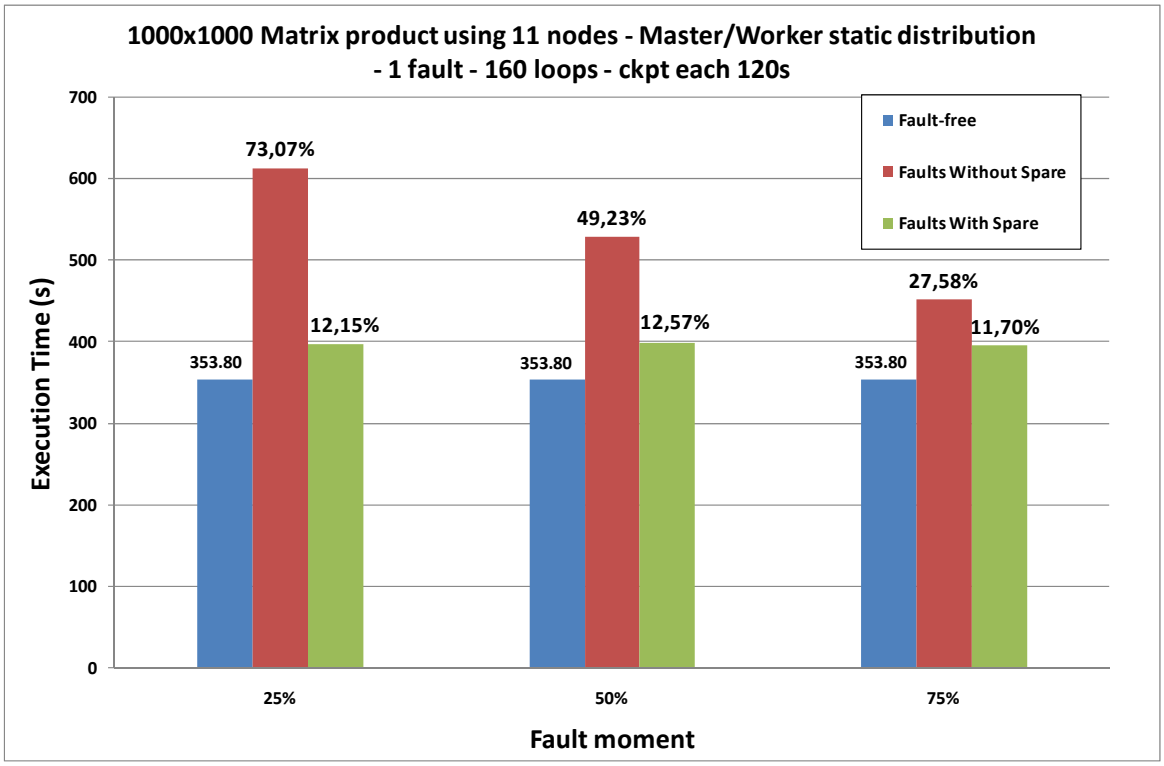


Figure 5-14: Results of matrix product using a master-work static distributed program injecting faults in different moments.

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

The Figure 5-14 contains a chart showing the results with the master-worker approach. This chart shows that the overhead caused by a recovery without spare (the red middle column in each fault moment) versus using spare (the green right column in each fault moment) with one fault occurring in different moments. The overhead not using spares shows

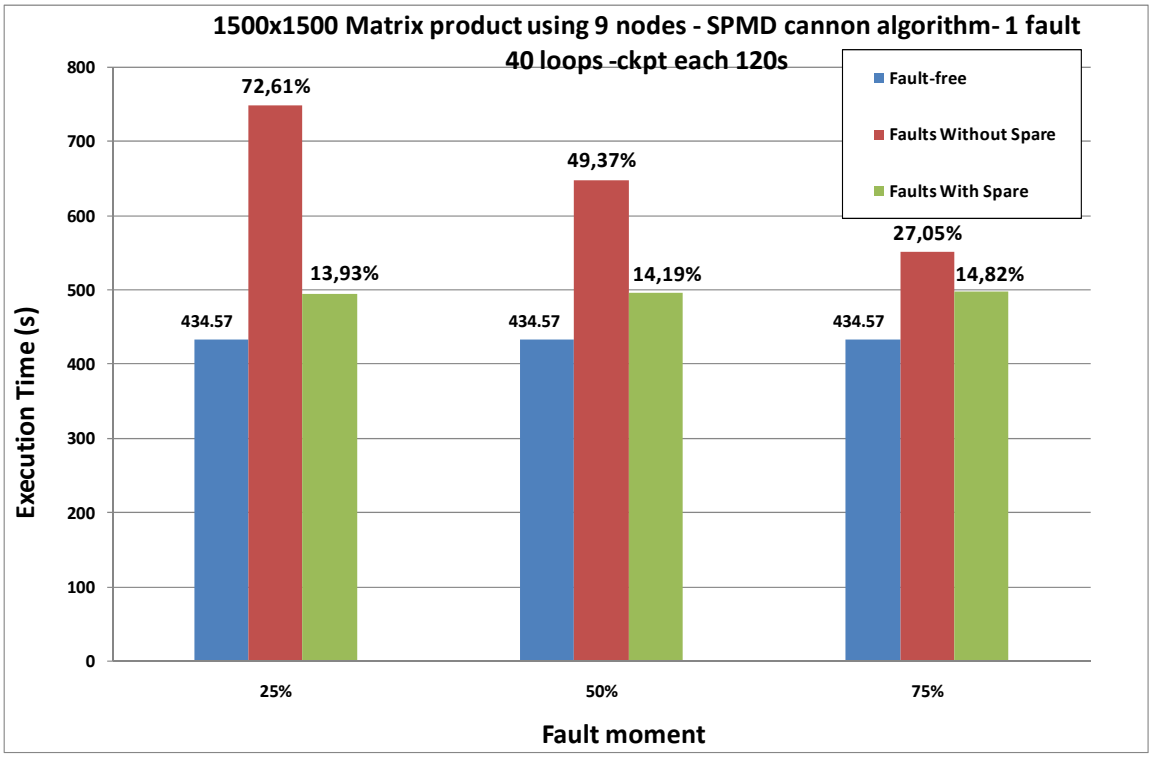


Figure 5-15: Results of matrix product using a SPMD program based in the cannon algorithm

itself inversely proportional to the moment when the fault occurs, generating greater overheads (reaching 73.07% in the worst case analyzed) in premature fault case, while using spare, the overhead keeps constantly and low despite the moment of the fault.

The Figure 5-15 shows a result chart with the SPMD program. An analogous behavior is perceived in relation with the overhead caused by not using spare nodes. The overhead caused by the spare nodes usage is slightly greater than the static distribution approach. This increment is due to the high coupling level in the SPMD approach, the time spent in the recovery affects directly the communications with the neighbors' processes and this delay continues propagating by the others process of the application, while the recovery in the master-worker approach only affects the failed worker.

The performability results of the experiments above are in TABLE 5-6 and TABLE 5-7. These numbers show the correlativeness between the performability and the fault mo-

TABLE 5-6: Performability results of matrix product using a master-work static distributed program injecting faults in different moments and using 11 nodes plus one spare

		Fault-free	25%	50%	75%
Without Spare	Unavailability	0.00209%	0.00209%	0.00209%	0.00209%
	Throughput	2826.46	1633.06	1894.13	2215.37
	Performability	2438.35	1408.82	1634.04	1911.17
With Spare	Unavailability	0.00228%	0.00214%	0.00212%	0.00211%
	Throughput	2826.46	2520.16	2510.67	2530.36
	Performability	2396.29	2164.37	2161.04	2179.22

ment, where as soon the fault occurs as lower will be the performability. In these experiments the unavailability is proportionally calculated according the number of nodes in each case, i.e., with a fault at 25%, during 25% of time it was n+1 nodes running (cluster size plus spare node) and during 75% it was only n nodes running.

5.3.2.2. Evaluating according with the number of nodes

In these experiments, the behavior of the fault recovery in different cluster sizes was evaluated. The current experiments suggest that the RADIC improvements presented in this work do not affect the scalability of a program.

It was run two approaches for a master-work matrix product: using a static distribution and using a dynamic distribution of matrix blocks. In both cases it was performed a product between two 1000 X 1000 matrixes. It was executed the program with four, eight and eleven nodes, with faults injected always at 25% of the execution time, approximately. The

TABLE 5-7: Performability results of matrix product using a SPMD program based in the cannon algorithm injecting faults in different moments and using nine nodes plus one spare.

		Fault-free	25%	50%	75%
Without Spare	Unavailability	0.00171%	0.00171%	0.00171%	0.00171%
	Throughput	5177.53	2999.46	3466.18	4074.87
	Performability	4649.51	2693.57	3112.69	3659.30
With Spare	Unavailability	0.00190%	0.00176%	0.00181%	0.00185%
	Throughput	5177.53	4544.26	4533.91	4509.11
	Performability	4552.56	4058.52	4027.74	3984.95

execution time was measured when using or not the spare nodes and comparing with a fault free execution time.

Figure 5-16 shows the results of executions with a dynamic load balancing approach. The load balancing can mitigate the side-effects of the RADIC regular recovery, and the spare nodes use is almost equal than not using it, being worse in the smallest cluster because the time spent for the spare use is greater than normal recovery, therefore the system remains more time with one process less (in a four nodes cluster it means 25%). Indeed, the processes in the overloaded node start to perform fewer tasks than other nodes, and their workload is distributed among the cluster, almost not affecting the execution time. As the cost of recovery using spare nodes may slightly greater, it should be better to use the basic protection level for these cases.

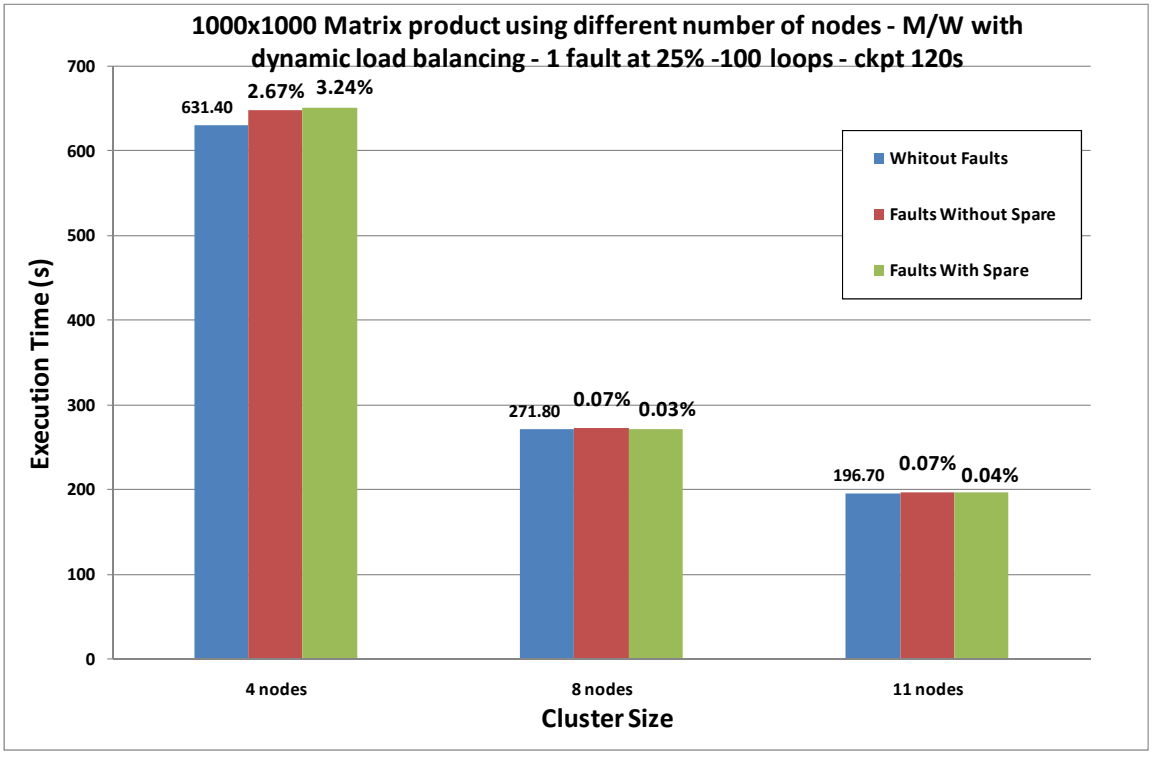


Figure 5-16: Results of matrix product using a master-worker program with dynamic load balancing running in different cluster sizes

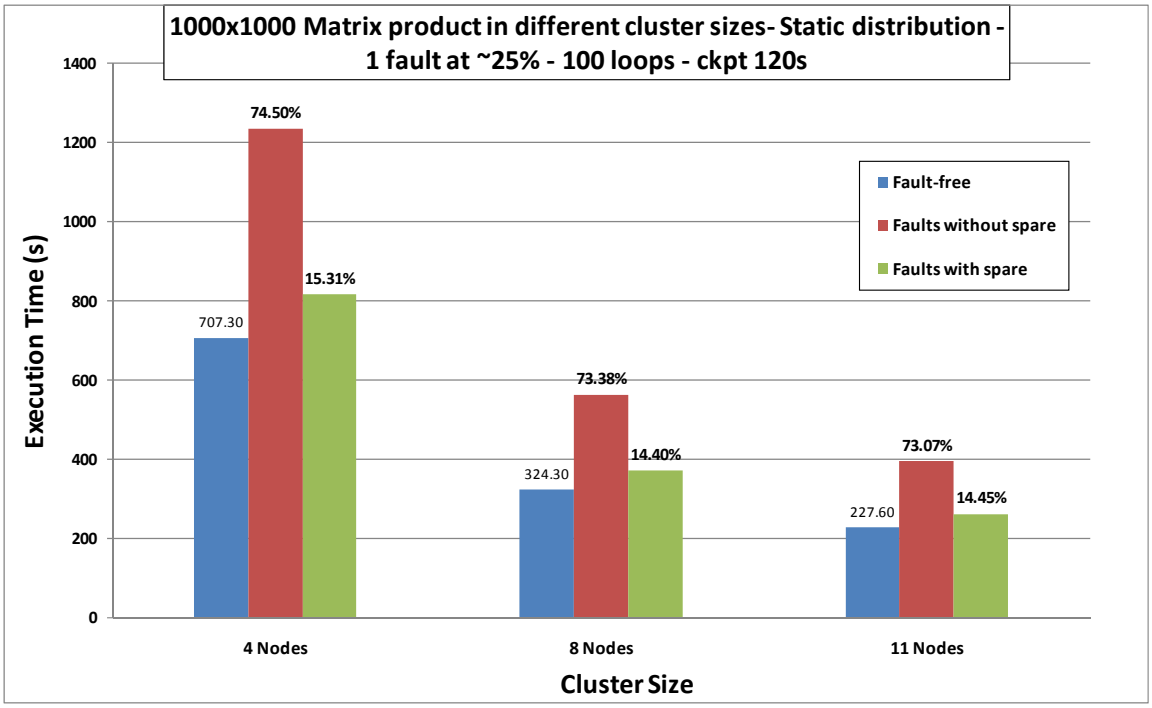


Figure 5-17: Results of matrix product using a master-worker program with static load distribution running in different cluster sizes

Figure 5-17 shows the benefits of using spare nodes using a static load distribution, the node that hosts the recovered process suffers a strong degradation, high affecting the overall execution time independently of the size of the cluster. By other side, using the spare nodes approaches, the overall impact in the execution time is low and stable, also independently of the number of nodes.

The numerical results of performability analysis for each case above are in TABLE 5-8 and TABLE 5-9. Those numbers show that in case of a dynamic load balancing, the use of spare nodes does not result in better performability in spite of better performance values for 8 and 11 nodes. This happens because the existence of one spare node affects the system unavailability increasing such value. In the other hand, using a static load balancing, the spare node use allows the improvement of the performability comparing with not using. It also should be noted that for the four nodes case, the unavailability is below the target, therefore the performability values are equal to performance.

TABLE 5-8: Performability results of matrix product using a master-worker program with dynamic load balancing running in different cluster sizes

	4 Nodes	8 Nodes	11 Nodes
Unavailability wo/Spare	**0.00076%**	**0.00152%**	**0.00209%**
Throughput Fault-free	**1583.78**	**3679.18**	**5083.88**
Performability Fault-free	**1583.78**	**3382.71**	**4385.81**
Throughput w/Fault wo/Spare	**1542.59**	**3653.60**	**5048.54**
Performability w/Fault wo/Spare	**1542.59**	**3359.20**	**4355.32**
Unavailability w/Spare	**0.00081%**	**0.00157%**	**0.00214%**
Throughput w/Spare	**1534.08**	**3668.17**	**5068.68**
Performability w/spare	**1534.08**	**3351.90**	**4353.08**

5.3.2.3. Non-Stop service experiments

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

Figure 5-18 shows the result chart of this experiment. In this experiment, it is per-

TABLE 5-9: Performability results of matrix product using a master-worker program with static load distribution running in different cluster sizes

	4 Nodes	8 Nodes	11 Nodes
Unavailability wo/Spare	**0.00076%**	**0.00152%**	**0.00209%**
Throughput Fault-free	**1413.83**	**3083.56**	**4393.67**
Performability Fault-free	**1413.83**	**2835.09**	**3790.37**
Throughput w/Fault wo/Spare	**810.20**	**1778.50**	**2538.67**
Performability w/Fault wo/Spare	**810.20**	**1635.19**	**2190.08**
Unavailability w/Spare	**0.00081%**	**0.00157%**	**0.00214%**
Throughput w/Spare	**1226.11**	**2695.42**	**3838.95**
Performability w/spare	**1226.11**	**2463.02**	**3296.97**

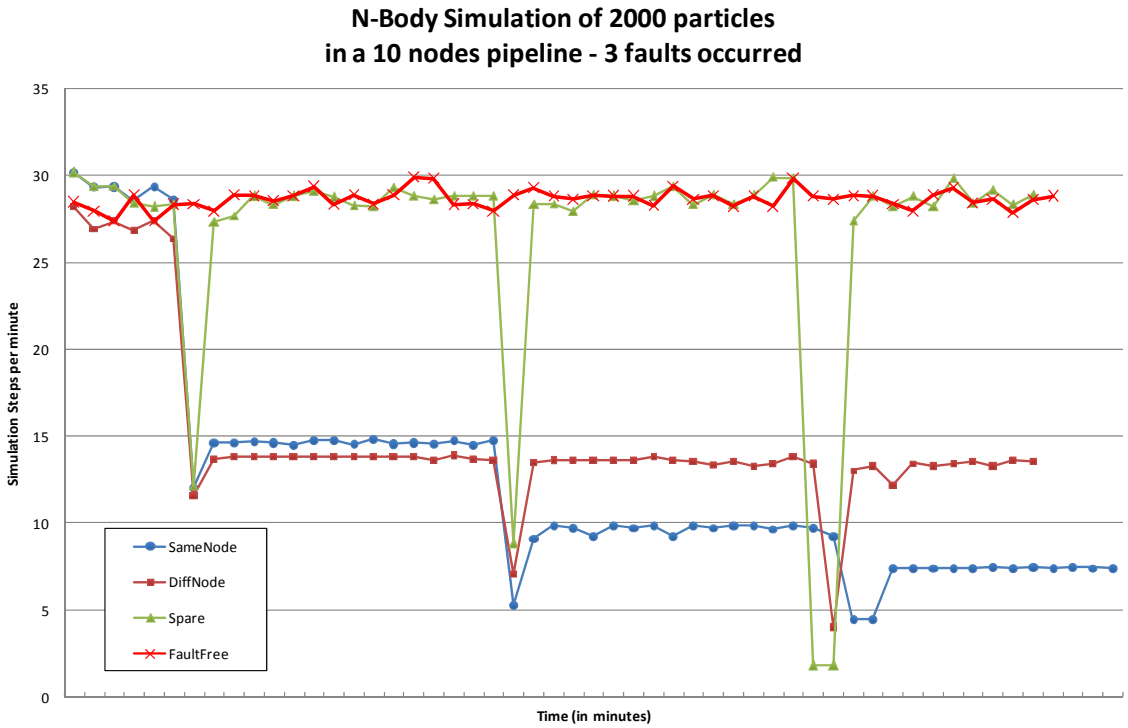


Figure 5-18: Results of an N-Body program running continuously after three faults in different situations.

ceived the influence of the application type over the post-recovery execution. When the three faults are recovered in different nodes, the application's throughput suffers an initial degradation, but in the subsequent faults, just changes a little. This behavior occurs because the pipeline arrangement: the degradation of the node containing the second recovered is masked by the delay caused by the first recovered process node. This assumption is confirmed when all faults processes are recovered in the same node, it is possible to perceive a degradation of the throughput after each failure. When executing with spare nodes presence, the system backs to the original simulation step rate after quick throughput degradation. It is also possible to see that the penalization imposed by the recovery process using spare is greater than the regular RADIC process, but this loss is quickly compensated by the throughput restoring in the remaining execution.

TABLE 5-10 contains the results of same performability analysis performed in Chapter 4 but now including the values related with the spare node use. The values of recovering in different nodes are not explicitly mentioned but as seen in the chart they are similar with the values when only one fault occurs. The performability value of the spare nodes use shows that the flexible dynamic redundancy was effective improving such a value despite the small increase on the unavailability due the additional spare nodes. It should be noted that such unavailability value would be greater if not using the dynamic insertion of spare nodes, i.e. would be necessary 3 spare nodes in order to tolerate 3 faults, but in this case it was needed only two, being one fixed and re-inserted.

TABLE 5-10: Performability behavior of an N-Body simulation after one, two and three faults recovered in the same node.

	Fault-free	After 1 Fault	After 2 Faults	After 3 Faults	With Spare node
Throughput (simulation steps/min)	**28.65**	**14.51**	**9.43**	**7.22**	**26.87**
Unavailability	**0.00190%**	**0.00171%**	**0.00152%**	**0.00133%**	**0.00228%**
Performability	**25.19222**	**13.02734**	**8.66778**	**6.81544**	**22.78464**

Chapter 6
Conclusions

The use of computer clusters in HPC area continues to increase. As the demand for computing power grows, more computing nodes are aggregated. Nowadays, such a number easily reaches more than 1000 nodes. Besides the quest for computing power, availability has also been a major concern. As the number of nodes increases, the probability of faults in some of these nodes rises at the same pace. In systems following a fail-stop semantic, a fault in a node leads to a failure and, consequently, an interruption of applications using that node. In this scenario, fault tolerance plays an important role providing high availability.

Performance and availability evaluation has been well studied in the past years. Lately, the concept of performability has enabled evaluating these two metrics conjunctly, allowing a global, accurate and real evaluation of the aforementioned systems.

The performability concept raises some questions when using a fault tolerant system: How do fault tolerant activities influence system performability? What are the root causes of overheads caused by a fault tolerant system which increases the system availability? How does a recovery process affect system performability? How do degraded systems affect system performability? After investigating these questions it was achieved a better understanding of how fault tolerant solutions influence system performability and how it can be possible to make improvements, even considering restrictions such as execution time or throughput with or without faults. A fault tolerance solution tolerates a certain number of faults, but it

can degrade performance (execution time, throughput). Therefore, how much is acceptable such a degradation?

The RADIC architecture was used as a study case. The *modus operandi* of its components was analyzed and the implications of its operation on system performability studied.

This work identified the root causes of performance overhead using different degrees of availability such as message logging and the data replication over *N*-protectors, and the root causes of performance degradation caused by the system configuration change after recovering from a fault using the RADIC architecture. All root causes directly influence system performability.

Taking into consideration these results, solutions based on the RADIC fault tolerance architecture were proposed to reduce the performance overhead in fault-free executions and avoid or fix performance degradation in the presence of faults, resulting in an improvement of the computer cluster's performability.

A technique for performing receiver-based pessimistic message logging that reduces message delivery latency was presented. This technique works by dividing the sending message into small pieces and establishing a pipeline between the observer at the message receiver and its protector, rather than performing a store-and-forward of each message such as the traditional message log approach. Such a technique reaches an overhead reduction of 80.48%. The influence of the piece size choice in this technique was analyzed.

The pipeline idea was used to increase the availability provided by RADIC with low overhead, reducing the risk for mission-critical applications. Such a solution is based on performing a pipelined data replication of checkpoints and message logs (redundant data) distri-

buted over a number of protectors, according to a desired degree of availability. The pipeline approach reduces the time required to perform the checkpoint data replication by up to 39%. It does by parallelizing the activity and allowing the tolerance of concurrent faults in correlated nodes imposing a maximum of 29.16% of overhead in comparison with not performing the replication. The number of protectors growth with low overhead allowed the system deal with mission-critical applications because of the better performability. This work was accepted to be presented in (SANTOS, G. et al., 2009)

On avoiding performance degradation, the design of a solution that avoids such behavior named resilient protection level was proposed. The implementation of this new feature, represented by the use of spare nodes, did not affect the RADIC characteristics of transparency, decentralization, flexibility and scalability. Hence, a transparent management of spare nodes was designed, which was able to request and use spare nodes without need for centralized information. The RADIC resilient protection level can dynamically insert new spare nodes during the application execution.

The RADIC resilient protection level also allows the replacement of faulty nodes transparently to the application. Therefore, if the system is degraded because of faults, it is possible to re-incorporate the failed nodes after fixing them *without* stopping the application. Such a mechanism reduces the execution time overhead in the presence of faults by five times in high coupled and static load balanced applications. However, in dynamic load balancing applications, the performability did not improve using spare nodes since the application redistributes the extra load over the remaining nodes.

The new features presented can be also applied to perform preventive maintenance tasks by injecting faults into specific nodes forcing the processes running on them to migrate

to a recently inserted spare node. These abilities represent the flexibility of RADIC's resilient protection level, beyond keeping RADIC's original structural flexibility.

On one side, the efficiency of the pipelining approach depends on the size of each communication. In large communications, the latency overhead was reduced by approximately four times, while in small communications it was perceived slightly worse than in the latency overhead with traditional logging. This behavior is because of the activities needed to perform a pipeline, such as control structures, and extra headers. A simple workaround would be to decide up front when to pipeline according to the size of communication. Furthermore, performing the data replication over many protectors represented a low overhead in comparison with the traditional approach. These efforts resulted in program execution times with low overheads despite the degree of availability chosen (number of protectors per observer), resulting in an improvement of system performability.

On the other side, experiments confirmed that the side effects caused by some recovery approaches depends on factors such as application characteristics, i.e. message pattern and parallel paradigm applied (pipeline, Master-Work or SPMD) and where the process is recovered. The fault recovery may affect the overall performance of the system, and the generated performance degradation can vary according to where the process recovers and the parallel paradigm applied. Another perceived relation regards application coupling. Applications with high coupling levels between the computing nodes tend to suffer more intensively with system configuration changes caused by the recovery process.

Moreover, it is possible to conclude that the use of a flexible redundancy scheme is an effective approach to avoiding the effects of system configuration changes. The presented solution has shown to be effective even in faults close to the application finishing. The RAD-

IC resilient protection level also shows an overhead caused by the recovery process, but the cost depends on factors such as fault moment, checkpointing interval, and process state size. Experimental results have shown execution times and throughput values very near to a failure-free execution. The initial idea of the resilient protection level was presented by Santos et al. (SANTOS, G. et al., 2006), with a complete evaluation of this solution (SANTOS, G. et al., 2008). The use of the resilient protection level for a stopless preventive maintenance was also presented by Santos et al. (SANTOS, G. et al., 2008)

These findings enhance the knowledge on fault tolerance issues and their influence on system performability, such as the effects of uncoordinated checkpointing and logging in a fully distributed fault tolerance solution, the relationship between application characteristics and behavior and the influence of a parallel paradigm in recovered applications.

6.1. Open lines

After much work, many open lines were found during the path to here. These open lines may represent future researches to be performed, which can expand the knowledge obtained from the performability study of RADIC.

It would be interesting to assess the performability of RADIC in large clusters and with different kind of applications. This study would provide a real knowledge about RADIC scalability using the *N*-protectors data replication or the new spare nodes information spreading. Due to the physical difficulties accessing these machines, a RADIC simulator would be necessary alongside a complete message-passing implementation of RADIC as presented by Fialho et al. (FIALHO, L. et al., 2009), and including the new contributions presented here.

The ideal number of spare nodes and its ideal allocation through the cluster remain undiscovered subjects. Further research might investigate how it is possible to achieve better results by allocating spare nodes according to requirements such as acceptable degradation level according to a performability index, or memory limits of a node.

New technologies are arriving each day. A permanent task will be to study how to adapt and use RADIC with the new trends of the high computing area. This includes questions such as how its performability behaves using *multicore* computers, how the characteristics of this architecture can be exploited and what the influence of using many protectors per observer in the performability of an architecture having many processes (and observers) per node is. Other questions include whether it is possible to have spare cores instead of only spare nodes, and if so, how this configuration influences performability.

Fault tolerance systems generally are complex systems. RADIC with its protection levels and configurations is no exception. In this work the RADIC performability was evaluated based on measurements. Considerably more work must be undertaken to generate a performability analytical model of RADIC including its characteristics. Such a study will be crucial helping better understand the architecture and providing the tools to improve the RADIC operation by tuning parameters such as checkpoint interval or protectors' mapping in order to achieve better performability. Furthermore, this model may be applied to foresee performability under some parameters.

A study about possible election policies to be used during the pipelined recovery process, during the node replacement feature or when an application must use a spare node or not will be useful for determining the ideal behavior for these situations, considering factors

such as load balance or time to recover. Performability indexes are useful for supporting such decisions.

The maintenance feature of resilient level remains a rarely explored subject. Additional research might address integrating this feature with a fault prediction scheme, which will allow RADIC to perform a proactive fault tolerance, thus, avoiding faults before they happen and further improving cluster performability.

Analysis of the communication buffers effects in the pipelined logging performance deserves to be investigated further, as does the pipelined logging behavior in larger clusters using different networks such as infiniband or myrinet, and different network topologies such as fat-tree, meshes and torus. Experimental results suggest that using a dynamic piece size according to the message size may produce better results, therefore, this subject must be addressed in further works.

Autonomic computing systems are a new trend. Based on the human autonomic system, this new trend establishes a new group of systems with the abilities of self-healing, self-configuring, self-protecting and self-optimizing. RADIC already provides the self-healing ability, while the RADIC protection level implements the self-configuring capacity. Hence new research might perform steps towards an autonomic fault tolerant system implementing the self-protecting and self-optimizing features.

References

AGARWAL, S., R. GARG, M.S. GUPTA, and J. E. MOREIRA. 2004. Adaptive incremental checkpointing for massively parallel systems. *In*: *ICS '04: Proceedings of the 18th annual international conference*. ACM, pp.277-286.

AGBARIA, A. and R. FRIEDMAN. 1999. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *In*: *HPDC '99: Proceedings of the The Eighth IEEE International Symposium*. IEEE Computer Society, p.31.

ALVISI, L., E. ELNOZAHY, S. RAO et al. 1999. An analysis of communication induced checkpointing. *In*: *Proc. Digest of Papers Fault-Tolerant Computing Twenty-Ninth Annual International Symposium on*. IEEE Computer Society, pp.242-249.

ALVISI, L. and K. MARZULLO. 1998. Message logging: pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*. **24**, pp.149-159.

BIRMAN, K.P. 2005. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer.

BOSILCA, G., A. BOUTEILLER, F. CAPPELLO et al. 2002. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. *In*: *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Baltimore: IEEE Computer Society, pp.29-29.

BOUTEILLER, Aurélien, Franck CAPPELLO, Thomas HERAULT et al. 2003. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. *In*: *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, p.25.

BOUTEILLER, A., T. HERAULT, G. KRAWEZIK et al. 2006. MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI. *International Journal of High Performance Computing Applications*. **20**, p.319.

BOUTEILLER, Aurélien, Pierre LEMARINIER, Geráude KRAWEZIK, and Franck CAPPELLO. 2003. Coordinated checkpoint versus message log for fault tolerant MPI. *In*: *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference*. IEEE; IEEE Computer Society.

BURNS, G., R. DAOUD, and J. VAIGL. 1994. LAM: An Open Cluster Environment for MPI. *In*: *Proceedings of Supercomputing Symposium '94.*, pp.379-386.

CHAKRAVORTY, S., C. L. MENDES, and L. V. KALÉ. 2006. Proactive Fault Tolerance in MPI Applications Via Task Migration. *In*: *Proceedings of International Conference on High Performance Computing*. Springer, pp.485-496.

CHANDY, K. M. and L. LAMPORT. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans.Comput.Syst.* **3**, pp.63-75.

CHTEPEN, M., F.H.A. CLAEYS, B. DHOEDT et al. 2009. Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids. *Parallel and Distributed Systems, IEEE Transactions on*. **20**, pp.180-190.

DALY, J. T. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*. **22**, pp.303-312.

DUARTE, A. 2007. *RADIC: A Powerful Fault Tolerant Architecture*. PhD thesis, Universitat Autònoma de Barcelona.

DUARTE, A., D. REXACHS, and E. LUQUE. 2006. Increasing the cluster availability using RADIC. *In*: *Proceedings of 2006 IEEE International Conference on Cluster Computing*. IEEE, pp.1-8.

DUARTE, A., D. REXACHS, and E. LUQUE. 2006. An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI. *In*: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*. Springer Berlin / Heidelberg, pp.150-157.

ELNOZAHY, E. N., Lorenzo ALVISI, Yi-Min WANG, and David B. JOHNSON. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*. **34**, pp.375-408.

ELNOZAHY, E. N. and J. S. PLANK. 2004. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE Trans.Dependable Secur.Comput.* **1**, pp.97-108.

ELNOZAHY, E. N. and W. ZWAENEPOEL. 1992. Manetho: Transparent Roll Back-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*. **41**, pp.526-531.

ELNOZAHY, E.N. and W. ZWAENEPOEL. 1994. On the use and implementation of message logging. *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on.*, pp.298-307.

EUSGELD, I. and F. FREILING. 2008. Introduction to Dependability Metrics. *In*: *Dependability Metrics*, Springer Berlin / Heidelberg, pp.1-4.

EUSGELD, I., J. HAPPE, P. LIMBOURG et al. 2008. Performability. *In*: *Dependability Metrics*, Springer Berlin / Heidelberg, pp.245-254.

FAGG, G. E. and J. J. DONGARRA. 2000. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. *In*: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting, Balatonfüred, Hungary, September 2000, Proceedings*. Balatonfüred: Springer Berlin / Heidelberg, pp.346-353.

FIALHO, L., A. DUARTE, G. SANTOS et al. 2009. Challenges and Issues of the Integration of RADIC into Open MPI. *In*: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Esboo, Finland, September 7-10, 2009. Proceedings*. Springer Berlin / Heidelberg, pp.73-83.

GAO, W., M. CHEN, and T. NANYA. 2005. A Faster Checkpointing and Recovery Algorithm with a Hierarchical Storage Approach. *In*: *High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on*. IEEE, pp.398-402.

GEIST, A. and C. ENGELMANN. 2002. *Development of naturally fault tolerant algorithms for computing on 100,000 processors*. [online]. [Accessed February 2008]. Available form World Wide Web: <http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf>

GRAY, J. and D. P. SIEWIOREK. 1991. High-availability computer systems. *Computer*. **24**, pp.39-48.

GROPP, W., E. LUSK, and A. SKJELLUM. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press.

GUTIN, G. and PUNNEN, A. P. (eds). 2007. *The traveling salesman problem and its variations*. Springer.

HAVERKORT, B. R., R. MARIE, G. RUBINO, and K. S. TRIVEDI. 2001. *Performability Modelling : Techniques and Tools*. John Wiley \& Sons, Inc.

HELARY, J. M., A. MOSTEFAOUI, R. H. B., and M. RAYNAL. 1997. Preventing useless checkpoints in distributed computations. *In*: *Proc. Sixteenth Symposium on Reliable Distributed Systems*. IEEE Computer Society, pp.183-190.

HUANG, Y. and Y. WANG. 1995. Why optimistic message logging has not been used in telecommunications systems. *In*: *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*. IEEE Computer Society, pp.459-463.

IZOSIMOV, V., P. POP, P. ELES, and Z. PENG. 2006. Synthesis of Fault-Tolerant Embedded Systems with Checkpointing and Replication. *In*: *Proceedings of the Third IEEE International Workshop on Electronic Design, Test and Applications*. IEEE Computer Society, pp.440-447.

JALOTE, P. 1994. *Fault Tolerance in Distributed Systems*. P T R Prentice Hall.

KALAISELVI, S. and V. RAJARAMAN. 2000. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*. **25**, pp.489-510.

KONDO, M., T. HAYASHIDA, M. IMAI et al. 2003. Evaluation of Checkpointing Mechanism on SCore Cluster System. *IEICE Transactions on Information and Systems*. **86**, pp.2553-2562.

KOREN, I. and C. M. KRISHNA. 2007. *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc.

KOZIOLEK, H. 2008. Introduction to Performance Metrics. *In*: *Dependability Metrics*, Springer Berlin / Heidelberg, pp.199-203.

LIANG, Y., Y. ZHANG, A. SIVASUBRAMANIAM et al. 2005. Filtering failure logs for a BlueGene/L prototype. *In*: *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE Computer Society, pp.476-485.

LIE, C. H., C. L. HWANG, and F. A. TILLMAN. 1977. Availability of Maintained Systems: A State-of-the-Art Survey. *IIE Transactions*. **9**, pp.247-259.

LI, Y. and Z. LAN. 2006. Exploit failure prediction for adaptive fault-tolerance in cluster computing. *In*: *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06*. IEEE Computer Society, pp.531-538.

LIU, Y., C. B. LEANGSUKSUN, H. SONG, and S. L. SCOTT. 2005. Reliability-aware Checkpoint/Restart Scheme: A Performability Trade-off. *In*: *Cluster Computing, 2005. IEEE International*. IEEE Computer Society, pp.1-8.

MALONEY, A. and A. GOSCINSKI. 2009. A survey and review of the current state of rollback-recovery for cluster systems. *Concurrency and Computation: Practice and Experience*. **9999**, p.n/a.

MALONEY, A. and A. GOSCINSKI. 2009. A survey and review of the current state of rollback-recovery for cluster systems. *Concurrency and Computation: Practice and Experience*. **9999**, p.n/a.

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

MEYER, J. F. 1995. Performability evaluation: where it is and what lies ahead. *In*: *Computer Performance and Dependability Symposium, 1995. Proceedings., International*. IEEE Computer Society, pp.334-343.

MEYER, J.F. 1992. Performability: a retrospective and some pointers to the future. *Performance Evaluation*. **14**, pp.139-156.

NAGARAJA, K., G. GAMA, R. BIANCHINI et al. 2005. Quantifying the Performability of Cluster-Based Services. *Parallel and Distributed Systems, IEEE Transactions on*. **16**, pp.456-467.

NAGARAJAN, A. B., F. MUELLER, C. ENGELMANN, and S. L. SCOTT. 2007. Proactive fault tolerance for HPC with Xen virtualization. *In*: *ICS '07: Proceedings of the 21st annual international conference*. ACM, pp.23-32.

NAM, H., J. KIM, S. HONG, and S. LEE. 1997. Probabilistic checkpointing. *In*: *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*. IEEE Computer Society, pp.48-57.

NI, L. M. and P. K. MCKINLEY. 1993. A Survey of Wormhole Routing Techniques in Direct Networks. *Computer*. **26**, pp.62-76.

OLINER, A.J., R.K. SAHOO, J.E. MOREIRA, and M. GUPTA. 2005. Performance implications of periodic checkpointing on large-scale cluster systems. *In*: *Proceedings of the*

19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18 - Volume 19. IEEE Computer Society, pp.8 pp.-.

PIEDAD, F. and M. HAWKINS. 2001. *High Availability: Design, Techniques, and Processes*. Prentice Hall PTR.

PLANK, J. S., Kai LI, and M. A. PUENING. 1998. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*. **9**, pp.972-986.

PLANK, J. S. and M. G. THOMASON. 2001. Processor allocation and checkpoint interval selection in cluster computing systems. *J.Parallel Distrib.Comput.* **61**, pp.1570-1590.

RAO, S., L. ALVISI, and H. M. VIN. 2000. The cost of recovery in message logging protocols. *IEEE Transactions on Knowledge and Data Engineering*. **12**, pp.160-173.

RAO, S., L. ALVISI, and H. M. VIN. 1999. *Egida: an extensible toolkit for low-overhead fault-tolerance*. Austin, TX, USA: University of Texas at Austin.

RODRÍGUEZ, G. 2008. *Compiler-assisted checkpointing of message-passing applications in heterogeneous environments*. PhD Thesis, A Coruña.

SABETTA, A. and H. KOZIOLEK. 2008. Measuring Performance Metrics: Techniques and Tools. *In*: *Dependability Metrics*, Springer Berlin / Heidelberg, pp.226-232.

SAHOO, R. K., A. J. OLINER, I. RISH et al. 2003. Critical event prediction for proactive management in large-scale computer clusters. *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining.*, pp.426-435.

SANCHO, J. C., F. PETRINI, G. JOHNSON, and E. FRACHTENBERG. 2004. On the feasibility of incremental checkpointing for scientific computing. *In*: *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April*. IEEE Computer Society, pp.58-.

SANTOS, G., A. DUARTE, D. REXACHS, and E. LUQUE. 2008. Increasing the Performability of Computer Clusters Using RADIC II. *In*: *Proc. Third International Conference on Availability, Reliability*. IEEE Computer Society, pp.653-658.

SANTOS, G., A. DUARTE, D. REXACHS, and E. LUQUE. 2008. Providing Non-stop Service for Message-Passing Based Parallel Applications with RADIC. *In*: *Proceedings of the 14th international Euro-Par conference on Parallel Processing*. Springer-Verlag, pp.58-67.

SANTOS, G., A. DUARTE, D. REXACHS, and E. LUQUE. 2006. Recuperando prestaciones en clusters tras la ocurrencia de fallos utilizando RADIC. *In*: *Proceedings of XII Congreso Argentino de Ciencias de la Computación (CACIC 2006)*. Potrero de los Funes, Argentina.

SANTOS, G., L. FIALHO, D. REXACHS, and E. LUQUE. 2009. Increasing the Availability Provided by RADIC with Low Overhead. *In*: *Proceedings of 2009 IEEE International Conference on Cluster Computing*. New Orleans, USA: IEEE, p.Accepted.

SHOOMAN, M. L. 2002. *Reliability of computer systems and networks*. John Wiley \& Sons, Inc.

SKJELLUM, Y. S. 2004. MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware. *CLUSTER COMPUTING*. **7**, pp.303-315.

SNELL, Q. O., A. R. MIKLER, and J. L. GUSTAFSON. 1996. Netpipe: A network protocol independent performace evaluator. *In*: *In Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*. ACTA Press.

SNIR, M., S. W. OTTO, D. W. WALKER et al. 1998. *MPI: The Complete Reference*. MIT Press.

SOARES, L. and J. PEREIRA. 2005. Experimental performability evaluation of middleware for large-scale distributed systems. *In*: *7th International Workshop on Performability Modeling of Computer and Communication Systems*., pp.1-4.

SONG, H., C. LEANGSUKSUN, R. NASSAR et al. 2006. Availability modeling and analysis on high performance cluster computing systems. *In*: *Proceedings of the First International Conference on Availability, Reliability and Security*. IEEE Computer Society, pp.305 - 313.

SQUYRES, J. M. and A. LUMSDAINE. 2003. A Component Architecture for LAM/MPI. *In*: *roceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, pp.379-387.

STROM, R. and S. YEMINI. 1985. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.* **3**, pp.204-226.

SUN, H., J. J. HAN, and H. LEVENDEL. 2003. Availability requirement for a fault-management server in high-availability communication systems. *IEEE Transactions on Reliability*. **52**, pp.238-244.

TOP500.ORG. 2008. *TOP500 Supercomputing Sites*. [online]. [Accessed 23 Feb 2009]. Available form World Wide Web: <http://www.top500.org>

TREASTER, M. 2005. A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems. *ACM Computing Research Repository*. **501002**, pp.1-11.

YOUNG, J. W. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM*. **17**, pp.530-531.

ZAMBONELLI, F. 1998. On the effectiveness of distributed checkpoint algorithms for domino-free recovery. *In*: *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computin*. IEEE Computer Society, pp.124-131.