Escola d' Enginyeria

Department d' Arquitectura de Computadors i Sistemes Operatius

Màster en Computaciò d'Altes Prestacions

RADIC: Un middleware de tolerancia a fallos que preserva el rendimiento.

Memoria del trabajo de "Iniciación a la Investigación. Trabajo de Fin de Máster" del "Máster en Computació d'altes Prestacions", realizada por **Hugo Daniel Meyer**, bajo la dirección de Dolores Rexachs. Presentada en la Escuela de Ingeniería (Departamento de Arquitectura de Computadores y Sistemas Operativos) de la Universitat Autònoma de Barcelona.

2011

Iniciación a la investigación.

Trabajo de fin de máster del

Máster en Computación de Altas Prestaciones.

Título: RADIC: Un middleware de tolerancia a fallos que preserva el rendimiento.

Realizada por Hugo Daniel Meyer en la Escuela de Ingeniería, en el Departamento Arquitectura de Computadores y Sistemas Operativos.

Dirigida por: Dolores Rexachs.

Firmado

Director Estudiante

Agradecimientos

En primer lugar debo agradecer a Dios que ha guiado mis pasos y decisiones, Él me ha puesto donde me encuentro hoy y me ha llenado de bendiciones y me ha hecho el camino más fácil.

Agradezco también a mi madre que ha sido un gran pilar en mi educación, y me ha facilitado todos los instrumentos para llegar a mis objetivos.

Doy gracias a mi novia Noemí, que ha sido un apoyo incondicional en esta aventura del viaje a pesar del gran sufrimiento que representa para ella esta separación física. Gracias por tu ayuda, tu comprensión, tu cariño y amor.

Agradezco a mis tutores, Lola y Emilio, quienes me han ayudado a crecer a nivel profesional y han ayudado a desarrollar y pulir este trabajo.

Agradezco también a todos mis amigos y compañeros tanto de Paraguay como de aquí (Carlos Núñez, Javier Martínez, Ronal Muresano y todos los demás) ya que son un apoyo en los momentos difíciles.

Doy también gracias a la comunidad Open MPI cuyos integrantes han colaborado bastante con el desarrollo de este trabajo, entre ellos Ralph Castain, Joshua Hursey, Jeff Squyres y otros.

Dedicatoria

Dedico este trabajo a toda mi familia, especialmente a mi madre, quienes son coautores del mismo.

Se lo dedico a una persona que ha sido mi apoyo, fuerza y sostén en esta dura estancia fuera de mi casa y de mi país, esa persona es mi novia Noemí.

Resumen

La tolerancia a fallos es una línea de investigación que ha adquirido una importancia relevante con el aumento de la capacidad de cómputo de los súper-computadores actuales. Esto es debido a que con el aumento del poder de procesamiento viene un aumento en la cantidad de componentes que trae consigo una mayor cantidad de fallos.

Las estrategias de tolerancia a fallos actuales en su mayoría son centralizadas y estas no escalan cuando se utiliza una gran cantidad de procesos, dado que se requiere sincronización entre todos ellos para realizar las tareas de tolerancia a fallos. Además la necesidad de mantener las prestaciones en programas paralelos es crucial, tanto en presencia como en ausencia de fallos.

Teniendo en cuenta lo citado, este trabajo se ha centrado en una arquitectura tolerante a fallos descentralizada (RADIC – *Redundant Array of Distributed and Independant Controllers*) que busca mantener las prestaciones iniciales y garantizar la menor sobrecarga posible para reconfigurar el sistema en caso de fallos. La implementación de esta arquitectura se ha llevado a cabo en la librería de paso de mensajes denominada Open MPI, la misma es actualmente una de las más utilizadas en el mundo científico para la ejecución de programas paralelos que utilizan una plataforma de paso de mensajes. Las pruebas iniciales demuestran que el sistema introduce mínima sobrecarga para llevar a cabo las tareas correspondientes a la tolerancia a fallos.

MPI es un estándar por defecto *fail-stop,* y en determinadas implementaciones que añaden cierto nivel de tolerancia, las estrategias más utilizadas son coordinadas. En RADIC cuando ocurre un fallo el proceso se recupera en otro nodo volviendo a un estado anterior que ha sido almacenado previamente mediante la utilización de *checkpoints* no coordinados y la relectura de mensajes desde el log de eventos. Durante la recuperación, las comunicaciones con el proceso en cuestión deben ser retrasadas y redirigidas hacia la nueva ubicación del proceso.

Restaurar procesos en un lugar donde ya existen procesos sobrecarga la ejecución disminuyendo las prestaciones, por lo cual en este trabajo se propone la utilización de nodos *spare* para la recuperar en ellos a los procesos que fallan, evitando de esta forma la sobrecarga en nodos que ya tienen trabajo.

En este trabajo se muestra un diseño propuesto para gestionar de un modo automático y descentralizado la recuperación en nodos *spare* en un entorno Open MPI y se presenta un análisis del impacto en las prestaciones que tiene este diseño. Resultados iniciales muestran una degradación significativa cuando a lo largo de la ejecución ocurren varios fallos y no se utilizan *spares* y sin embargo utilizándolos se restablece la configuración inicial y se mantienen las prestaciones.

Abstract

Fault tolerance is a research line that has gained significant importance with the increasing of the computing power of today's super-computers. The increasing of processing power comes with an increase in the number of components that brings also an increase in the number of failures.

Today's fault tolerance strategies are mostly centralized and these do not scale when using a large number of processes, since synchronization is required between them to perform the fault tolerance tasks. Maintain performance in parallel applications is crucial, in the presence or absence of fault.

According to the above, this work has focused on a decentralized fault-tolerant architecture (RADIC – Redundant Array of Distributed and Independant Controllers) that seeks to maintain the initial performance and ensure the lowest possible overhead to reconfigure the system in case of failure. The implementation of this architecture has been made in the message passing library called Open MPI. This is one of the most used message passing library in the scientific world to execute parallel programs. Initial tests show that the system introduces minimal overhead to perform fault tolerances tasks, and also show that performance is restored as it was before failure.

MPI is a fail-stop standard and some implementations that add fault tolerances use a coordinated strategy. In RADIC when a failure occur the failed process recovers in another node rolling back to a previous saved state made by using an uncoordinated strategy of checkpoint and by reprocessing the saved log. During restart, the communications with the failed process should be delayed and redirected to the new process location.

Restoring processes in a place where processes already exists overload the application and the performance decrease. In this work is proposed the inclusion of spare nodes to restore failed processes in them, avoiding performance degradation.

In this work we propose an automatic and decentralized method to manage the recovery of failed processes in spare nodes in an Open MPI environment and is also presented an analysis of the failure impact in the performance. Experimental evaluation shows a significant degradation when failures occur along a parallel execution and there is no spare nodes, nevertheless by using spares, the initial configuration and the initial performance may be restored.

Índice General

Índice de Tablas

Índice de Figuras

Capítulo 1.
Introducción

Los computadores han evolucionado bastante, con el paso de los años se han ido añadiendo y agrupando cada vez más y más componentes en pequeños espacios lo cual ha traído consigo problemas de refrigeración o de disipación. Se ha optado por la alternativa de crear agrupaciones de máquinas, lo cual se conoce como *clúster* de computadoras, que a partir de aquí denominaremos simplemente *clúster*. Han aparecido los computadores *multicore* que constan de más de un núcleo de procesamiento dentro de un mismo circuito integrado.

Los requerimientos actuales dentro de los sistemas de cómputo han obligado a incrementar tanto el número de componentes que se incluyen dentro de un chip, así como la cantidad de máquinas que conforman los *clústers* (TOP 500 Project, 2011), de esta forma se busca mejorar los tiempos de ejecución mediante el cómputo distribuido. El crecimiento en la cantidad de nodos seguirá en aumento en los próximos años, y al parecer no existe un límite para ello. Los fallos también han crecido debido al aumento de componentes del sistema y además de que son sistemas que funcionan a tiempo completo y casi siempre con bastante utilización, ya que en los centros de cómputo se ejecutan generalmente aplicaciones que duran varios días, por lo cual el impacto de un fallo puede ser muy grave.

Teniendo en cuenta lo mencionado anteriormente es evidente que a una mayor cantidad de elementos la probabilidad de que alguno falle se incrementa, es más los fallos en estos supercomputadores actuales ocurren en promedio dos veces al día (Schroeder, y otros, 2007), esto impacta en el tiempo que se puede estar ejecutando sin interrupción por un fallo y las interrupciones y/o degradación del sistema puede darse si no se cuenta con una arquitectura tolerante a fallos adecuada (Figura 1.1). Las arquitecturas actuales buscan maximizar el tiempo medio de interrupciones en los sistemas, y tratan de continuar con la ejecución de sus tareas paralelas inclusive en presencia de fallos, buscando además pagar el mínimo *overhead* por esta protección.

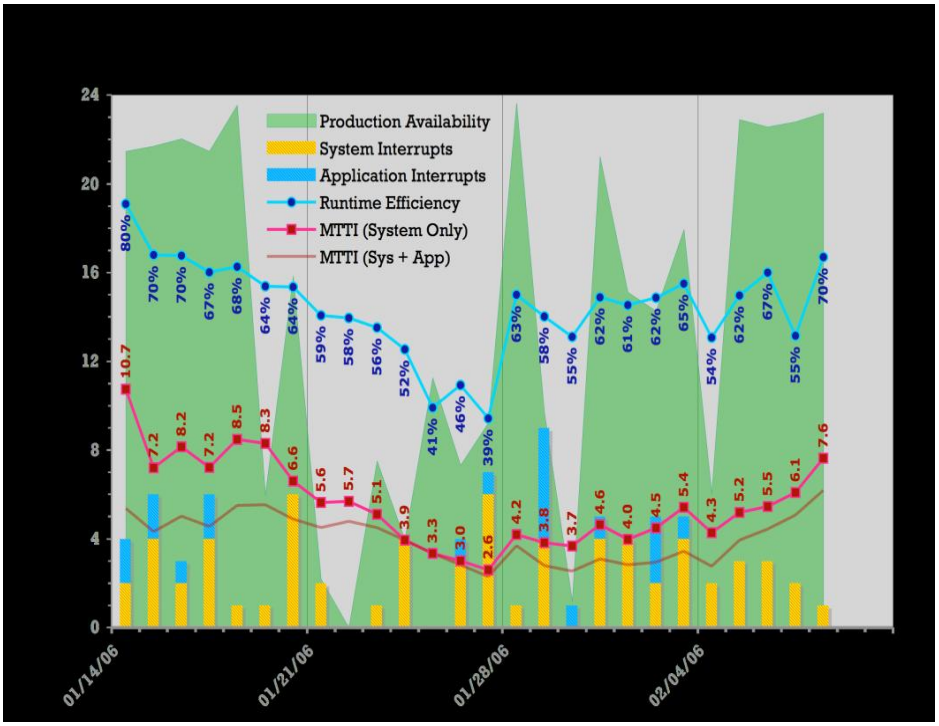

Figura 1.1 Promedio de fallos en supercomputadores actuales.

El tiempo de ejecución de aplicaciones paralelas ha adquirido gran importancia, ya que los problemas del mundo real exigen resultados en el menor tiempo posible para lo cual requieren un gran poder de cómputo. Los fallos pueden costar bastante en aplicaciones como por ejemplo las simulaciones de: sistemas meteorológicos, prototipos de hardware, prototipos de naves aeroespaciales y aplicaciones en áreas como las del diseño gráfico donde la renderización de imágenes requiere gran poder de cómputo, y perder algo de ese computo extiende los tiempos y reditúan en pérdidas económicas.

Se han desarrollado muchas técnicas que aumentan la fiabililidad y gran disponibilidad a los sistemas paralelos y distribuidos. Las técnicas más utilizadas incluyen la utilización de transacciones, comunicaciones grupales, replicación y técnicas basadas en *rollback-recovery* (Elnozahy, y otros, 2002). Cada una de estas técnicas implica diferentes relaciones de compromiso y enfocan la tolerancia a los fallos de diferentes maneras.

En este documento se hablará principalmente de una arquitectura tolerante a fallos que utiliza protocolos de *rollback-recovery* como base. RADIC – *Redundant Array of Distributed Independant fault tolerance Controllers* (Duarte, 2007), es una arquitectura

transparente que provee tolerancia fallos a sistemas de paso de mensajes que son utilizados comúnmente para desarrollar aplicaciones paralelas. La arquitectura RADIC actúa como una capa que aísla la aplicación paralela de los fallos en los nodos del sistema, enmascarando y tolerando fallos sin necesitar recursos extras para cumplir con sus objetivos. Es decir, RADIC permite construir un sistema fiable utilizando elementos no fiables.

Como antecedentes de este trabajo existe un prototipo que implementa las características básicas de la arquitectura RADIC, este prototipo no soporta todas las primitivas del estándar MPI (Duarte, y otros, 2006), también existe otro prototipo que analiza la integración de RADIC en Open MPI (Fialho, 2008), este prototipo no soporta la gestión de nodos *spare*, es importante generalizar la definición de la arquitectura RADIC adaptada a una librería estándar.

Este trabajo de iniciación a la investigación, trabajo de fin de máster presenta extensión del protocolo de tolerancia a fallos que propone RADIC adaptado a Open MPI, esto permitirá considerar restricciones de prestaciones en caso de fallos, en concreto se enfoca el estudio en las consecuencias que trae un fallo a una aplicación paralela, puesto que cuando falla algún componente, si el mismo no es reemplazado por uno de repuesto (por ejemplo, un nodo), la aplicación queda con menos recursos de los asignados inicialmente y por lo tanto la carga queda desbalanceada y el tiempo de ejecución empeora.

Se realiza ha diseñado e implementado una propuesta para integrar en el controlador descentralizado de RADIC para Open MPI, el reemplazo de nodos y el mantenimiento de un modo automático y además se evalúa el impacto en las prestaciones.

El tiempo es oro, no deseamos desperdiciar el tiempo de cómputo, por lo cual tolerar un fallo perdiendo la menor cantidad de tiempo y/o datos tiene sustancial importancia.

1.1. Justificación

Las estrategias de tolerancia a fallos actuales en su mayoría son centralizadas y estas no escalan cuando se utiliza una gran cantidad de procesos, dado que se requiere sincronización entre todos ellos para realizar la protección utilizando *checkpoint* coordinado y para la recuperación después de un fallo, siendo compleja la reconfiguración del sistema si no se dispone del mismo entorno. Además la necesidad de mantener las prestaciones en programas paralelos es crucial, tanto en presencia como en ausencia de fallos.

MPI (*"Message Passing Interface"*) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores. Existen varias implementaciones del estándar, pero no existe una versión "oficial" adecuada a las necesidades de todo el mundo.

El estándar MPI 2.0 (Forum MPI, 2009) no provee mecanismos para gestionar fallos en los sistemas de comunicación, y tampoco provee mecanismos para gestionar fallos en procesadores o nodos. En general las implementaciones tienen como política por defecto la *fail-stop*, es decir, si falla en algún momento la comunicación por algún motivo, se aborta toda la aplicación paralela, y la mayoría de las implementaciones actuales se comportan de esta manera.

MPI es un estándar de paso de mensajes donde un fallo significa la finalización de la aplicación, en algunas implementaciones la opción más utilizada para tratar este tipo de problema es la de volver de un modo no automático al último checkpoint coordinado, los cuales son explicados en detalles más adelante. Normalmente no se implementa un controlador que monitorice los fallos y actúe en caso de fallos.

Los fallos impactan de manera crítica el funcionamiento del sistema, y obligan a volver a un estado global consistente. Si no se cuenta con una arquitectura transparente que permita esto, el usuario debe ser el encargado de la gestión de los fallos y la posterior recuperación.

Open MPI da soporte a redes heterogéneas, una de las más utilizadas es TCP/IP. Los sockets que permiten la comunicación entre los nodos de un computador paralelo son cerrados inesperadamente cuando ocurre un fallo, lo cual hace necesaria la redirección de los mensajes ya sea de manera transparente o no si se quiere seguir funcionando después del fallo.

En resumen, vemos que utilizando el estándar MPI, no suele haber protección porque considera los sistemas fiables o la protección no se suele adaptar a las características del sistema, de la aplicación y a los requisitos del usuario, la alternativa más usual es el *checkpoint* coordinado. Además la recuperación no suele ser transparente, por lo tanto el tiempo hasta la recuperación completa es elevado y además si se cuenta con un esquema de tolerancia a fallos centralizado, el mismo no escala porque requiere coordinación entre los procesos que componen el ambiente paralelo.

Teniendo en cuenta lo explicado anteriormente, este trabajo se ha centrado en una arquitectura tolerante a fallos descentralizada (RADIC – *Redundant Array of Distributed and Independant Controllers*) que busca mantener las prestaciones iniciales y garantizar la menor sobrecarga posible para reconfigurar el sistema en caso de fallos.

1.2. Middleware de tolerancia a fallos

La arquitectura RADIC está basada en dos componentes principales que le permiten tolerar fallos y son llamados: protectores y observadores. Los nodos cuentan cada uno con un protector y cada proceso que pertenece a la aplicación tiene un observador. A grandes rasgos puede decirse que estos dos elementos se encargan, entre otras tareas, de realizar el log de mensajes pesimista basado en el receptor y los *checkpoints* no coordinados que utiliza RADIC.

Las implementaciones del estándar MPI o librerías más utilizadas son Open MPI (ARM, Auburn University, Bull, Chelsio Communications, Cisco Systems Inc., Computer Architecture Group, Computer Science Department, University of British Columbia,

EverGrid, IBM, Oracle, 2011) y MPICH 2 (Intel Corporation; IBM; Cray Inc.; Microsoft; Ohio State University; Myricom; University of British Columbia, 2011), como antecedente de este trabajo existe un prototipo de la implementación de RADIC se ha incluido en la librería Open MPI dado que es una de las más utilizadas. El objetivo de que RADIC pase de un prototipo a formar parte de una librería ampliamente utilizada dentro de la comunidad científica, se plantea para que de esta forma la arquitectura pueda ser utilizada con aplicaciones reales.

De acuerdo a un análisis hecho en (Fialho, 2008) se optó por Open MPI, donde se puede ver que existe cierta compatibilidad a nivel de diseño entre RADIC y Open MPI. Además, la implementación de Open MPI es modular y permite una adaptación más sencilla al momento de la implementación. Las pruebas iniciales demuestran que el sistema introduce mínima sobrecarga para llevar a cabo las tareas correspondientes a la tolerancia a fallos.

RADIC-OMPI (Fialho, 2008) es una implementación del nivel básico de la arquitectura RADIC dentro de la librería Open MPI, en la misma se buscan evitar los problemas comentados, pero aún no son considerados otros niveles de protección (las prestaciones dentro de la misma), dado que no se cuenta con elementos redundantes.

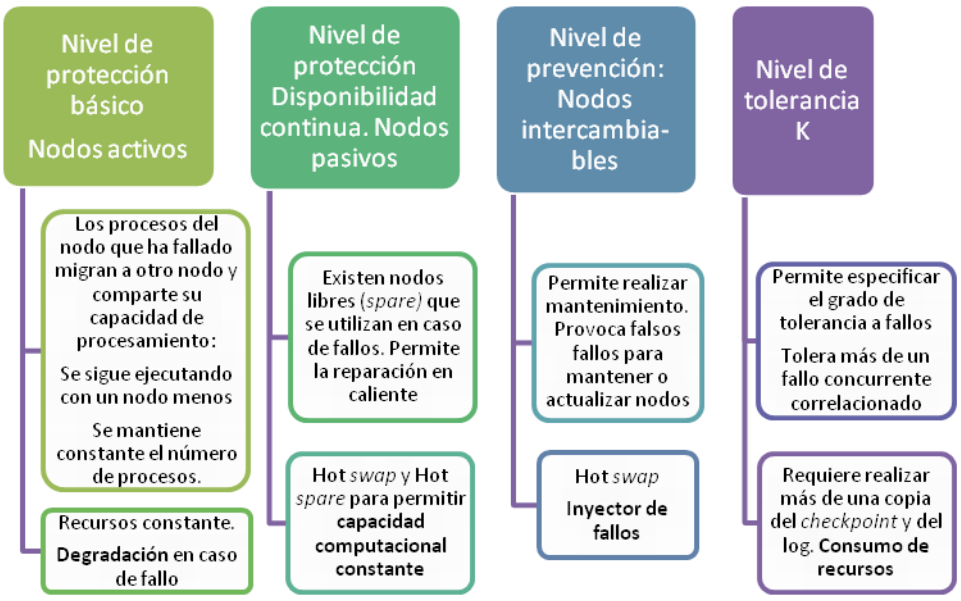

Figura 1.2. Niveles de protección de RADIC.

Además deben de considerarse otros aspectos que afectan las prestaciones como:

- rendimiento contra frecuencia de checkpoint: flexibilidad y adecuación del intervalo de checkpoint para balancear costos.
- el sistema se degrada después de uno o varios fallos que van ocurriendo a lo largo del tiempo: integrar en la arquitectura la utilización de nodos *spare* que evite la degradación por la pérdida de nodos.

Este trabajo muestra el diseño propuesto para gestionar de un modo automático, y descentralizado, la recuperación en nodos *spare* en un entorno Open MPI y controlar su impacto en las prestaciones, evitando una degradación significativa cuando a lo largo de la ejecución ocurren varios fallos.

1.3. Objetivos Generales

Como se ha mencionado en el apartado anterior, la tolerancia a fallos en los sistemas paralelos y distribuidos es un tema de vital importancia en los clúster actuales, más aún cuando se utiliza una plataforma de paso de mensajes implementada de acuerdo al estándar MPI 2.0 (Forum MPI, 2009) cuya política por defecto es *fail-stop,* es decir, si falla en algún momento la comunicación por algún motivo, se aborta toda la aplicación paralela.

Actualmente los fallos que ocurren afectan gravemente a las prestaciones de una aplicación paralela, si es que no acaban con ellas. Por esto es necesario contar con elementos redundantes para poder reemplazar a los dañados y de esta forma tratar de que las prestaciones iniciales se mantengan. Se busca que los nodos de repuesto (Spares) se integren en la arquitectura de tolerancia a fallos de manera transparente y automática.

Lo explicado anteriormente da la posibilidad de intercambiar nodos en caliente (*hot swap*), es decir mientras que la aplicación sigue funcionando. Los beneficios del intercambio en caliente son evidentes, dado que permite reconocer componentes fallados y reemplazarlos sin comprometer la disponibilidad del sistema, y buscando afectar al mínimo las prestaciones.

En este trabajo se proponen mejoras a los sistemas tolerantes a fallos existentes en la actualidad, principalmente a la implementación de RADIC hecha en Open-MPI (Fialho, 2008) mediante la integración de la utilización de nodos de repuesto, que a partir de ahora llamaremos nodos *spare*.

El objetivo principal de este trabajo es controlar el impacto en las prestaciones de las arquitecturas tolerantes a fallos tras un fallo, evitando una degradación significativa cuando a lo largo de la ejecución ocurren varios fallos. Para ello se propone:

- gestionar de un modo automático, y descentralizado, la recuperación en nodos *spare* en un entorno OpenMPI.
- Permitir la sustitución de nodos que han fallado (*hot swap*) o la incorporación de nuevos nodos *spare* (*hot spare*).

Mediante la utilización de nodos *spare* en la arquitectura RADIC se aumentará el rendimiento de las aplicaciones y la disponibilidad de los *clusters* de computadores. Además, con este sistema tolerante a fallos se busca obtener un mejor rendimiento que otros sistemas tolerantes a fallos.

Además se busca convertir RADIC en un producto comercial al incluirlo con todas sus características y mejoras dentro de la implementación de MPI denominada Open-MPI (Indiana University and Partners, 2011).

1.4. Objetivos Particulares

Modificar y adaptar la implementación actual de RADIC realizada en Open-MPI para permitir la utilización de nodos *spare* en caso de que ocurran fallos, y la reparación de nodos que han fallado, de esta manera evitar la sobrecarga en los nodos y el desbalanceo de carga al momento de recuperar el proceso que ha fallado.

Incluir nodos *spare* se hará de forma transparente, sin necesidad de información centralizada e introduciendo el menor *overhead* posible. Además con estas modificaciones

se podrá dar soporte al mantenimiento preventivo de nodos que tienen alta probabilidad de fallos.

Proponer un modelo para configurar el sistema en función de las características de la aplicación y los requisitos del usuario, definiendo el nº de nodos *spare*, ubicación, funcionamiento

Evaluar la propuesta y validar experimentalmente que presenta un resultado similar a las propuestas actuales y se mejorarán los tiempos de ejecución de una aplicación después de la ocurrencia de un fallo, ya que se mantendrán las características iniciales, por lo tanto, las prestaciones del sistema sin degradación en presencia de fallos.

1.5. Descripción de los siguientes capítulos

A continuación se ofrece una pequeña reseña de lo que se encontrará más adelante en esta memoria.

- En el capítulo 2 se tratan los conceptos básicos de la tolerancia a fallos así como las estrategias de tolerancia a fallos más utilizadas, pasando por los mecanismos de *checkpoint* y los protocolos de log. Además se tratan los problemas y soluciones comunes al utilizar cada una de estas opciones.
- En el capítulo 3 se introduce la arquitectura RADIC, con sus políticas y mecanismos. Se trata cada uno de los componentes que forman parte de esta arquitectura distribuida de tolerancia a fallos y se trata además el estándar MPI.
- En el capítulo 4 se mencionan los trabajos relacionados al realizado en esta tesis de maestría. Se destacan las ventajas y desventajas de cada uno de los métodos y se los compara con la implementación de RADIC en Open MPI.
- En el capítulo 5 se presenta la propuesta y su implementación dentro de Open MPI, destacando los problemas enfrentados y las soluciones encontradas, así como las características y mejoras introducidas.
- En el capítulo 6 se muestran los primeros resultados experimentales obtenidos, en el cual se busca mostrar que con los nodos *spare* se obtiene

una ganancia en las prestaciones de las aplicaciones cuando ocurren fallos y se restauran los procesos fallados en estos elementos redundantes.

- En el capítulo 7 se presentan las conclusiones de este trabajo, y los objetivos que fueron alcanzados.

Capítulo 2.
Estrategias de Tolerancia a Fallos

2.1. Introducción

El crecimiento del número de nodos en los *clústers* de computadores ha incrementado también la probabilidad de fallos, y esto ha exigido además el crecimiento y desarrollo de técnicas que permitan detectar los fallos y tolerarlos sin perder el cómputo realizado, o perdiendo sólo una pequeña fracción del mismo, dejando el sistema en un estado consistente (los estados consistentes serán definidos más adelante) a partir del cual puede continuar la ejecución.

El objetivo fundamental de los esquemas de tolerancia a fallos en sistemas paralelos distribuidos es que el trabajo total se ejecute correctamente, inclusive cuando falle alguno de los componentes del sistema. Estas cualidades se buscan pagando un mínimo *overhead* en ausencia de fallos, y con la mínima degradación en presencia de fallos. En este capítulo se tratarán las distintas alternativas de tolerancia a fallos y los beneficios de cada una de ellas, empezando por la replicación de datos y siguiendo con las distintas alternativas que envuelven las estrategias de *rollback-recovery*.

Para lograr la tolerancia a fallos es importante tener mecanismos de protección frente a fallos, detección y una vez diagnosticado el fallo, mecanismos de recuperación y reconfiguración del sistema que permita continuar la ejecución aislando al nodo que ha fallado (Figura 2.1).

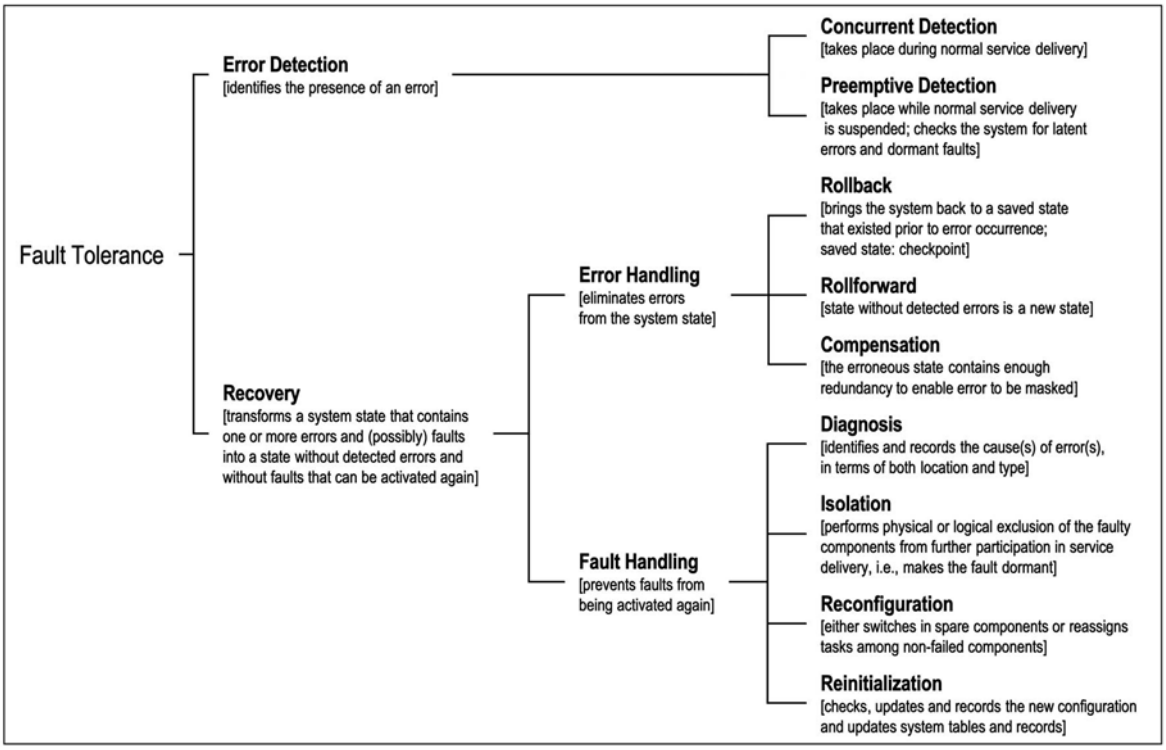

Figura 2.1. Componentes de la tolerancia a fallos.

2.2. Técnicas de redundancia

La tolerancia a fallos en los sistemas se logra mediante técnicas de redundancia (Avizienis, y otros, 2004). La redundancia puede existir en cualquier nivel, por ejemplo mediante: hardware extra (redundancia a nivel hardware), la ejecución de varias versiones de un mismo programa (redundancia a nivel software), repetición de operaciones y comparación de resultados y la utilización de técnicas de replicación de datos o códigos detectores y correctores de errores (redundancia de información) (Rodrigues de Souza, 2006) o redundancia de estados mediante la utilización de *checkpoints*, , información que permite la reejecución de operaciones en caso de fallos (redundancia temporal).

La replicación de HW o SW es un esquema de tolerancia a fallos utilizado en sistemas empotrados o con pocos componentes, pero no es una estrategia adecuada para computadores paralelos con un elevado número de componentes.

Con un modelo de programación paralela *Master/Worker* se tiene redundancia implícita si se hace un código para el *Master* y otro para el *Worker*, También existe

redundancia de procesos (código) con el paradigma (*SPMD)*. En la Figura 2.2 puede observarse el modelo *Master/Worker*, donde el *Master* distribuye los trabajos, los *Workers* computan cada uno su segmento y devuelven los resultados al *Master* para que este los procese. De esta forma existe una redundancia de procesos, la estrategia entonces para contar con tolerancia a fallos sería la replicación de datos y de esta manera se simplificaría la gestión de estados congruentes (Weissman, 1998).

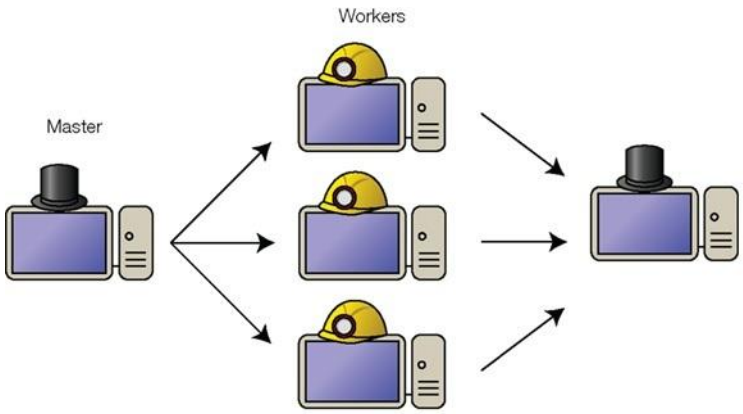

Figura 2.2 Modelo de programación Master/Worker.

Existen trabajos que proponen un modelo funcional de tolerancia a fallos en entornos *multicluster* distribuidos geográficamente que utilizan el modelo *Master/Worker*, un ejemplo es el FTDR (*Fault Tolerant Data Replication*) (Rodrigues de Souza, 2006).

La replicación de datos podría ser una solución eficiente si se conoce el paradigma de programación, tal y como hemos visto para el modelo *Master/Worker* (en el caso de fallo de un *worker*), o SPMD, pero en otros casos donde la redundancia de procesos no es tal, es necesario no solo replicar los datos, sino también los procesos y los protocolos de *rollback-recovery* proporcionan estrategias de tolerancia a fallos más generales. En el siguiente apartado se abordan estos métodos. Vemos que si se busca un esquema de tolerancia a fallos transparente, que no necesite tener información del modelo de la aplicación debemos recurrir a estrategias de *rollback-recovery*.

2.3. Conceptos básicos para protocolos Rollback-Recovery

Los protocolos de *rollback-recovery* tratan al sistema distribuido como una colección de procesos que se comunican a través de una red. Lo que se busca al utilizar protocolos de *rollback-recovery* es resumir o recuperar la computación desde un estado consistente global, más abajo se explica lo que quiere decir que un estado es global consistente, y como se guardan estos estados y los eventos no determinísticos que lo rodean. Uno de los requisitos de estos protocolos es tener acceso a algún tipo de almacenamiento estable que sobreviva a posibles fallas para de esta forma poder ser restaurados a partir de los datos almacenados en dicho almacenamiento (Figura 2.3).

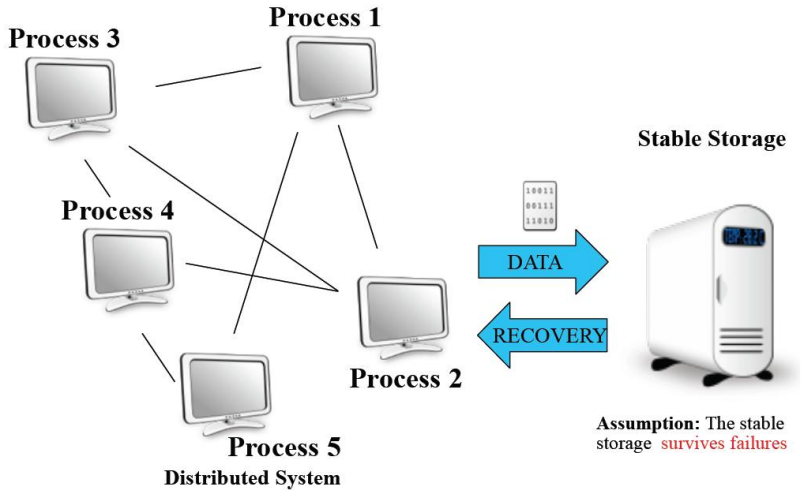

Figura 2.3 Tolerancia a fallos basada en *Rollback-Recovery*.

Cuando un fallo ocurre, se utiliza la información salvada para reiniciarse desde un estado anterior al momento del fallo, si el estado almacenado es muy reciente, entonces existe una mínima perdida en la computación, en caso contrario, no es así (Figura 2.4).

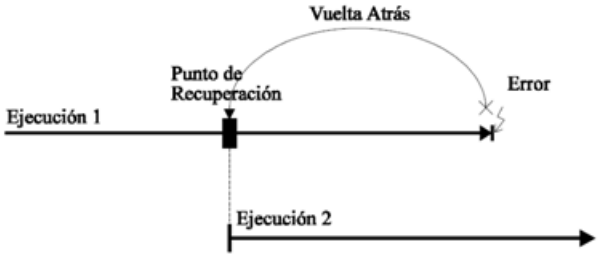

Figura 2.4. Recuperación hacia atrás

La información de recuperación contiene al menos los estados de los procesos participantes, estos estados se denominan *checkpoints*. Algunos protocolos de *rollback-recovery* necesitan además contar con un log de eventos ya sean de interacción con dispositivos de entrada/salida, comunicación entre procesos, etc.

Hoy en día se busca los protocolos de *rollback-recovery* sean transparentes (no requieran intervención humana) y de esta manera reducir los tiempos de reparación/recuperación y su impacto en el tiempo de ejecución de las aplicaciones en presencia de fallos.

En los sistemas de paso de mensajes (que actualmente ocupan un puesto preponderante en los sistemas paralelos escalables) se complican los mecanismos de *rollback-recovery* porque los mensajes son eventos que inducen a una cierta dependencia entre los procesos.

La interdependencia de procesos trae consigo algunos problemas asociados, por ejemplo, si un proceso falla puede hacer que otros procesos que no han fallado vayan a un estado anterior para que se sincronice con el proceso fallado, esto es lo que se conoce como *rollback propagation*. En determinadas condiciones se podría volver tan atrás como el comienzo, si un proceso hace que otro retroceda y así sucesivamente con todos hasta llegar al estado inicial, perdiendo todo el trabajo hecho. Esto es llamado “efecto dómino” (Randell, 1975), y además requiere tener todos los *checkpoints* anteriores guardados.

El efecto dómino puede ocurrir cuando se utilizan *checkpoints* no coordinados e independientes, que son justamente los que utiliza RADIC, para evitar este problema se han diseñado técnicas, como por ejemplo el log de mensajes, del cual se hablará más adelante.

En un sistema de paso de mensajes se cuenta con procesos que se comunican entre sí para de esta manera ejecutar tareas de manera paralela generalmente e interactuar con el mundo exterior (Figura 2.5)

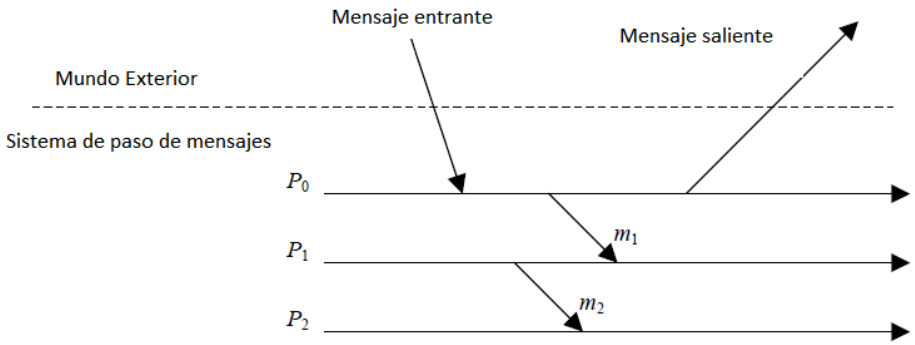

Figura 2.5 Sistema de paso de mensajes con 3 procesos y su interacción.

La ejecución de un proceso es una secuencia de estados entre distintos intervalos, cada uno iniciado por un evento no determinístico, por ejemplo, si llega un mensaje a un proceso, generalmente este empieza a computar con la información de este mensaje, lo cual da inicio a la ejecución de una nueva fase y lleva a un nuevo estado. La ejecución durante cada estado es determinístico, por ejemplo, si la ejecución de un proceso inicia en un estado y es sometido a los mismos eventos no deterministas una y otra vez se llegará a un mismo resultado. Existe un concepto llamado *piecewise deterministic assumption* (PWD) que es una asunción que indica que se puede detectar y capturar información suficiente acerca de los eventos no deterministas que inician los intervalos de estado.

Para que un sistema o parte de él sea reiniciado correctamente luego de un fallo debe restaurarse a un estado consistente con el comportamiento observado en el sistema antes de que ocurra el fallo (Strom, y otros, 1985). Para cumplir con esta consistencia el protocolo utilizado debe mantener la información de las interacciones internas entre los procesos así como las externas.

A continuación se hacen algunas definiciones importantes que ayudarán a comprender mejor los protocolos de *rollback-recovery* (Kalaiselvi, y otros, 2000):

DEFINICIÓN 1 (CHECKPOINT LOCAL).

Un *checkpoint* local es un evento que graba el estado de un proceso presente en un determinado procesados en un determinado instante.

DEFINICIÓN 2 (CHECKPOINT GLOBAL).

Es una colección de *checkpoints* locales, uno de cada procesador. Para realizar un *checkpoint* global consistente, debe hacerse en un estado consistente, los estados consistentes son explicados a continuación.

En general los protocolos de *rollback-recovery* difieren en la forma en la que ven la fiabilidad de la red, ya que algunos consideran a la red fiable y que devuelven el mensaje en orden FIFO (Chandy, y otros, 1985) y otros asumen que la red puede perder, duplicar o reordenar mensajes (Johnson, 1990).

2.3.1. Estados Consistentes

El estado global de un sistema está conformado por todos los estados individuales (*checkpoint* de los procesos y los estados de los búferes de los canales de comunicación. Entonces, un estado consistente sería por ejemplo aquel en el que si en el estado de un proceso refleja la recepción de un mensaje, y el estado del emisor correspondiente refleja el envío de ese mensaje (Chandy, y otros, 1985). Continuando con las definiciones, cabe mencionar dos más (Kalaiselvi, y otros, 2000):

DEFINICIÓN 3 (RELACIÓN "OCURRE ANTES" DE LAMPORT **(Lamport, 1978)**).

1) Si a y b son dos eventos que ocurren en el mismo proceso y a ocurre antes que b, entonces $a \rightarrow b$.

2) Si a es el evento que envía un mensaje, y b es el evento que recibe el mensaje en otro proceso, entonces $a \rightarrow b$.

DEFINICIÓN 4 (EVENTOS CONCURRENTES).

Dos eventos a y b se dice que son concurrentes si $(a \not\rightarrow b)$ y $(b \not\rightarrow a)$. Se denota concurrencia por medio de $\perp$.

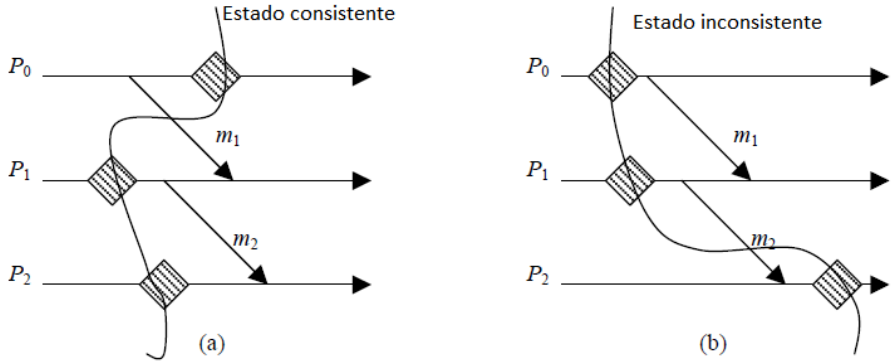

Figura 2.6. a) Estado consistente del sistema. b) Estado inconsistente del sistema.

En la Figura 2.6 (a) puede verse un estado consistente global del sistema porque se muestra que el mensaje m_1 ha sido ya enviado pero no ha sido recibido en P_1 y esto es consistente porque el mensaje ha dejado ya el emisor y está en la red. Sin embargo el caso (b) no es consistente porque el proceso P_2 en teoría ya recibió un mensaje que no fue aún enviado por P_1. Está claro que esta inconsistencia se hará efectiva si falla P_1 y vuelve a enviar el mensaje m_2.

2.3.2. Mensajes en tránsito

Un ejemplo de mensaje en tránsito es el observado en la Figura 2.6 (a) donde m_1 fue enviado pero aún no fue recibido. Con estos mensajes debe lidiarse de alguna forma, ya que si por ejemplo falla el proceso P_1 de la Figura 2.6 (a) y es recuperado por medio de un mecanismo de *rollback-recovery* de alguna manera debe entregársele el mensaje por el cual estará esperando. Por ejemplo pidiendo al proceso que retorne a un *checkpoint* anterior.

Si se está considerando un protocolo de comunicación confiable por debajo del protocolo de *rollback-recovery* solo se deberán gestionar los mensajes en tránsito cuando ocurre un fallo, y entregárselos de alguna manera al receptor, por ejemplo, con un mecanismo de log (sobre estos se hablará más adelante). Sin embargo si se considera un protocolo de comunicación no confiable, se deberán gestionar y distinguir los fallos del propio protocolo por ser no fiable de los fallos en los procesos en sí.

2.3.3. Log de Mensajes

De acuerdo a lo que se mencionó anteriormente, si se utiliza un mecanismo de c*heckpoint* no coordinado es necesaria la grabación de los eventos no deterministas para que de esta forma el proceso al restaurarse pueda arrancar desde su último estado guardado y releer los mensajes recibidos posteriormente al punto en el cual se hizo el *checkpoint*.

Utilizando log de mensajes se evita el problema del "efecto dominó" (Koren, y otros, 2007) que fue explicado anteriormente. La recuperación basada en log de mensajes se apoya en la asunción *piecewise deterministic* (*PWD)* (Strom, y otros, 1985), por lo tanto el protocolo *rollback-recovery* puede identificar los eventos no determinísticos y guardar la información necesaria para poder releerlos al momento de una restauración.

Eventos no determinísticos no son solo los mensajes recibidos, sino también entradas desde el mundo exterior, o interrupciones de un proceso a otro por algún motivo.

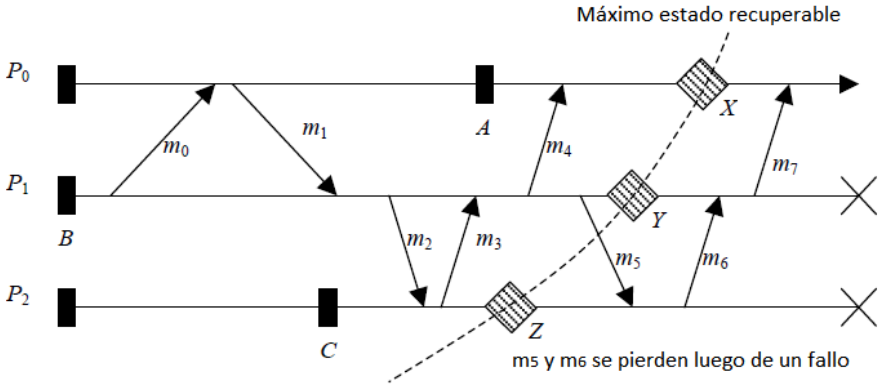

Figura 2.7 Ejemplo de log de mensajes para reproducción determinista.

Si se considera la Figura 2.7 y se supone un fallo en los procesos P_1 y P_2 antes de que puedan guardar los mensajes m_5 y m_6, y luego son restaurados los procesos, el mensaje m_7 será un mensaje huérfano, ya que P_2 no garantiza que podrá generar de vuelta el mensaje m_6 y por lo tanto P_1 no garantiza la generación de m_7 por lo cual P_0 quedaría como un proceso huérfano y deberá volver a atrás. Los puntos X, Y, Z indican el máximo estado recuperable, que sería el estado consistente más reciente al cual se podría volver si m_5 y m_6 se hubiesen guardado. Como esto no es así, los procesos P_0 y P_2 vuelven a los *checkpoints*

A y C respectivamente, y P_1 vuelve al B, y vuelve a releer los mensajes m_1 y m_3 que había recibido y están guardados, y de esta forma el sistema vuelve a un punto estable.

2.4. Protocolos de Checkpoint

En los sistemas multiprocesador la probabilidad de la ocurrencia de un fallo en algún componente es alta. Si se considera una tasa de fallos alta, la realización de *checkpoints* y la utilización de *rollback-recovery* se hacen bastante útiles, pero la realización de *checkpoints* es más complicada en sistemas distribuidos porque cada uno lleva la ejecución por su cuenta y no existe un reloj global, lo cual complica la sincronización para realizarlos.

Existen aspectos que deben ser tenidos en cuenta cuando se utilizan protocolos de *checkpoint*, como los siguientes (Kalaiselvi, y otros, 2000):

- **Frecuencia de *checkpoints*:** los algoritmos de *checkpoint* son ejecutados a la par del cómputo que realiza el proceso, por lo tanto debe ser minimizado el *overhead* introducido por los mismos. Los *checkpoints* deben permitir la recuperación rápidamente sin perder demasiado trabajo realizado, lo que hace que sean necesarios realizarlos frecuentemente, pero introduciría demasiado *overhead*. Entonces el número de *checkpoints* que deben realizarse debe asegurar la mínima perdida de información en caso de fallos y añadir el mínimo de *overhead* a la ejecución sin fallos.
- **Contenido del *checkpoint*:** El estado de un proceso debe ser salvado en un almacenamiento estable para poder ser restaurado posteriormente. El estado incluye código, datos y el contenido de la memoria y registros que utiliza el proceso.

Además de los aspectos mencionados deben de tenerse en cuenta los *overheads* introducidos por los *checkpoints*, en las siguientes dos definiciones se los destacan.

DEFINICIÓN 5 (*OVERHEAD* DE COORDINACIÓN).

En un sistema distribuido es necesaria la coordinación entre los procesos para obtener un estado global consistente, aunque la misma podría evitarse con la utilización de

logs para guardar eventos no deterministas. Algunos sistemas envían información de coordinación en los mensajes regulares. Estos introducen *overhead* en la ejecución.

DEFINICIÓN 6 (*OVERHEAD* AL GUARDAR EL CONTEXTO).

El tiempo que toma salvar el contexto global del proceso es proporcional al tamaño del contexto, además si el almacenamiento estable no está disponible en cada nodo del sistema, entonces debe ser transferido por medio de la red, lo cual también introduce un costo.

En la Figura 2.8 se observa un resumen de los protocolos de *checkpoint*, en este trabajo se hablará de los más utilizados.

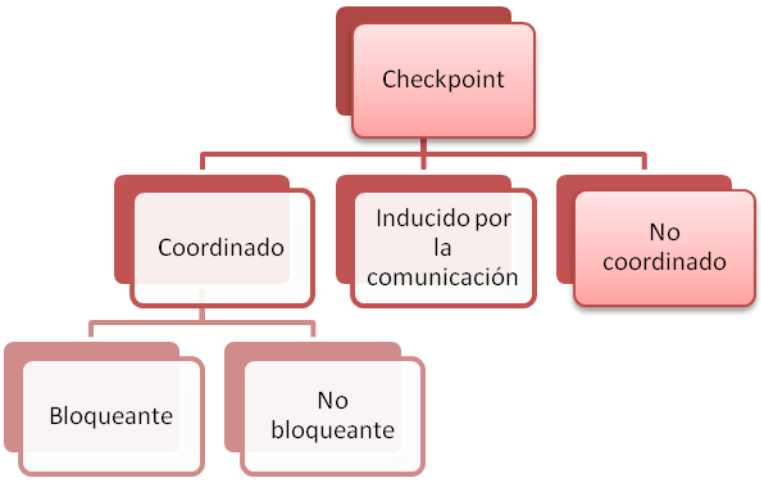

Figura 2.8. Protocolos de *checkpoint.*

2.4.1. Checkpoint no coordinado

La utilización de *checkpoints* no coordinados trae consigo muchas ventajas para los sistemas distribuidos, ya que es una solución escalable porque no se requiere un acuerdo entre los procesos para realizar *checkpoints* (Figura 2.7). Los procesos poseen autonomía para decidir en qué momento realizar *checkpoints*. Las principales ventajas son la autonomía (por ejemplo el proceso puede hacer *checkpoints* cuando lo que debe grabar es pequeño) y la escalabilidad que proporciona.

Pero no todo es ventajoso al utilizar el enfoque no coordinado, dado que también trae consigo problemas como la susceptibilidad al efecto dominó (necesidad de log), el

almacenamiento se complica un tanto ya que cada proceso debe guardar sus *checkpoints* y cada tanto deshacerse de los que ya no son útiles. Y además ese necesaria una coordinación global para computar la línea de recuperación.

Durante la ejecución normal, es decir sin fallos, los procesos calculan la interdependencia entre sus *checkpoints* de acuerdo a (Bhargava, y otros, 1988). Esta técnica consiste en enviar información sobre el *checkpoint* en cada mensaje que se envía a otro proceso. El proceso receptor usa esta información para calcular la dependencia entre él y el emisor. Entonces si ocurre un fallo, el proceso de recuperación inicia el *rollback* solicitando mediante un *broadcast* información sobre las dependencias de acuerdo a lo que almacena cada proceso. Cuando un proceso recibe esa solicitud, detiene su ejecución y responde con la información de dependencia guardada en su almacenamiento estable. Entonces el proceso que realizó el pedido calcula la línea de recuperación y una vez que lo sepa de vuelta realiza un *broadcast* con este cálculo. Entonces aquellos procesos cuyo estado actual pertenece a la línea de recuperación simplemente continúa su ejecución, en otro caso hace *rollback* al *checkpoint* indicado por la línea de recuperación.

2.4.2. Checkpoint coordinado

Las estrategias de *checkpoint* coordinado requieren que los procesos lleguen a un acuerdo y sincronicen sus *checkpoints* para formar un estado global consistente con los mismos. Esta estrategia trae consigo mayor simplicidad al momento de la recuperación y no es susceptible al efecto dominó dado que cada proceso siempre reinicia desde su *checkpoint* más actual. Además es necesario que cada proceso guarde nada más el *checkpoint* más reciente, porque los demás procesos en caso de recuperarse son autosuficientes con la información que poseen, y de esta forma se elimina el *overhead* de tener varias versiones de *checkpoint* y de la eliminación de los ya no utilizados.

La principal desventaja de este método es el *overhead* por sincronización entre los procesos para que todos lleguen a un estado consistente y formen un estado global consistente, evitando los mensajes en tránsito. Algunas propuestas bloquean todas las comunicaciones mientras el protocolo de *checkpoint* se ejecuta (Tamir, y otros, 1984). Un coordinador realiza un *checkpoint* y envía un mensaje a todos solicitándoles que hagan lo

mismo, y cuando los procesos reciben ese mensaje, detienen su ejecución, limpian los canales de comunicación y realizan un intento de *checkpoint* y avisan al coordinador enviando un *ack*. Luego de que el coordinador recibe el *ack* de todos los procesos envía a todos un mensaje de *commit* para que todos completen sus *checkpoints* siendo de esta forma un protocolo de dos fases. Esta alternativa es bastante costosa, por lo cual se prefieren los mecanismos de *checkpoint* no bloqueantes.

Una alternativa de *checkpoint* coordinado no bloqueante puede ser utilizando la idea de *distributed snapshot* (Chandy, y otros, 1985), en donde son utilizados marcadores que indican los pedidos de *checkpoint*. En este protocolo el iniciador realiza un *checkpoint* y hace *broadcast* de un marcador (un pedido de *checkpoint*) a todos los procesos. Cada proceso cuando recibe el primer marcador realiza el *checkpoint* y reenvía el marcador a todos los demás procesos antes de mandar cualquier mensaje que corresponda a la aplicación, esto funciona usando canales confiables FIFO. Si es canal no es FIFO, el marcador puede ser enviado junto con cada mensaje post *checkpoint*.

2.5. Técnicas de rollback-recovery basados en Log de Mensajes

Los protocolos de *checkpoint* basados en log de mensajes consideran que la ejecución de procesos puede ser modelada por una secuencia de eventos deterministas, cada uno de ellos iniciado por un evento no determinista (Strom, y otros, 1985). Dicho evento no determinista podría ser la recepción de un mensaje desde otro proceso o desde él mismo.

Este tipo de protocolo asume que todos los eventos no deterministas pueden ser identificados y guardados en almacenamiento estable. Durante una ejecución sin fallos del proceso paralelo cada proceso guarda los eventos no deterministas. Si además cada uno realiza sus respectivos *checkpoints* se evita volver muy atrás al momento de una recuperación. Cuando ocurre un fallo, el proceso fallido se recupera a partir del *checkpoint* almacenado y del log para releer los eventos no deterministas (mensajes recibidos) en la secuencia que ocurrieron los mismos.

Los protocolos de *rollback-recovery* garantizan que luego de la recuperación ningún proceso queda huérfano. Las diferentes alternativas de implementación afectan de manera diferente el rendimiento de la aplicación en ausencia de fallos, simplicidad en la recuperación y así re ejecución correcta de los procesos. A continuación se mencionan los tres tipos de protocolos basados en log existentes (Figura 2.9). En este trabajo solo se tratarán los mecanismos de logs que cuentan con más implementaciones.

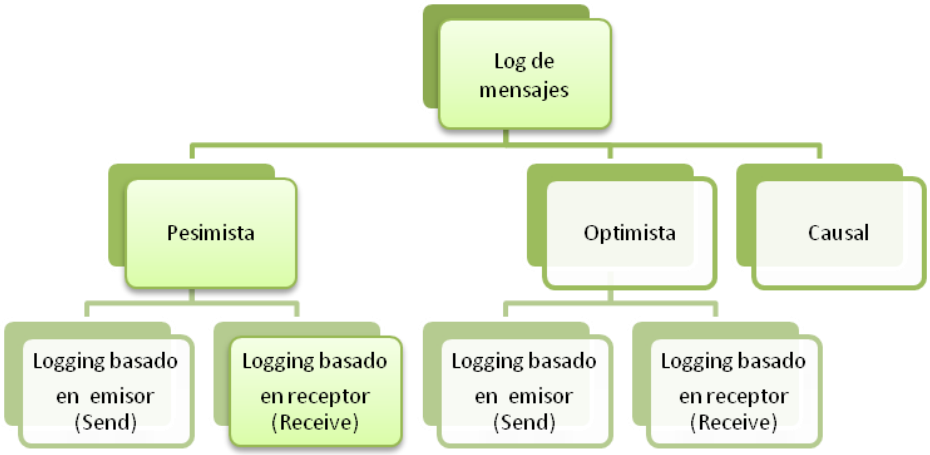

Figura 2.9. Protocolos de Log.

Los mecanismos de log que se mencionan a continuación pueden ser implementados basándose en el emisor o en el receptor. Cuando un mecanismo de log está basado en el emisor, los mensajes guardados son los enviados, en contrapartida, cuando el mecanismo está basado en el receptor, son los mensajes recibidos los que son almacenados.

2.5.1. Log pesimista

Este tipo de protocolo está diseñado asumiendo que un fallo puede ocurrir luego de un evento no determinístico en la ejecución. Esta asunción es "pesimista" dado que en realidad los fallos son ocasionales. En su forma más estricta, el log pesimista guarda en almacenamiento estable cada uno de los eventos no deterministas antes de que se permita al mismo afectar en alguna forma el cómputo. Esto quiere decir que si un mensaje no fue grabado, entonces ningún proceso depende de él.

Además de los mecanismos de log propios, los procesos también deberían realizar *checkpoints* periódicos para limitar la cantidad de trabajo que deben re ejecutar en una recuperación. Un problema es el *buffer* de log, el *checkpoint* permite gestionar dicho *buffer*.

En la Figura 2.10 pueden verse los procesos (con sus respectivos mensajes guardados) P_0 [m_0, m_4, m_7], P_1 [m_1, m_3, m_6] y P_2 [m_2, m_5]. Si ocurre un fallo en los procesos P_1 y P_2, ellos reinician desde el *checkpoint* B y C respectivamente y releyendo sus mensajes en el orden que ocurrieron. Esto garantiza que P_1 y P_2 repetirán el mismo proceso hasta llegar a un estado consistente con el estado del proceso P_0 que incluye la recepción del mensaje m_7.

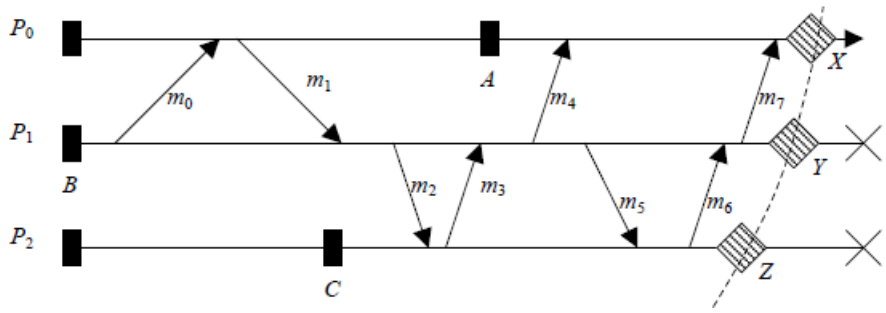

Figura 2.10. Log Pesimista.

En un sistema de log pesimista el estado observado de cada proceso es recuperable, y esta propiedad trae consigo algunas ventajas (Elnozahy, y otros, 2002):

- Los procesos pueden enviar mensajes al mundo exterior sin utilizar ningún protocolo especial.
- Los procesos pueden reiniciar desde el último *checkpoint*, limitando de esta forma la pérdida de cómputo.
- La recuperación es más simple dado que los efectos de un fallo son confinados solamente a los procesos que fallan. Los demás procesos siguen su ejecución normalmente y no quedan huérfanos dado que los procesos se restauran al último estado que incluye las interacciones más recientes con los otros procesos.

- La recolección de basura es sencilla. Los *checkpoints* antiguos y logs ocurridos antes del *checkpoint* más reciente pueden ser descartados ya que no será utilizados.

Por estas ventajas obviamente se paga un precio, y el precio es una **degradación** en la performance que ocurre por el log síncrono, por este motivo, las implementaciones deben utilizar técnicas especiales para reducir los efectos negativos de este tipo de log. Este tipo de log (junto con el protocolo de *checkpoint* no coordinado) es el que es utilizado en la arquitectura RADIC (Duarte, 2007), la cual es utilizada como base para el desarrollo de esta tesis, y en la cual se incluyen estas mejoras para optimizar el rendimiento de las aplicaciones.

2.5.2. Log optimista

A diferencia del protocolo descripto anteriormente, aquí los procesos guardan sus logs de manera asincrónica a almacenamiento estable (Strom, y otros, 1985). Estos protocolos asumen que el log concluirá antes de que un fallo ocurra. Los mensajes son guardados temporalmente en un almacenamiento volátil y periódicamente es guardado en almacenamiento estable. Con este tipo de log, la aplicación no debe bloquearse esperando a que los mensajes se guarden efectivamente en almacenamiento estable por lo cual el *overhead* en ausencia de fallos es mínimo. Pero estas ventajas traen mayor complicación a la hora de la recuperación y de la recolección de basura.

Si un proceso falla los mensajes guardados en almacenamiento volátil se pierden y los estados que eran iniciados por estos mensajes no podrán ser recuperados, es más, si el proceso fallado envío un mensaje durante alguno de los estados que no puede ser recuperado, el receptor de este mensaje se convierte en un proceso huérfano y debe volver atrás hasta antes de la recepción del mensaje.

Si se considera la Figura 2.11 y suponemos que el proceso P_2 falla antes de que el mensaje m_5 fuese guardado en almacenamiento estable. Entonces el proceso P_1 se vuelve un proceso huérfano y debe volver atrás para deshacer los efectos de la recepción del

mensaje huérfano m_6. El retroceso (*rollback*) del proceso P_1 fuerza al proceso P_0 a volver atrás para deshacer los efectos de la recepción del mensaje m_7.

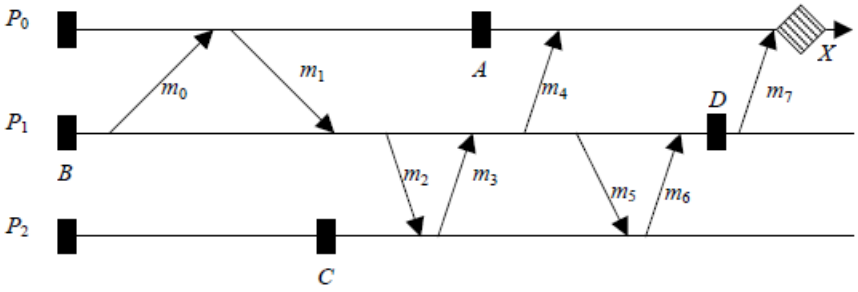

Figura 2.11. Log Optimista.

Para realizar correctamente los *rollbacks*, el mecanismo de log optimista utilizado lleva cuenta de las dependencias causales debidas a la comunicación durante la ejecución sin fallos. Cuando un fallo ocurre la información de dependencias es usada para calcular y recuperar el último estado global consistente. De acuerdo a lo expuesto es obvio que los mecanismos de recolección de basura no son triviales, dado que es necesario contar no solo con el *checkpoint* más reciente. Por ejemplo, si el proceso P_2 de la Figura 2.11 falla, fuerza al proceso P_1 a reiniciar desde el *checkpoint* B en lugar del *checkpoint* D.

La recuperación en los protocolos optimistas puede ser síncrona o asíncrona. En la recuperación síncrona (Johnson, 1989) todos los procesos corren un protocolo de recuperación para computar el estado recuperable más reciente basada en las dependencias y la información del log guardada y luego realizar los *rollbacks* correspondientes. En la recuperación asíncrona, un proceso fallado reinicia enviando un anuncio de *rollback* a todos o un mensaje de restauración para iniciar una nueva "encarnación" (Strom, y otros, 1985). Cuando un proceso recibe un anuncio de *rollback*, el proceso vuelve atrás si detecta que se ha vuelto huérfano con respecto a ese anuncio, y luego envía a todos el aviso de su propio *rollback*. Dado que pueden existir múltiples anuncios de "encarnaciones" del mismo proceso, cada proceso necesita llevar cuenta de la dependencia de su estado con cada "encarnación" de todos los procesos para así detectar estados huérfanos.

2.6. Factores que influyen en las prestaciones

A continuación se citan brevemente algunos de los factores que entran en juego cuando se trabaja con tolerancia a fallos.

Siempre existe una latencia en la detección de errores que es necesario minimizar, dado que cuanto más rápidamente se detecte un fallo, más rápido se tomarán acciones correctivas, para esto, es necesario introducir ciertos mecanismos de protección que introducen *overhead* a la ejecución. Además es casi inevitable una degradación luego de la ocurrencia de un fallo, está degradación está causada por las tareas de recuperación, reejecución y reconfiguración del sistema. En la siguiente lista se citan y resumen estos factores que introducen *overhead* o degradación en el sistema al utilizar tolerancia a fallos:

- Costo operacional (Gestión de *checkpoints*, logs, fallos, etc).
- *Overhead* de protección.
    - *Overhead* de los *checkpoints*.
    - Retrasos en los mensajes por la utilización de logs.
- Costo en recursos.
    - Almacenamiento.
    - Ancho de banda de la red.
- Transparencia.
    - Aplicación: Dado que no puede modificarse una aplicación, se introduce *overhead* en otras capas para ser transparentes.
    - Administración: los procesos automáticos tienen su costo, dado que consumen recursos.
- Adaptación a las averías de componentes (Degradación).
    - Cambios en el *mapping* inicial, migración de procesos.
    - Desbalanceo de carga.
    - Desintonización del sistema.

Los sistemas de tolerancia a fallos denominados de “Prestaciones constantes” cuentan con módulos de reserva que permiten restaurar los componentes dañados, ya sea añadiendo elementos redundantes de manera dinámica o reparando los fallados y

reintroduciéndolos en el sistema. En los sistemas denominados de “Recursos Constantes” la degradación se presenta, ya que cuando un componente falla, el trabajo que realizaba el mismo es derivado a otro.

Algunos de los problemas con lo que se debe lidiar para evitar la degradación de las prestaciones son los siguientes:

- Diagnóstico de fallos rápido para evitar la propagación de los efectos del fallo. (todos los procesos que dependen de la comunicación con el proceso parado se irán parando poco a poco).
- Interconexión rápida y tolerante a fallos: si falla la interconexión y aísla a un nodo, se considera fallo de nodo.
- Recuperación de tareas; mecanismos de comunicación y sincronización
- El componente afectado debe ser desconectado y el sistema se debe reconfigurar.

Capítulo 3.
RADIC

3.1. Arquitectura RADIC

En el trabajo (Duarte, 2007) se ha propuesto una arquitectura de tolerancia a fallos denominada RADIC (*Redundant Array of Distributed and Independent fault tolerance Controllers*). Esta es una arquitectura de tolerancia a fallos para Sistemas de Paso de Mensajes que se encarga de enmascarar los fallos que pueden ocurrir en un sistema paralelo y de tolerarlos para poder continuar la ejecución.

Esta arquitectura utilizando sus políticas provee Transparencia, Descentralización, Escalabilidad y Flexibilidad, en la Tabla 3-1 pueden observarse estas características y una breve descripción de cómo se consiguen.

Tabla 3-1. Características de RADIC.

Característica	¿Cómo se consigue?
Transparencia	• No son necesarios cambios en la aplicación. • No es necesaria intervención alguna del administrador o programador para gestionar los fallos.
Descentralización	• No existe un elemento central o plenamente dedicado. • Todos los nodos computan y también se encargan de la protección.
Escalabilidad	• Las operaciones de RADIC no son afectadas por el número de nodos en un computador paralelo.
Flexibilidad	• Los parámetros de tolerancia a fallos son ajustados de acuerdo a los requerimientos de la aplicación. • La arquitectura puede adaptarse a la estructura de la computadora paralela y el patrón de fallos.

Las características son conseguidas mediante la utilización de las políticas y mecanismos que son mostrados en la Figura 3.1.

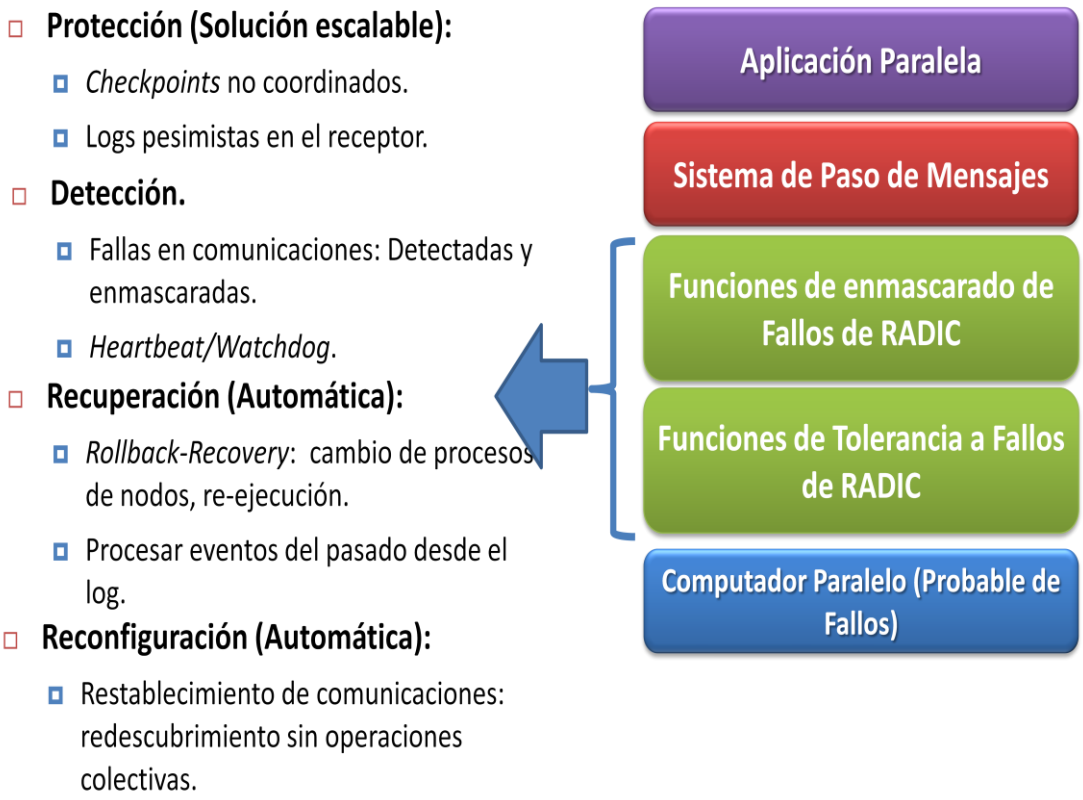

Figura 3.1. Políticas y mecanismos de RADIC.

Utilizando las políticas mostradas en la figura anterior es como RADIC implementa sus mecanismos de enmascaramiento y tolerancia a fallos. Estos mecanismos están actualmente ubicados en unas capas intermedias, entre el sistema de paso de mensajes y el computador paralelo, el cual puede fallar.

La capa más baja de RADIC implementa los mecanismos de tolerancia a fallos y la capa de más arriba se encarga de enmascarar los fallos de la librería de paso de mensajes. Estas capas se encargan de gestionar los fallos automáticamente, e incluso de sobrevivir a ellos.

3.2. Componentes funcionales

La arquitectura RADIC dispone de un controlador totalmente distribuido que cuenta con dos componentes principales, los cuales se encargan de las labores de tolerancia a fallos que fueron mencionadas en la Figura 3.1. Las entidades se denominan "observadores" y "protectores", donde cada proceso que se ejecuta en la aplicación paralela cuenta con un observador ligado a él, y cada nodo cuenta con un protector presente en él, que sería como un demonio monitor residente.

3.2.1. Observador

Como ya se ha mencionado, los observadores son procesos que están atados a la aplicación. Los observadores implementan el mecanismo de paso de mensajes para la aplicación paralela, y cada uno se encarga de realizar las siguientes tareas:

- Buscar un protector en el cual almacena sus *checkpoints* y logs.
- Realizar *checkpoints* del proceso con el cual está vinculado y guardar los mensajes recibidos (Log de mensajes). Los *checkpoints* y el log de los mensajes recibidos son enviados a un "protector" en otro nodo.
- Detectar y enmascarar fallos entre las comunicaciones de los procesos y de los procesos con su protector.
- En la etapa de recuperación luego de un fallo, debe releer los mensajes desde el log y buscar un nuevo protector.
- Mantener una tabla en donde están mapeados los nodos con su Rank MPI, esta tabla es llamada *radictable*, y en ella se indica el lugar de cada proceso de aplicación y sus respectivos protectores.

3.2.2. Protector

Como ya se ha mencionado anteriormente, existe un proceso protector en cada nodo del computador paralelo. Cada uno de estos elementos, se comunica con dos protectores vecinos, formando de esta forma entre todos los protectores un sistema de protección paralelo y descentralizado. Mediante esta comunicación entre vecinos un

protector puede detectar el fallo de otro nodo y encargarse de las tareas de recuperación. En la Figura 3.2 puede observarse una red de protectores en 4 nodos, los cuales se comunican entre sí para monitorear sus estados.

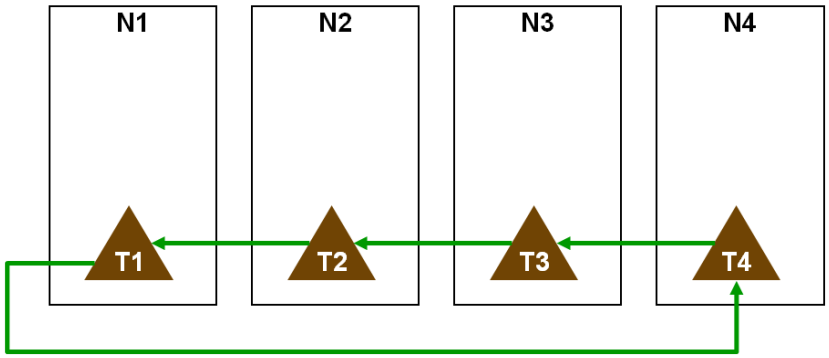

Figura 3.2. Red de Protectores en RADIC.

Esta relación entre los protectores se debe a que entre ellos se implementa un mecanismo de *heartbeat/watchdog*, donde cada uno cuenta con un *watchdog* que espera por *heartbeats* de un vecino. El protector T1 de la Figura 3.2 dispara un *watchdog* y espera por un *heartbeat* del protector T2. Es preciso mencionar que los protectores también se comunican con los observadores en otros nodos y en el propio nodo.

Las tareas de protección que tienen a cargo cada uno de los protectores son las siguientes:

- Guardar *checkpoints* y logs de los mensajes recibidos por los procesos a los cuales protegen.
- Monitorear los vecinos para detectar fallos mediante el protocolo de *heartbeat/watchdog*.
- Reconfigurar los mecanismos de detección y protección luego de la ocurrencia de un fallo.
- Implementar el mecanismo de recuperación.

3.2.3. Controlador RADIC

Los componentes que fueron explicados en las dos sub-secciones anteriores no trabajan de manera independiente, sino que trabajan en conjunto para crear un sistema de

tolerancia a fallos transparente, descentralizado, escalable y flexible. En la Figura 3.3 puede observarse como cada proceso observador va ligado a un proceso de la aplicación, por ejemplo, el observador O1 está ligado al proceso P1, el O2 al proceso P2, y así sucesivamente. En el caso de los protectores puede verse que existe uno por nodo que pertenece al sistema, para el nodo N1 está el protector T1, para el N2 existe el T2, y así sucesivamente.

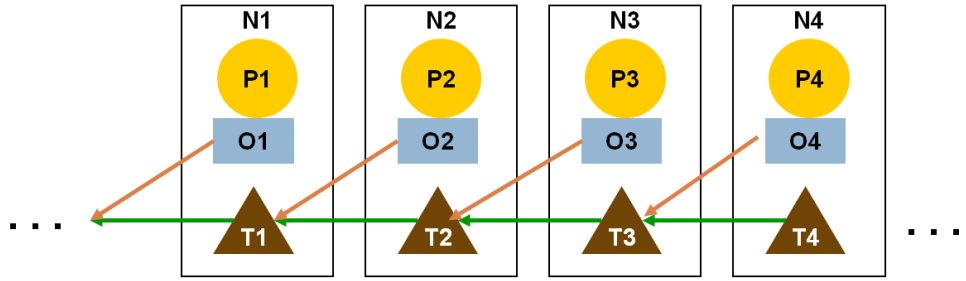

Figura 3.3. Aplicación paralela utilizando RADIC.

El controlador RADIC utiliza un protocolo de *rollback-recovery* con logs pesimistas basados en el receptor (2.5.1) y una estrategia de *checkpoint* no coordinado (2.4.1) para formar así una arquitectura de tolerancia a fallos descentralizada y escalable. Este protocolo no requiere de ningún tipo de sincronización global en los momentos de *checkpoint* así como en los momentos de recuperación, los únicos componentes envueltos en la recuperación son los procesos fallidos y sus respectivos protectores.

Además de las tareas de tolerancia a fallos, los observadores son responsables de mecanismo de paso de mensajes, esto requiere que se tenga una tabla que permita mapear la transmisión y la recepción de los mensajes entre cada proceso.

Para la reconfiguración del sistema existe una estructura de datos, la *radictable*, que se inicializa en todos los componentes de RADIC, esta tabla se actualiza utilizando un algoritmo determinista que permite garantizar la reconfiguración del sistema de un modo descentralizado

3.3. Funcionamiento de la Arquitectura RADIC

La arquitectura RADIC dentro de las tareas relacionadas a la tolerancia a fallos, implementa el mecanismo de paso de mensajes que es utilizado por la aplicación paralela.

El envío de mensajes entre procesos cuando se utiliza RADIC es hecho a través del observador, este se encarga de enviar el mensaje a través del canal de comunicación, y de la misma forma, es el observador del receptor el que recibe el mensaje, y realiza el log pesimista y luego pasa el mensaje al proceso aplicación.

Para realizar la labor explicada en el párrafo anterior, el observador utiliza una tabla de enrutamiento denominada *radictable.* En esta tabla se encuentra el mapeo entre el identificador de cada proceso (*Rank* MPI) y el identificador de cada nodo o su dirección. En la Tabla 3-2 puede observarse un ejemplo de *radictable* en el cual no solo se mapea lo que ya se ha mencionado, sino que también se lleva un reloj lógico de la cantidad de mensajes que ha enviado el observador a algún destino, así como los mensajes que ha recibido, para que así en el momento de la recuperación el proceso pueda saber cuántos mensajes debe releer del log antes de empezar a recibir mensajes nuevos.

Tabla 3-2. *Radictable* en un computador paralelo con N nodos.

Id del proceso	Dirección	Protector	Reloj lógico de mensajes enviados	Reloj lógico de mensajes recibidos
0	Nodo 0	Nodo N	2	3
1	Nodo 1	Nodo 0	0	0
2	Nodo 2	Nodo 1	1	1
…	…	…	…	…

3.3.1. Tareas de almacenamiento de estado

Los componentes de RADIC (protectores y observadores) colaboran para guardar el estado de una aplicación paralela, para lo cual es necesario contar con almacenamiento estable para guardar los *checkpoints* y logs requeridos para llevar a cabo el protocolo de *rollback-recovery*.

Como ya se ha mencionado cada observador se encarga de realizar *checkpoints* del proceso al que está ligado y se los envía al nodo en el que se encuentra su protector, tal y como se observa en Figura 3.4. El *checkpoint* es una operación atómica, mientras esta operación está en progreso el proceso no puede comunicarse. Esto debe ser distinguido de un fallo, por lo cual un intento de comunicación con un proceso que está realizando un *checkpoint* no debe ser considerado un fallo.

Los protectores operan como un sistema de almacenamiento confiable y distribuido. Es confiable porque los *checkpoints* y mensajes recibidos de un proceso no son almacenados en el nodo en el cual reside, sino en otro, entonces si falla el nodo donde éste reside, puede ser recuperado en otro lugar partiendo de la información almacenada.

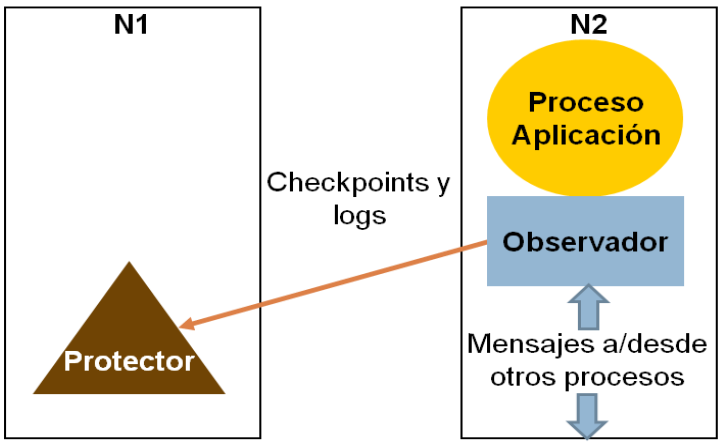

Figura 3.4. Relación observador-protector.

El log de mensajes pesimista permite actualizar y complementar la información almacenada por los *checkpoints* no coordinados que utiliza RADIC. La utilización de este log permite evitar el efecto dominó (Koren, y otros, 2007) y minimizar la cantidad de *checkpoints* que son mantenidos.

3.3.2. Detección de fallos

La detección de fallos es una tarea que realizan los protectores por medio del mecanismo de *Heartbeat/Watchdog* y los observadores detectan fallos cuando tratan de comunicarse con un proceso y el mismo no responde.

Los **protectores** realizan dos tareas con respecto a la detección, una es un monitoreo pasivo (*watchdog*) y la otra un monitoreo activo (*heartbeat*). De acuerdo a lo explicado puede concluirse que cada protector está compuesto por estas dos partes, las cuales pueden observarse en la Figura 3.5. El ciclo *heartbeat/watchdog* determina que tan rápido detecta un protector un fallo de su vecino, obviamente aquí hay una relación de compromiso, ya que un ciclo corto reducirá el tiempo de detección pero también incrementará el *overhead* en el canal de comunicación.

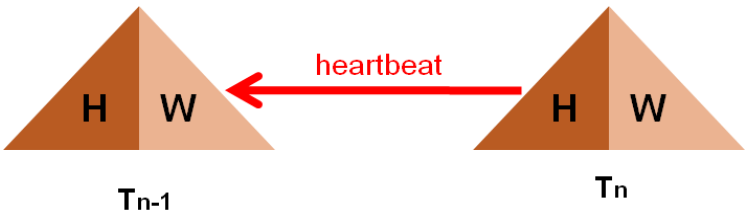

Figura 3.5. Heartbeat y Watchdog en protectores.

El *watchdog* espera por el *hearbeat* de su vecino, si este expira trata de contactar al protector del cual esperaba el *heartbeat* para confirmar el fallo, y si lo confirma inicia la recuperación y reconfigura el sistema de detección.

Los **observadores** también son capaces de detectar fallos cuando falla la comunicación con otros procesos, o cuando la comunicación con su protector falla. Ya que el observador solo se comunica con su protector para transferir *checkpoints* y logs, es necesaria otra forma para verificar de manera rápida si el protector falla. Esto está contemplado en RADIC, dado que se implementa una comunicación entre el observador y el protector local (el que se ejecuta en el mismo nodo). Cuando un protector detecta un fallo en otro nodo, este protector avisa a los observadores en su nodo, y cuando los observadores reciben este mensaje establecen un nuevo protector y realiza un *checkpoint.*

Un observador no puede confirmar un fallo (diagnóstico) solamente por un fallo de comunicación, ya que si el proceso con el que trata de comunicarse está realizando un *checkpoint,* éste no le responderá ya que tiene todas las comunicaciones cerradas, por lo cual el proceso que trató de comunicarse debe contactar con el protector de dicho proceso destino, residente en el destino usando su *radictable* y de esta forma verifica si el fallo de

comunicación corresponde realmente a un fallo. Si realmente ha fallado, el protector inicia la fase de recuperación

3.3.3. Recuperación

Cuando los protectores y/o observadores detectan un fallo, los dos actúan en el proceso de recuperación para restablecer el estado consistente de la aplicación paralela y la estructura del controlador RADIC.

La forma en la cual trabajan ambos componentes en la recuperación se detalla en la Tabla 3-3.

Tabla 3-3. Actividades del protector y del observador en la recuperación.

Protector	Observador
Emisor del *Heartbeat*: • Establece un nuevo destino de *heartbeat*. • Restablecer el mecanismo de *heartbeat*. • Comandar a los observadores locales que hagan *checkpoint*.	Sobreviviente: • Establecer un nuevo protector (actualizar la *radictable*). • Realizar *checkpoint*.
Watchdog: • Esperar por un nuevo emisor de *heartbeat*. • Restablecer el mecanismo de *watchdog*. • Recuperar al proceso fallado.	Recuperado: • Establecer un nuevo protector (actualiza la *radictable*). • Reejecutar. • Releer los mensajes desde el log. • Realizar un nuevo *checkpoint* • Copiar el *checkpoint* y log actuales al nuevo protector.

Cuando ocurre un fallo el sistema se comporta de acuerdo a lo que se puede observar en la Figura 3.6 y si no existen nodos extras (nodos *spare*), queda como la

configuración que se puede ver en la Figura 3.6 d), donde observamos que el nodo N2 está sobrecargado con dos procesos.

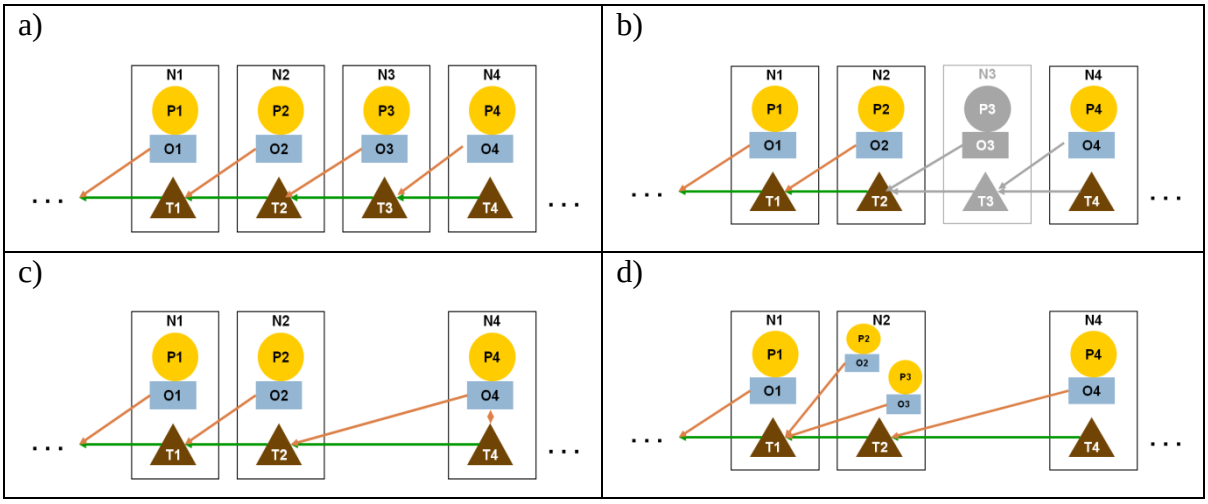

Figura 3.6. a) Ejecución normal sin fallos. b) Fallo en el nodo N3. c) Restablecimiento del *watchdog* y seteo del nuevo protector para P4. d) Restauración del proceso P3 en N2.

El proceso fallado es restaurado en su protector, e inmediatamente luego de que la recuperación culmina cada observador se conecta a un nuevo protector, y el observador recuperado obtiene la información acerca de su nuevo protector del protector residente en el nodo local.

Obviamente esto trae consigo un desbalanceo de carga, y por supuesto trae consigo un impacto en las prestaciones, dado que un nodo quedará sobrecargado. Este problema es el que busca ser solucionado en este trabajo mediante la utilización de nodos *Spare* o de repuesto que permita mantener las prestaciones de una aplicación inclusive en presencia de fallos, esto será explicado en la sección en la cual se habla de la propuesta de este trabajo.

3.3.4. Enmascaramiento de fallos

El enmascaramiento de fallos es una tarea que está a cargo del observador. En caso de fallos el observador asegura que los procesos sigan comunicándose a través del mecanismo de paso de mensajes, por este motivo es que cada observador se encarga de manejar todos los mensajes enviados y recibidos del proceso al que observan.

Cuando ocurre un fallo el protector lo detecta e inicia el proceso de recuperación del proceso fallado, si está en el modo básico esta recuperación la realiza en su mismo nodo, de acuerdo a esto, todos los procesos que traten de comunicarse con el proceso fallado detectarán un fallo de comunicación. Entonces, cuando el proceso que quiere comunicarse falla, debe empezar a buscar la nueva ubicación de este proceso y el procedimiento que se sigue es el descripto la Figura 3.7.

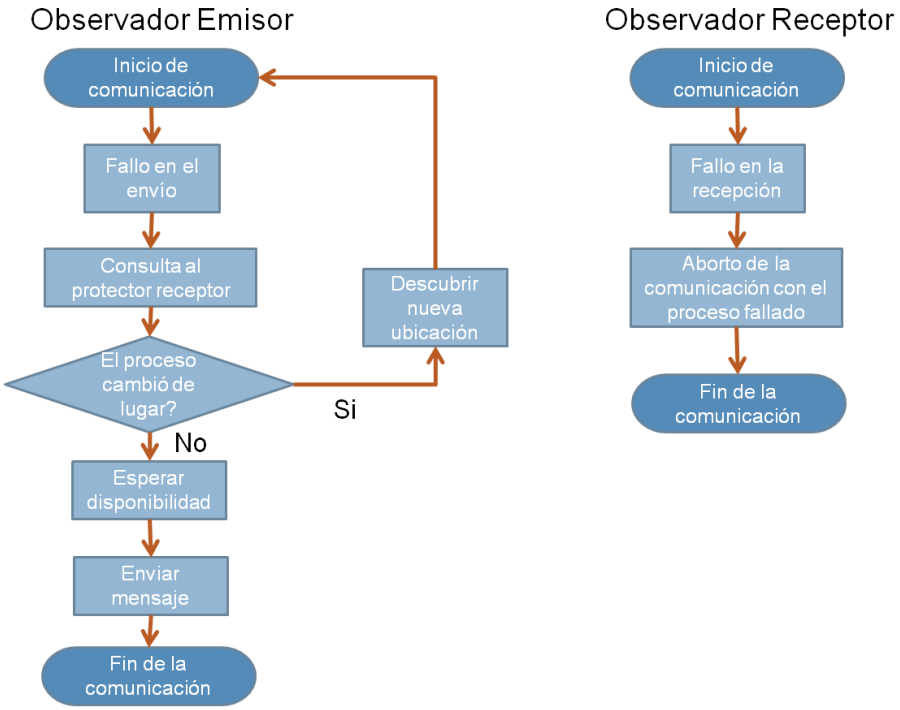

Figura 3.7. Algoritmos de detección para el observador emisor y para el receptor.

Si se considera que el proceso que falló se restaure en su protector de acuerdo a lo que se observa en la Figura 3.6 d) y cuando alguien trate de comunicarse con él obtendrá un error de comunicación y consultará al protector T2 por el estado del proceso P3, y entonces T2 informará que ya no es responsable de P3 porque ahora T1 se ocupa de la protección de él.

Para saber quién es el protector actual de P3, el observador emisor usa su *radictable* para seguir la cadena de protección. El emisor sabe que T2 ya no está protegiendo a P3, entonces el probable protector de P3 se busca utilizando un algoritmo determinista en la *radictable*. Cuando el emisor sabe quién es el probable protector de P3 lo contacta y pregunta por el estado del proceso. Si el protector confirma la ubicación de P3, el emisor

actualiza su *radictable* y reintenta de la comunicación, en caso contrario sigue buscando al protector.

Cuando el proceso se restaura desde un *checkpoint* anterior y resume la ejecución desde ese punto, y si el proceso recibió mensajes desde que ha realizado este último *checkpoint*, dichos mensajes se encontrarán en el log. El observador utiliza este log para dar a la aplicación los mensajes a medida que los vaya requiriendo.

Si el proceso restaurado reenvía mensajes durante su recuperación, el observador destino descartará estos mensajes repetidos, este mecanismo es simple gracias a que se utiliza un reloj lógico, tal y como se ve en la Tabla 3-2.

Una vez que se termina de reejecutar (log de mensajes consumido), hace un nuevo checkpoint en su protector. Es importante resaltar que todos estos procesos deben ser totalmente transparente a MPI, ya que como hemos visto, considera que tiene una transmisión de mensajes fiable y si detecta un error en la comunicación provocar una parada por fallo (*fail stop*).

Lograr que sea totalmente transparente dependerá de la implementación de la librería MPI que se esté utilizando

3.4. Otros niveles de protección de la Arquitectura RADIC

Con la arquitectura básica propuesta se puede tolerar 1 fallo, si ocurre un segundo fallo mientras se está ejecutando una de las operaciones atómicas de recuperación y reconfiguración, RADIC dejará de funcionar. Si se quiere un grado de protección mayor, se debe disponer de más de una copia de la información redundante de protección (*checkpoint* y log).

Hemos analizado que cuando se produce un fallo, se pierde capacidad de cómputo y se produce un desbalanceo de carga, esta pérdida de capacidad de cómputo y desbalanceo se irá aumentando si a lo largo de la ejecución ocurren nuevos fallos, por lo que puede

llegar a tener una gran repercusión en el tiempo de ejecución de la aplicación, incluso como para que RADIC provoque una parada segura por pérdida de capacidad computacional o porque RADIC requiere un mínimo de 3 nodos para funcionar por lo tanto si un observador, en su radictable detecta que sólo hay 2 nodos debe forzar un checkpoint y parar.

En el nivel básico de RADIC que ha sido explicado en este capítulo, cuando un fallo ocurre, se continúa la ejecución con un nodo menos, manteniendo constante el número de procesos, pero ocurre una degradación en las prestaciones.

En el nivel de protección de RADIC denominado de “Disponibilidad Continua” existen nodos libres (*spare*) que son utilizados en caso de fallos y permiten por lo tanto la reparación en caliente, utilizando esta alternativa se permite una capacidad computacional constante. La utilización de nodos *spare* permite realizar mantenimiento preventivo provocando falsos fallos para mantener o actualizar nodos, para esto es necesario utilizar un inyector de fallos.

En resumen, si se quiere mantener la capacidad de cómputo es necesario incluir la gestión de nodos pasivos tal y como se explica en el Capítulo 5.

Capítulo 4.
Estado del arte

4.1. Introducción

De acuerdo a lo que ya se ha mencionado anteriormente uno de los protocolos más utilizados para proveer tolerancia a fallos en los sistemas es el de *rollback-recovery*. En este protocolo se utilizan mecanismos de *checkpoint/restart*, donde el estado de una aplicación paralela es salvado para una posible restauración posterior.

Existen varias librerías de *checkpoint-restart* entre las cuales se pueden citar: *libckpt* (Plank, y otros, 1995), *Condor checkpoint library* (Litzkow, y otros, 1997), *BLCR (Berkeley Lab's Checkpoint/Restart)* (Duell, 2003) y una librería que trabaja en espacio de usuario llamada *DMTCP (Distributed MultiThreaded Checkpointing)* (Ansel, y otros, 2009). Estos procedimientos difieren en como implementan el mecanismo de *checkpoint*, cuanto del estado del proceso es preservado, como es guardado, etc.

La mayoría de las estrategias de *checkpoint/restart* distribuidas requieren la coordinación entre los procesos individuales para crear estados consistentes para la aplicación paralela. La mayoría de las técnicas de *checkpoint* caen en una de las tres categorías: coordinados, no coordinados o inducidos por mensajes.

En la Figura 4.1, extraída de (Cappello, 2008) se citan algunas implementaciones de modelos de tolerancia a fallos clasificados de acuerdo al nivel en el cual están implementados, de acuerdo a su forma de operación, y a las políticas de *checkpoint* y log que son utilizadas.

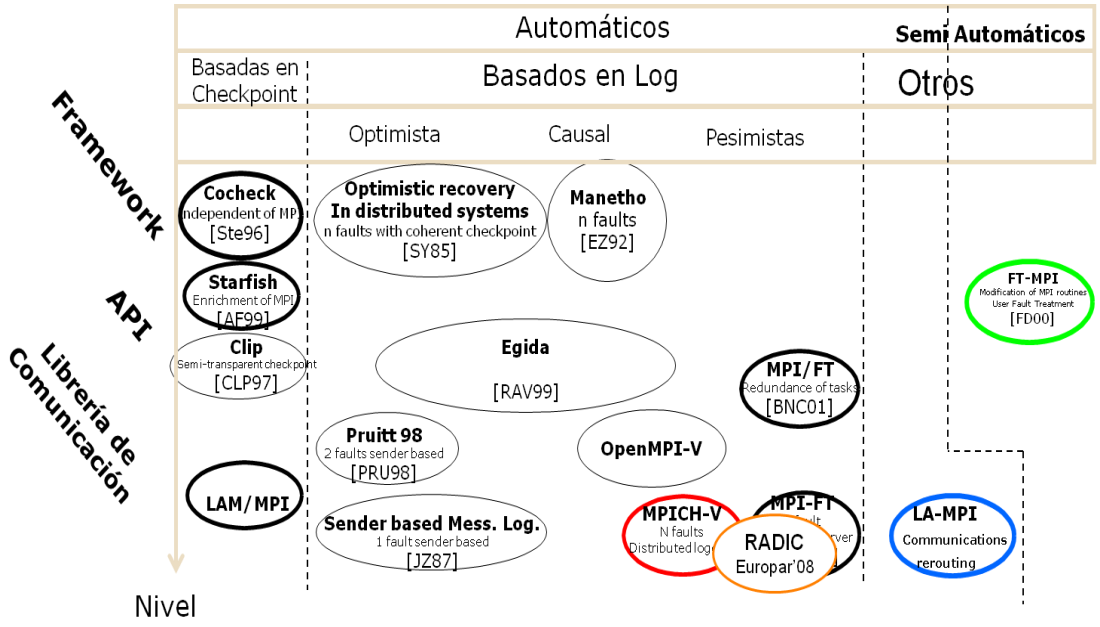

Figura 4.1. Implementaciones de sistemas de tolerancia a fallos en librearías MPI.

Algunas librerías han intentado integrar técnicas de tolerancia a fallos en sus implementaciones utilizando una de las alternativas mencionadas anteriormente. En la siguiente sección se clasifican algunas implementaciones de protocolos de tolerancia a fallos dentro de librerías de paso de mensajes.

4.2. Tolerancia a fallos en implementaciones de paso de mensajes basadas en MPICH

En **MPICH-V** (Bosilca, y otros, 2002) se utiliza un protocolo de *checkpoint-restart* no coordinado en conjunto con log de mensajes para preservar el estado del proceso y recuperar un proceso fallado en caso de fallos automáticamente. Esta implementación se ha diseñado para computación a gran escala utilizando redes heterogéneas.

Como ya se mencionó se utiliza un mecanismo de *checkpoint/restart* no coordinado y se lo complementa con log de mensajes pesimistas remotos. **MPICH-V1** (Bosilca, y otros, 2002) está bien diseñado para grids y computación global dado que pueden soportar una tasa alta de fallos, pero requiere un alto ancho de banda para lograr buenas prestaciones.

MPICH-V2 (Bouteiller, y otros, 2003) está diseñado para redes homogéneas y para computación a gran escala. A diferencia de la versión 1, MPICH-V2 requiere un pequeño número de componentes estables para lograr buenas prestaciones. Se utiliza un protocolo de *checkpoint* no coordinado junto con un log de mensajes pesimistas basados en el receptor.

MPICH-VCL está diseñado para obtener bajas latencias. Se utiliza un mecanismo de *checkpoint* no coordinado basado en el algoritmo de Chandy-Lamport (Chandy, y otros, 1985) para disminuir el *overhead* en la ejecución sin fallos. Cuando ocurre un fallo todos los nodos son reiniciados, inclusive aquellos que no han fallado, esto lo hace menos resistente a fallos que los basados en log, y está diseñado básicamente para clusters de media escala.

Una de las implementaciones más recientes es **MPICH-PCL** (Buntinas, y otros, 2008) que siguen la misma arquitectura que fue implementada en las demás versiones de MPICH, por lo tanto requiere elementos dedicados para realizar *checkpoints* y el manejo de mensajes.

En este proyecto se utiliza un *logger* de eventos y un servidor de procesos que hacen *checkpoint* que aligera el *overhead* de sus técnicas de tolerancia a fallos. En esta implementación se utiliza el sistema de *checkpoint/restart* Condor (Litzkow, y otros, 1997).

4.3. Tolerancia a fallos en implementaciones de paso de mensajes basadas en Open MPI

A continuación se describen las implementaciones de mecanismos de tolerancia a fallos dentro de la librería Open MPI (Fialho, 2008). Además se mencionará el mecanismo de tolerancia a fallos que actualmente está implementado en la versión oficial de Open MPI.

Algunas técnicas integran tolerancia a fallos interactiva (FT-MPI (Fagg, y otros, 2000)) así como recuperación de fallos de red (LA-MPI (Graham, y otros, 2003)).

FT-MPI busca ofrecer una API de MPI para el usuario final para manejar la librería de comunicación, y éste implementa toda la especificación MPI-1.2, algunas partes del estándar MPI-2 y extiende alguna semántica de MPI para dar a la aplicación la posibilidad de recuperar un proceso fallado.

LA-MPI persigue dos objetivos principales que son la tolerancia a fallos en las redes de computadores y obtener altas prestaciones. La tolerancia a fallos en la red es lograda gracias a la implementación de un mecanismo de *checksum* con retransmisión. Esto es utilizado porque es más ligero que el protocolo clásico de TCP/IP.

El protocolo permite la utilización de caminos redundantes para obtener anchos de banda altos, enviando por porciones los datos en paralela.

LAM/MPI (Sankaran, y otros, 2003) modulariza su enfoque de *checkpoint/restart* y esto le permite utilizar múltiples sistemas de *checkpoint/restart* en su implementación. Esta implementación soporta comunicación sobre TCP y Myrinet, además soporta BLCR y SELF como sistemas de *checkpoint/restart* pero solamente soporta el protocolo de *checkpoint* coordinado por lo cual solamente puede realizar *checkpoints/restarts* de la aplicación paralela entera. LAM/MPI también requiere que el sistema de *checkpoint/restart* provea un hilo de comunicación con *mpirun* para iniciar el *checkpoint* del proceso paralelo. Las herramientas de línea de comandos permiten la realización de *checkpoints* asíncronos pero no provee ninguna API para la realización de *checkpoint/restart* síncronos.

4.4. Otras implementaciones

Otras implementaciones de MPI integran técnicas de *checkpoint/restart* para guardar y recuperar el estado de la aplicación paralela.

Starfish (Agbaria, y otros, 1999) provee soporte para protocolos de checkpoint coordinados y no coordinados. *Starfish* utiliza su propio sistema distribuido para proveer un mecanismo de *checkpoint*.

Egida (Rao, y otros, 1999) provee una gramática para experimentar con nuevos protocolos de log de mensajes. Esta gramática es bastante simple y permite expresar protocolos de *rollback/recovery*.

CoCheck (Stellner, 1996) utiliza el sistema de checkpoint/restart de Condor (Litzkow, y otros, 1997) y protocolo de coordinación de checkpoint que les permite observar los canales de la red antes de realizar un checkpoint.

4.5. Checkpoint/restart en Open MPI

El diseño utilizado en la implementación del protocolo de *checkpoint/restart* que se encuentra en Open MPI (Hursey, y otros, 2007) trata de combinar las mejores cualidades de las implementaciones descriptas más arriba. Dado que Open MPI consta de una arquitectura modular se han podido agregar módulos de tolerancia a fallos que pueden ser utilizados o modificados de acuerdo a las necesidades.

En (Hursey, y otros, 2007) se propone un mecanismo de *checkpoint/restart* coordinado, los *checkpoints* son tomados por cada proceso de manera distribuida, pero es necesario que se coordinen para crear un estado global consistente, lo cual requiere obviamente detener la ejecución de todos los procesos a la vez.

Existe un *Snapshot Coordinator* que es quien recibe la petición de realizar un *checkpoint* y avisa a los procesos que componen la aplicación paralela para que detenga su ejecución y realicen un *checkpoint*.

El *Snapshot Coordinator* está compuesto por tres subcoordinadores:

- *Global Coordinator:* este forma parte del proceso *mpirun* (proceso padre de los procesos de la aplicación), es responsable de interactuar con las herramientas de línea de comandos para generar una referencia global que

representa la agregación de todos los archivos remotos en una imagen global y se encarga de monitorizar el progreso del pedido de *checkpoint*.

- *Local Coordinator:* este coordinador forma parte del demonio que reside en cada nodo denominado ORTED (*Open Runtime Environment Daemon*) y trabaja en conjunto con el *global coordinator* para iniciar el *checkpoint* de un proceso residente en su nodo y para mover los archivos al *global coordinator* para guardar una imagen global.
- *Application Coordinator:* este coordinador es parte de cada proceso de la aplicación en el sistema distribuido y es responsable de iniciar el *checkpoint* de cada proceso individualmente.

Cada proceso además cuenta con un *thread* de notificación de *checkpoints* que está en constante espera por una solicitud de *checkpoint* y cuando recibe una petición de este tipo inicia la comunicación entre cada uno de los coordinadores para llevar a cabo el *checkpoint* global.

Además de las capacidades de tolerancia a fallos que se agregan a Open MPI se agregan capacidades como la migración de procesos en frio, log de mensajes y recuperación transparente.

La desventaja de esta implementación es que utiliza *checkpoints* coordinados lo cual hace que la aplicación paralela se detenga por completo y guarde un estado global consistente, además en caso de que un proceso falle, todos los demás procesos deben volver atrás hasta el último *checkpoint*.

Capítulo 5.
Propuesta e Implementación

5.1. Introducción

Tal y como hemos visto, los fallos que ocurren afectan gravemente a las prestaciones de una aplicación paralela, si es que no acaban con ellas. Por esto es necesario contar con elementos redundantes para poder reemplazar a los dañados y de esta forma tratar de que las prestaciones iniciales se mantengan. Se busca que los nodos de repuesto (*spares*) se integren en la arquitectura de tolerancia a fallos de manera transparente y automática.

Lo explicado anteriormente da la posibilidad de intercambiar nodos en caliente (*hot swap*), es decir mientras que la aplicación sigue funcionando. Los beneficios del intercambio en caliente son evidentes, dado que permite reconocer componentes fallados y reemplazarlos sin comprometer la disponibilidad del sistema, y buscando afectar al mínimo las prestaciones.

5.2. Degradación de prestaciones

Considerando la arquitectura RADIC (Duarte, 2007) y la ocurrencia de un fallo, el sistema se comportará de acuerdo a lo que se observado en la Figura 3.6. Este comportamiento es el básico de RADIC, que tolera los fallos en la aplicación paralela, pero permite la degradación del sistema hasta un nivel aceptable. Si este nivel de degradación es excedido, se detiene la ejecución y se reinicia la aplicación con la configuración inicial.

En aplicaciones paralelas las prestaciones son cruciales, por lo cual es necesario algún mecanismo de tolerancia a fallos que no solo introduzca un *overhead* reducido en una ejecución sin fallos sino que en presencia de fallos pueda mantener la sintonización o nivel de carga inicial para cada uno de los nodos que conforman la aplicación.

Como ya se ha mencionado, RADIC es un sistema de tolerancia a fallos flexible y en (Santos, y otros, 2008) se ha propuesto un nuevo nivel de protección en RADIC que permite restaurar la configuración del sistema evitando la perdida permanente de nodos dentro del sistema paralelo.

La propuesta para evitar la pérdida de prestaciones consiste básicamente en la introducción de nodos *Spare* que se ocupen de adoptar a los procesos de un nodo fallado restableciendo de esta forma la configuración del sistema.

En la Figura 5.1 puede verse cómo se comporta una aplicación paralela en cuanto a las prestaciones que ofrece cuando ocurre un fallo. Si el sistema no cuenta con tolerancia a fallos el comportamiento por defecto de la librería de paso de mensajes es detener la ejecución en caso de fallos (*fail-stop*). Si se utiliza RADIC sin nodos *spare* las prestaciones del programa paralelo se reducen dado que un nodo queda sobrecargado y la aplicación se desbalancea. Si se utilizan nodos *spare* de acuerdo a lo propuesto en (Santos, y otros, 2008) las prestaciones de la aplicación se degradan momentáneamente hasta que el proceso se recupere en el nodo *spare* y de esta forma se restaura la configuración inicial y las prestaciones son mantenidas.

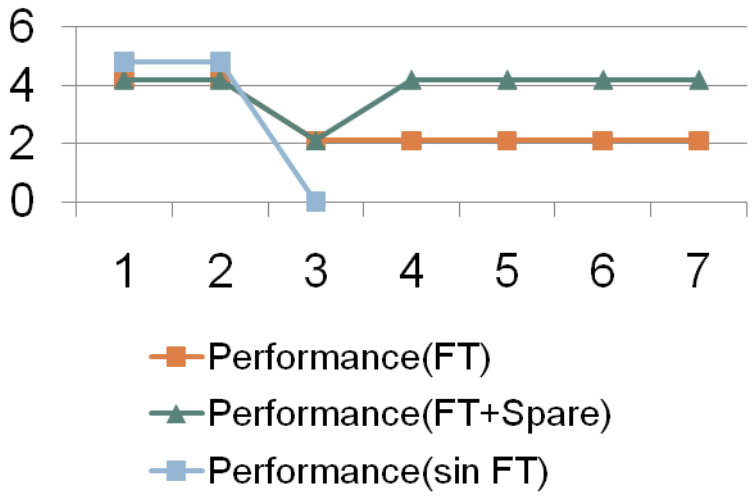

Figura 5.1. Modelo de comportamiento prestacional de una aplicación paralela en caso de fallos.

En la siguiente sección se tratará mínimamente sobre la implementación de RADIC dentro de Open MPI y luego sobre la propuesta de este trabajo, que consiste en la incorporación de nodos *spare* dentro de dicha implementación.

5.3. Nodos Spare en RADIC

La inclusión de nodos *spare* dentro de la arquitectura RADIC ya fue planteada en (Santos, y otros, 2008). En la Figura 3.6 se puede observar el comportamiento por defecto de la arquitectura RADIC cuando ocurre un fallo, y obviamente tal y como se observa el trabajo queda desbalanceado ya que el sistema tiene un nodo menos (pérdida de capacidad computacional), se produce un cambio de *mapping* de la aplicación y un desbalanceo de la carga (un nodo queda con más trabajo) y por lo tanto las prestaciones se degradan y el sistema avanza de acuerdo al ritmo del más lento.

Tabla 5-1. *Sparetable*

Id del *spare*	Dirección	Observadores
0	Nodo 5	0
1	Nodo 6	0
2	Nodo 7	1
…	…	…

Si son utilizados nodos *spare* se mantiene la capacidad computacional y se puede respetar la distribución inicial de carga, restableciendo una configuración similar a la inicial.

En la Figura 5.2 se puede observar el operar de RADIC cuando existen nodos *spare*, en la Figura 5.2 a) se muestra el funcionamiento de una aplicación paralela con un nodo *spare* NS. En la Figura 5.2 b) ocurre un fallo en el nodo N3 y en la Figura 5.2 c) el protector T2 que protege a P3 consulta una tabla llamada *sparetable* (Tabla 5-1) para saber la dirección de los nodos *spares* y la disponibilidad del mismo, es decir, saber si ya tiene uno o más observadores (o procesos) en él. En la Figura 5.2 d) T2 transfiere el *checkpoint* del proceso P3 a NS y por último en la Figura 5.2 e) se reinicia el proceso y se restablece la cadena de protección.

Si el sistema paralelo se queda sin nodos *spare* se utiliza el mecanismo por defecto de RADIC que fue descripto en la Figura 3.6, restaurando al proceso o procesos fallados en el protector del mismo.

En el procedimiento descripto existen varios problemas que deben ser subsanados, uno de ellos es lo que ocurre mientras el proceso está en trámites de restauración, dado que si otro proceso trata de comunicarse con él, este obtendrá un fallo de comunicación y se comunicará con el protector del proceso que ha fallado, y este le dirá que espere para comunicarse de nuevo con él en el lugar en el cual se restaurará (protector o *spare*).

El restablecimiento de la cadena de protección se realiza cuando el observador cuyo protector ha fallado busca un nuevo protector y consulta al siguiente en la cadena de protección de acuerdo a su *radictable* y este le indica quien será su nuevo protector (el *spare* o él mismo).

En este momento el nodo que ha fallado puede ser reparado, una vez reparado se puede volver a conectar al sistema como un nuevo *spare*.

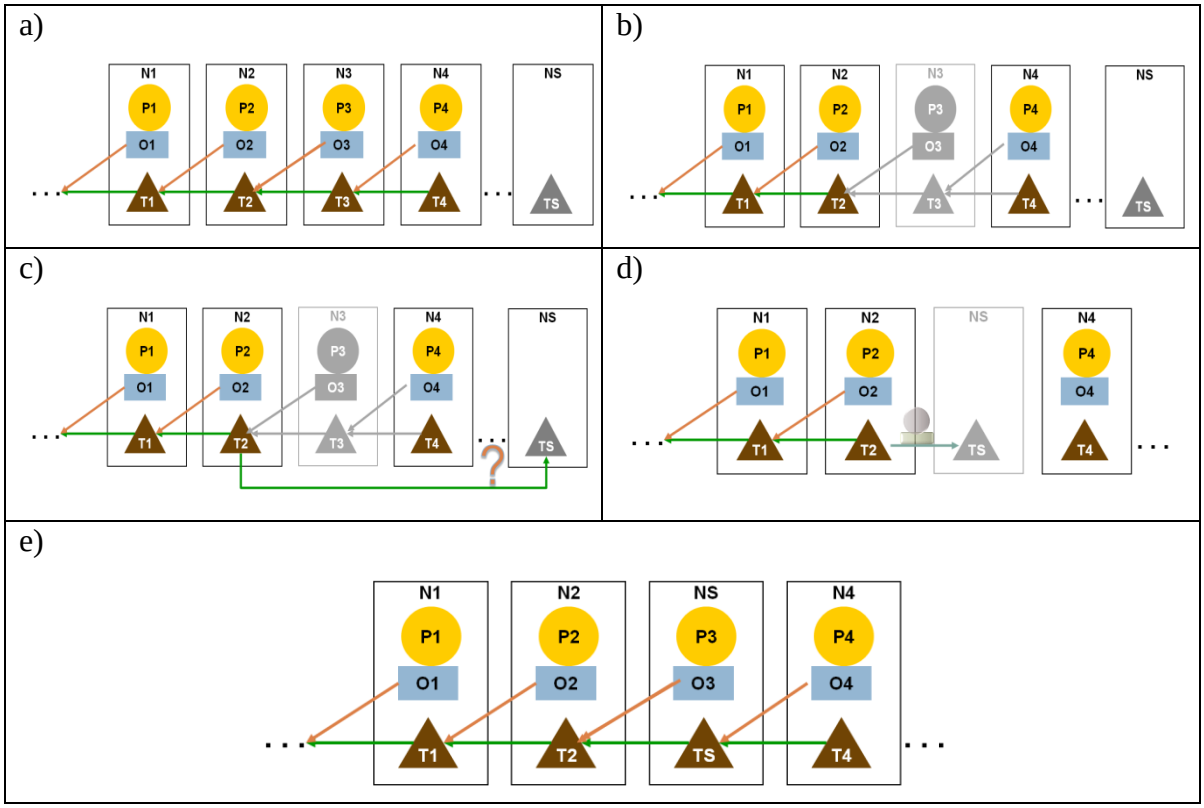

Figura 5.2. RADIC con nodos *Spares*. a) Ejecución antes del fallo con un nodo *spare*. b) Fallo en el nodo 3. c) El protector T2 de P3 consulta su *Sparetable* y al NS por la disponibilidad. d) T2 transfiere el *checkpoint* de P3 a NS. e) TS reinicia a P3 y se restablecen las comunicaciones y cadena de protección.

Además de lo explicado más arriba, son necesarias políticas para gestionar los nodos *spare* y es necesario además responder a ciertas preguntas:

- ¿Cómo se dan a conocer los nodos *spare*?:
    - Pueden ser conocidos desde el arranque de la aplicación paralela, por lo cual cada protector ya cuenta desde el inicio con la *sparetable* cargada con los nodos *spare* y así pueden utilizarlos en cualquier momento. Si un nodo *spare* es ocupado más adelante por un proceso, los demás actualizarán la condición del *spare* en su *sparetable* ya sea bajo demanda o por un mecanismo basado en un envío oportunista mediante mensajes tipo *tokens* entre protectores (Jalote, 1994).
    - *Hot Insertion:* los nodos *spare* aquí son añadidos en caliente, una vez que la aplicación ya está en marcha, también pueden ser insertados nodos *spare* cuando el nodo es introducido, si es así, necesariamente debe anunciarse a los protectores por medio de *tokens* (Jalote, 1994) para que estos lo actualicen su *sparetable*. El token lo activa el nodo *spare* y debe circular por todos los nodos con un algoritmo determinista de paso entre nodos, y se para cuando el *token* llegue de nuevo al nodo *spare*.
- Estados de los *Spare*:
    - Pasivo: el *spare* no forma parte de la aplicación paralela y se encuentra esperando como se ve en la Figura 5.2 a).
    - Activo: el *spare* forma parte de la aplicación, y tiene en su nodo al menos un proceso residente (Figura 5.2 e).
- Situaciones de competencia:
    - En caso de que más de un protector consulte su *sparetable* porque han ocurrido fallos en sus respectivos procesos protegidos y ambos consultan al *spare* (Figura 5.2 c)) para ver si está disponible, y el *spare* solamente responde de manera positiva a la primera petición recibida, es decir, no existe ningún tipo de elitismo.
- ¿Qué ocurre si se acaban los nodos *Spare*?
    - Si se acaban los nodos *spare* se utiliza el mecanismo básico de RADIC que fue mostrado en la Figura 3.6.

- ¿Qué ocurre con los nodos que son recuperados?
    - Los nodos que habían fallado y son reparados pueden ser reintroducidos como *spares* mediante *hot insertion*.
- ¿Qué ocurre si hay un fallo antes de que el token haya terminado de circular?
    - Si el protector que necesita al *spare*, ya conoce su existencia, lo utiliza.
    - Si el protector que necesita el *spare* no conoce su existencia, utiliza el mecanismo básico y cuando el protector del nodo sobrecargado vea que existe un spare, en el siguiente *checkpoint* lo utilizará.
- Enmascaramiento de Fallos:
    - La inserción de nodos *spare* dentro de la arquitectura RADIC que no tiene información centralizada introduce la necesidad de definir un algoritmo determinista para que cuando ocurra un fallo y se restaura en un *spare* y algún proceso trata de comunicarse consultará a su protector para ubicarlo y si no se encuentra restaurado ahí debe buscar (utilizando el algoritmo definido) al proceso restaurado recorriendo la *sparetable*. Dado que solo los protectores disponen de la *sparetable* debe existir más comunicación entre los observadores y los protectores que residen en el mismo nodo.

Los problemas de implementación de las políticas descriptas y sus soluciones son explicados en mayor detalle en las siguientes secciones.

5.4. Implementación en Open MPI

En esta sección se describen mínimos detalles de la implementación de RADIC dentro de Open MPI y también de la implementación de la propuesta en la misma librería. Mayores detalles de la inclusión de RADIC dentro de esta librería son descriptos en (Fialho, 2008) por lo cual esta sección solo se centra en lo que corresponde a la inclusión de nodos *spare*.

En primer lugar se destacan las características generales de Open MPI (Indiana University and Partners, 2011) para luego describir la implementación realizada y las dificultades enfrentadas.

5.3.1. Arquitectura Open MPI

Open MPI utiliza una arquitectura modular denominada MCA (*Modular Component Architecture*) que cuenta con tres capas funcionales que son:

a. *Root*: el cual provee básicamente todas las funciones MCA.
b. *Frameworks*: que específica las interfaces funcionales a los servicios específicos (Ej. Transporte de mensajes).
c. *Components*: que son las implementaciones de cada *framework* en particular (Ej. TCP).

Los *frameworks* están divididos en tres sub-capas que se complementan entre sí y son:

a. *Open MPI (OMPI)*: provee la API básica para la escritura de aplicaciones paralelas.
b. *Open Run-Time Environment (ORTE):* provee el ambiente de ejecución para las aplicaciones paralelos y otros servicios para la capa superior.
c. *Open Portable Access Layer (OPAL):* es la capa que provee abstracción a algunas funciones del sistema operativo.

En la Figura 5.3 a) se muestra la arquitectura básica de Open MPI y la interrelación entre las capas.

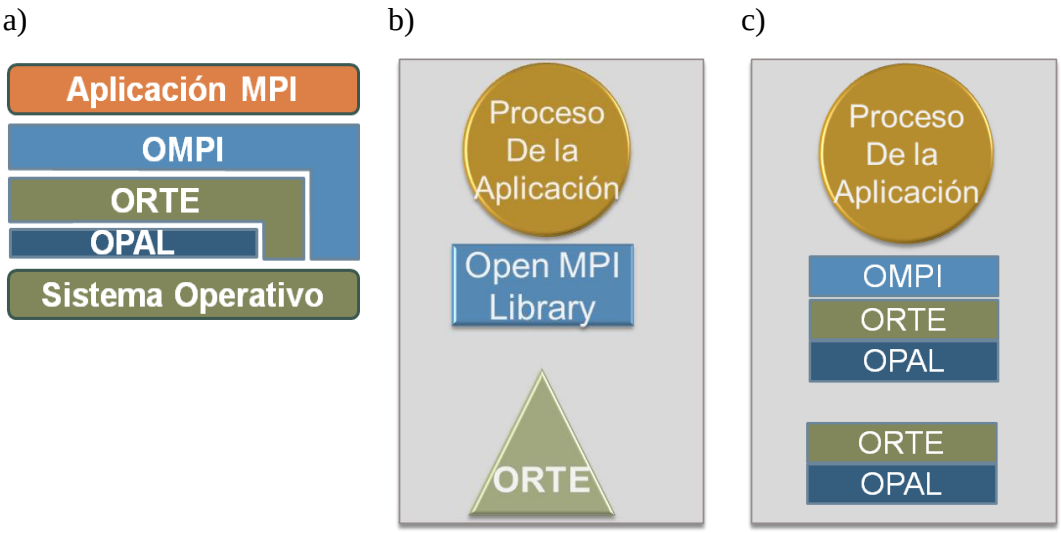

Figura 5.3. a) Arquitectura MCA de Open MPI. b) Nodo Open MPI con una aplicación. c) Estructura del demonio ORTE y de la librería Open MPI.

Cuando una aplicación paralela es lanzada, se instancia un demonio ORTE en cada uno de los nodos que componen la aplicación paralela, estos se comunican entre sí para crear el entorno paralelo en el cual residirá la aplicación. Una vez que este paso es realizado las aplicaciones paralelas son lanzadas y empieza la ejecución.

En la Figura 5.3 b) se muestra un modelo de un nodo en el cual se ejecuta una aplicación Open MPI y en la Figura 5.3 c) se muestra la composición del demonio ORTE que tiene además un componente del *framework* OPAL.

El *framework* OMPI provee una pila de tres *sub-frameworks* que son:

- *Point-to-point Management Layer (PML)*: es el framework que permite encapsular protocolos de transmisión.
- *Byte Transfer Layer (BTL)*: implementa los drivers de comunicación.
- *BTL Management Layer (BML)*: es el *framework* que actúa de contenedor para los drivers implementados en el BTL.

Además como ya se ha destacado anteriormente Open MPI implementa un mecanismo de tolerancia a fallos no transparente, dado que es manual (depende del usuario) y utiliza una herramienta que permite enviar solicitudes de *checkpoint/restart* al *Snapshot Coordinator (SnapC)* que es el encargado de iniciar el protocolo coordinado de *checkpoint* de Open MPI. Los *checkpoints* generados son transferidos mediante el

framework "*Filem Manager*" (*FileM*) y todas estas comunicaciones se realizan utilizando el *framework* "*Out of Band*" (*OOB*).

5.3.2. RADIC en Open MPI

Se ha optado por implementar RADIC dentro de la librería Open MPI dado que es una librería bastante utilizada en el mundo científico y además que gracias a su arquitectura modular la implementación e inclusión de RADIC en la librería es un poco más sencilla. Un estudio profundo de esta implementación se ha realizado en (Fialho, 2008) por lo cual esta sección solo se encarga de destacar aspectos relevantes del tema.

El primer prototipo de RADIC fue llamado RADICMPI (Duarte, y otros, 2006), la cual fue desarrollada como una librería de paso de mensajes pero bastante limitada, dado que no contaba con todas las operaciones con las que cuentan las librerías de paso de mensajes más potentes.

En los siguientes párrafos se comentan algunos puntos importantes de la implementación que se denomina RADIC-OMPI (Fialho, 2008).

El observador debido a su comportamiento ha sido integrado como un componente **PML** de Open MPI y de esta manera se asegura la existencia de un observador por cada proceso de aplicación.

El protector en contraposición al observador debe existir en cada nodo, por lo cual fue integrado dentro del **demonio ORTE**. Para definir el mapeo inicial de observador/protector se utiliza un algoritmo simple, donde el observador calcula que su protector es el siguiente nodo lógico, y si fuese el último nodo establece al primero como su protector.

En lo que se refiere a los mecanismos de tolerancia a fallos y su implementación en Open MPI pueden hacerse las siguientes observaciones:

- ***Checkpoints no coordinados*:** para la realización de cada *checkpoint* los canales de comunicación que utiliza MPI deben ser cerrados para garantizar que no existen mensajes en tránsito. El *checkpoint* es iniciado a través de una función *callback* que es disparada por un temporizador (intervalo de *checkpoint*), esta función se encarga de verificar que no existen comunicaciones. Una vez que se ha creado el *checkpoint* este es transferido utilizando el *framework* ***FileM*** y cuando termina esta transmisión el observador se encarga permitir de vuelta la transmisión de mensajes, y quien recibe el checkpoint se encarga de limpiar los logs almacenados.
- ***Log de mensajes*:** dado que el observador se encuentra en PML el mismo se asegura de que todas las comunicaciones pasen por él y puede hacer log de los mismos y luego transmitirlos utilizando el *framework* ***OOB*.**
- ***Mecanismo de detección de fallos*:** los fallos son detectados cuando los intentos de comunicación fallan, lo que supone modificar las capas más bajas para enviar errores a PML. Además de esto también se define un mecanismo de *heartbeat/watchdog* donde los protectores fijan un tiempo para enviar *heartbeats* al siguiente nodo lógico dentro de la cadena de protección, el protector receptor también cuenta con un temporizador (*watchdog*) que se reinicia con cada *heartbeat* recibido.
- ***Gestión de fallos:*** en Open MPI el mecanismo de actuación por defecto es *fail-stop* por lo cual RADIC debe añadir al demonio **ORTE** funciones de recuperación cuando es detectado un fallo. Si el protector confirma el fallo por alguno de los mecanismos con los que cuenta, inicia el proceso de recuperación.
- ***Recuperación:*** la recuperación está compuesta por tres fases. En la primera fase el protector reinicia el proceso fallado desde el *checkpoint*. En la segunda fase se reejecuta a partir del *checkpoint*, para procesar mensajes de entrada el log es releído para llevar al proceso al punto en el cual fallo. El observador restaurado debe escoger su nuevo protector. Además todos los protectores involucrados deben restablecer la cadena de protección.
- ***Reconfiguración:*** En la última fase deben reconfigurarse las comunicaciones y para lograr estos objetivos deben modificarse las capas más bajas de Open MPI para que se redirijan las comunicaciones a la nueva

dirección del protector. Esta información puede actualizarse bajo demanda, es decir, cuando quieren comunicarse con el proceso restaurado consultan las tablas de RADIC para ubicar al proceso o por la difusión de tablas por medio de *tokens* (Jalote, 1994).

5.3.3. Integración de *Spares* en la implementación de RADIC

De acuerdo a lo que ya se ha expresado anteriormente, la introducción de nodos *spares* en RADIC añade la necesidad de un algoritmo para que sea determinista el procedimiento de ubicar a los procesos recuperado, esto es solucionado aumentando la comunicación entre los observadores y los protectores locales para que de esta manera los observadores puedan consultar la *sparetable* y recorrerla hasta encontrar al proceso restaurado. Una vez se localiza se actualiza la radictable, y no es necesario volver a buscarlo, a no ser que ocurra un nuevo fallo.

La implementación actual permite la inclusión de nodos *spare* desde el arranque de la aplicación y establecidos como nodos desactivados, entonces, se puede tener un número fijo, limitado pero configurable de nodos *spare*.

Cuando arranca la aplicación cada nodo establece a su protector y carga la *sparetable* (Tabla 5-1), esto se hace cuando cada demonio **ORTE** carga el componente **PML** que le corresponde, y esta tabla se actualiza cuando algún observador es cargado dentro del nodo *spare*.

Cuando ocurre un fallo y el protector del proceso fallado se entera, en lugar de recuperarlo en su mismo nodo, busca en la *sparetable* y consulta al primer *spare* que encuentre disponible para verificar su condición, y el *spare* aceptará adoptar al proceso fallado si está disponible y en caso de que varios protectores traten de utilizar al mismo *spare*, éste solo atenderá la primera solicitud.

Si se agotan los *spares* disponibles, el mecanismo utilizado para restaurar al proceso fallado es el mecanismo explicado en la Figura 3.6.

5.3.4. Dificultades enfrentadas

La implementación de RADIC en Open MPI trae consigo complicaciones, dado que RADIC es una arquitectura totalmente distribuida y no utiliza operaciones colectivas, sin embargo, Open MPI utiliza operaciones colectivas para diseminar información de contacto, para realizar *checkpoints*, etc.

A continuación se citan otras dificultades enfrentadas:

- La inserción de nodos no pertenecientes a la aplicación como nodos desactivados. Dado que Open MPI no está preparado para integrar nuevos nodos lo que se puede es incluir nodos desactivados mediante el *hostfile*, donde son incluidos utilizando el marcador "^" de acuerdo a lo descripto en (Indiana University and Partners, 2011).
- La integración de un sistema de tolerancia a fallos descentralizado en una arquitectura centralizada. Open MPI no está preparado para utilizar estrategias de *checkpoint* no coordinado, por esto es necesario modificar varios módulos internos para utilizar la librería de *checkpoint* para guardar el estado de cada proceso individualmente y no como un todo.
- El anuncio de los nodos *Spare* es una complicación dado que no se pueden utilizar operaciones colectivas. Se utiliza el mecanismo de *token* que irá recorriendo todos los nodos del clúster.
- La transferencia de los *checkpoints* y reinicio de la aplicación a partir de él, dado que no basta con reiniciar el proceso utilizando la API de BLCR (Duell, 2003), ya que al reiniciar el proceso debe ser incluido como hijo del demonio **ORTE** residente en el nodo en el cual se restaura, lo cual implica modificar las estructuras internas de Open MPI.
- El enmascaramiento de fallos supone retrasar las comunicaciones de los procesos que se quieren comunicar con el proceso fallado hasta que la restauración culmine, es decir, es necesario engañar a la librería para que no finalice la aplicación. Es decir, es necesario impedir el comportamiento *fail-stop* que se utiliza por defecto.
- El restablecimiento de las comunicaciones es uno de los mayores problemas, dado que se debe diseminar bajo demanda la nueva ubicación

del proceso restaurado y además se deben reconfigurar los canales de comunicación de Open MPI, lo cual supone cambios en los *endpoints* de los procesos de aplicación y de los demonios.

5.5. Considerando MPI3

Actualmente se encuentra en definición el estándar MPI 3 (2011), en el cual se busca básicamente mejorar la gestión de errores por parte del usuario, o facilitar la misma dando más y mejores informes de fallos a los usuarios para que los mismos sean capaces de gestionarlos.

Para lograr estos objetivos cada implementación del estándar MPI buscará mejorar la gestión de fallos, pero no se propone claramente cómo tratar esos fallos y como recuperar procesos que han fallado ni la política a seguir, quedando eso a cargo de las implementaciones.

Claramente puede verse no habrá problemas en la integración de RADIC en este estándar, dado que RADIC proveerá a las implementaciones lo necesario para que a partir de la captura del error se puedan tomar medidas para restaurar a el/los procesos que han sido víctimas de un nodo que ha fallado.

De acuerdo a lo expuesto puede verse que en lugar de contraponerse, ambas ideas se complementarán, y la implementación de RADIC se verá favorecida por la mayor cantidad de información que circulará a capas superiores lo cual facilitará la captura de errores y su gestión.

Además de esto, existe un plan para desarrollar una versión de RADIC que se independiente de la librería de comunicaciones lo cual permitirá tener un *middleware* de tolerancia a fallos independiente y que mantiene todas las características propias de RADIC.

Capítulo 6.
Evaluación experimental

6.1. Ambiente experimental

En la Tabla 6-1 se describe brevemente la máquina paralela en la cual se han llevado a cabo los experimentos que son descriptos más abajo.

Tabla 6-1. Ambiente de pruebas - Dell *PowerEdge* M600.

COMPONENTES		DESCRIPCIÓN
Hardware	Arquitectura	
	Procesador	Intel(R) Xeon(R) CPU E5430 @ 2.66 GHz quad-core 6MB L2
	Número de procesadores	2
	Memoria	16 GB Fully Buffered DIMMs (FBD) 667 MHz
	Chipset	Intel 5000P
	Controladores RAID	SAS6/ir (H/W based) con soporte RAID 0/1
	Almacenamiento interno	2 discos SATA (7.2k rpm) de 120 GB.
	Comunicaciones	Tarjeta doble integrada Broadcom(R) NetXtreme IITM 5708 Gigabit Ethernet con *firmware TOE* e iSCSI
Software	Sistema Operativo	Red Hat Linux 5 Enterprise Linux Kernel 2.6.18-8.el5 #1 SMP x86_64 GNU/Linux
	Gestión del sistema	• Dell OpenManage software tools. • OpenManage Server Administrator - 1:1 monitoring agents. • Integrated Dell Remote Access Controller (iDRAC). • Out of Band alerting, status, inventory and troubleshooting via Secure Web GUI/CLI. • Remote Virtual Media (vMedia) and Virtual KVM (vKVM). • IPMI 2.0 support. • Ganglia v3.1.1.
	Compiladores	• Intel Fortran77/90/95 C V10.1.12 • GNU Fortran77 C C++ V4.1.2 • Open MPI Versión 1.7

6.2. Aplicaciones utilizadas

En esta sección se explican las aplicaciones que se han utilizado para testear la implementación de RADIC dentro de Open MPI. Las aplicaciones que se han utilizado pertenecen a los *NAS Parallel Benchmark* (NPB) (Wechsler, 1995) y además se utilizó una aplicación de multiplicación de matrices estática, las aplicaciones se explican brevemente a continuación.

6.2.1. Multiplicación de matrices estática

Como test inicial se ha utilizado una multiplicación de matrices paralela estática, siguiendo un modelo de programación *Master/Worker*. En esta aplicación la matriz A es distribuida por filas y la matriz B es distribuida entera a cada proceso. Cada uno de los procesos que compone la aplicación calcula su parte y devuelve el resultado al proceso *master*.

Se ha utilizado una distribución de la carga estática, para que se añada a la degradación de la pérdida computacional, la degradación debida al desbalanceo de carga. De este modo, cuando falla un nodo, en el modo básico de RADIC otro nodo debe computar su trabajo y el trabajo del proceso que ha recuperado. Por lo tanto la degradación debe ser apreciable.

6.2.2. NAS Parallel Benchmark

Los *benchmarks* paralelos NAS son ampliamente utilizados para evaluar el rendimiento de maquinas paralelas, y proveen reportes como salida, en los cuales se presentan los tiempos de ejecución, la cantidad de procesadores utilizados, el *speedup*, la eficiencia y las prestaciones del sistema (Wong, y otros, 1999).

Los resultados proveen comprensión acerca de las prestaciones promedio del sistema paralelo. Dado que los algoritmos utilizados son conocidos y la existencia de un estándar de paso de mensajes (MPI) (Forum MPI, 2009) hacen sencilla la utilización de

estos *benchmarks* para realizar análisis comparativos de las arquitecturas paralelas, por lo tanto también son útiles para realizar la comparación de la implementación de RADIC en Open MPI contra la implementación pura de Open MPI.

En este trabajo se ha utilizado una de las aplicaciones que está presente dentro del set de *benchmarks* de NPB. Es preciso destacar además que existen clases para cada una de las aplicaciones que componen el set de NPB, cada clase tiene un *workload* diferente e incremental. En los resultados que son mostrados en la siguiente sección han sido utilizadas las clases B y C del *benchmark* LU, donde el workload de la clase C es mayor que la de B (Bailey, y otros, 1994).

El *benchmark* LU utiliza una matriz *sparse* y resuelve la multiplicación de una matriz triangular superior por una matriz triangular inferior. Debido a la inestabilidad de este método, por ejemplo si un elemento de la diagonal es cero, es necesario pre-multiplicar la matriz por una matriz de permutación.

6.3. Resultados experimentales

En esta sección se presentan los resultados preliminares obtenidos con la implementación de RADIC en Open MPI utilizando las aplicaciones descriptas en la sección anterior.

En la experimentación con multiplicación de matrices se ha establecido un intervalo de *checkpoint* de 30 segundos, y en los experimentos realizados con los *benchmarks* NAS se ha fijado nada más que un *checkpoint* aproximadamente en la mitad de la aplicación.

6.3.1. Sobrecarga introducida por la tolerancia a fallos

Aquí se presentan los resultados experimentales de la implementación de RADIC en Open MPI teniendo en cuenta la sobrecarga que introduce la utilización de los *checkpoints* no coordinados (realización y transferencia) y la realización y escritura a disco del log pesimista basado en el receptor.

En la Tabla 6-2 son presentados los tiempos obtenidos al realizar experimentos con la aplicación de multiplicación de matrices que fue explicada anteriormente. Los tiempos han sido obtenidos con 8, 16 y 32 procesos utilizando 4 nodos y con 64 procesos utilizando 8 nodos con la librería Open MPI sin tolerancia a fallos y utilizando RADIC.

Tabla 6-2. Tiempo (seg.) con y sin tolerancia a fallos de la multiplicación de matrices estática.

		Tamaño de matrices		
Procesos	**Tipo**	**1000**	**2000**	**3000**
8	**Sin TF**	198,3	1665,4	6004
	Con TF	201	1675,5	6047,2
16	**Sin TF**	102,2	789,7	2643,4
	Con TF	104,7	803,3	2759,3
32	**Sin TF**	67,52	458,6	1305,5
	Con TF	72,1	544,3	1493,2
64	**Sin TF**	107,8	485,8	1231,5
	Con TF	147,9	639,6	1557,9

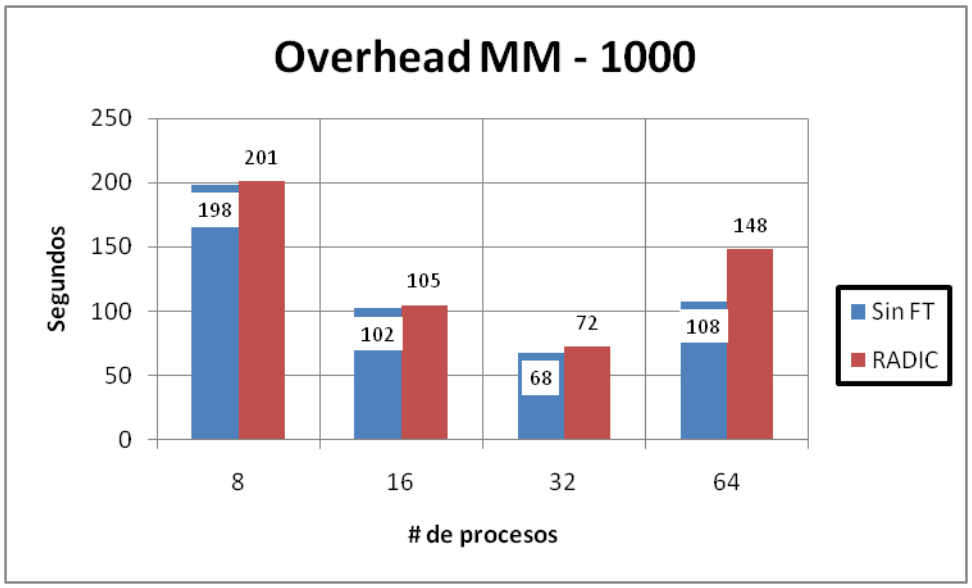

Figura 6.1. *Overhead* introducido en la multiplicación de matrices de tamaño 1000.

En la Figura 6.1, Figura 6.2 y Figura 6.3 se pueden observar los *overheads* que introduce la implementación actual de RADIC en Open MPI comparado contra la ejecución normal en Open MPI sin agregados de tolerancia a fallos. Vemos que para 8, 16, y 32 procesos RADIC introduce un pequeño *overhead*. En el peor caso la sobrecarga

introducida por RADIC es del 37% si se considera la Figura 6.1. La hipótesis de RADIC es que el sistema de tolerancia a fallos escale con la aplicación, desde el momento en el cual la aplicación deja de escalar, RADIC también lo hace, y en los tiempos empeoran.

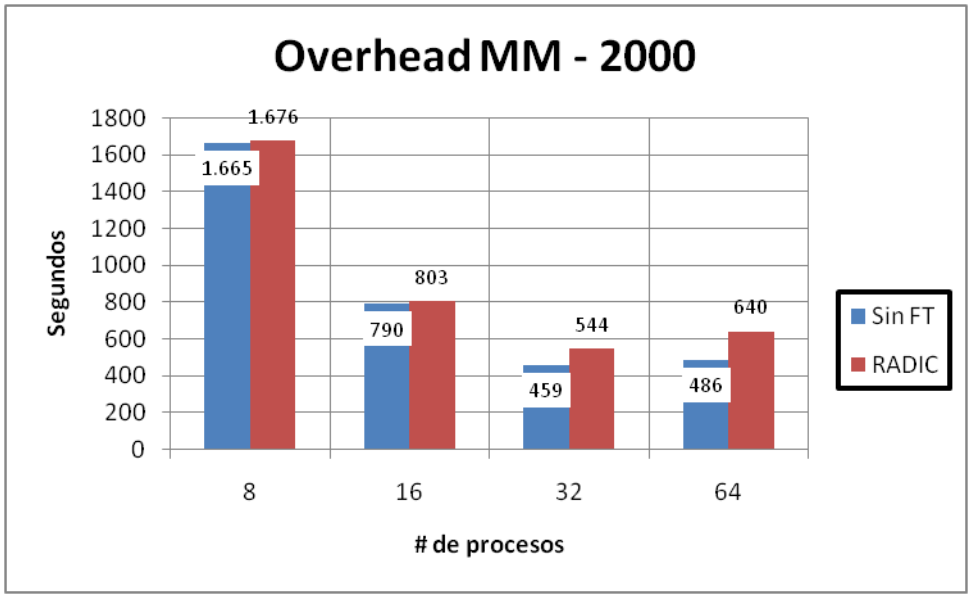

Figura 6.2. *Overhead* introducido en la multiplicación de matrices de tamaño 2000.

En el peor caso de la Figura 6.2 el *overhead* introducido por RADIC es del 31%, y es cuando se utilizan 64 procesos, al igual que en el caso anterior, esto es debido a que la aplicación ya no escala al utilizar 64 procesos.

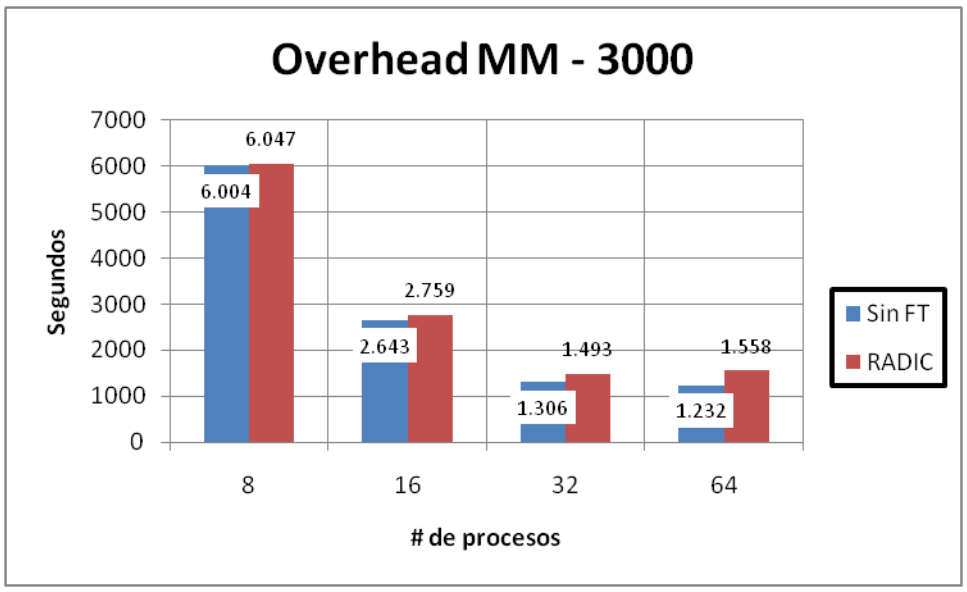

Figura 6.3. *Overhead* introducido en la multiplicación de matrices de tamaño 3000.

En la Figura 6.3, donde el tamaño de las matrices es de 3000 x 3000, el *overhead* introducido en el peor de los casos es del 26%, lo cual es menor que en los casos anteriores, este comportamiento se acerca al esperado, que el *overhead* está relacionado con la relación cómputo-comunicación de la aplicación, si la ejecución de la aplicación está limitada por comunicación RADIC afecta significativamente al tiempo de ejecución.

En el caso de la multiplicación de matrices la mayor parte del *overhead* es causado por la realización y transferencia de los checkpoints, dado que esta aplicación no realiza demasiada comunicación (que impacta en el *overhead* introducido por el log) ya que el *master* envía las porciones de las matrices a los *workers* y cuando acaba la computación, recoge estos resultados.

También se ha analizado el *benchmark* LU que pertenece a los NPB, en la Tabla 6-3 se muestran los tiempos obtenidos sin tolerancia a fallos, se muestran los *overheads* introducidos cuando sólo se realizan Logs, cuando se realizan y envían checkpoints y el tiempo total de ejecución al utilizar RADIC.

Tabla 6-3. Tiempos obtenidos con el benchmark LU clases B y C.

		Tiempos (segundos)			
Procesos	Clase	Sin TF	Con Log	Con Checkpoint	Con RADIC
4	B	86,45	93,9	93,27	102,63
	C	400,74	433,61	419,68	446,69
8	B	67,68	74,36	78,09	99,01
	C	224,73	259,77	235,81	276,33
16	B	49,62	54,42	77,88	109,6
	C	154,13	246,34	187,60	275,9
32	C	132,7	358,5	245,14	416,17

En la Figura 6.4 y en la Figura 6.5 pueden verse los tiempos obtenidos utilizando el *benchmark* LU que pertenece a los NPB. En ambos gráficos se discriminan los o*verheads* introducidos por la realización de Logs, realización y envío de *checkpoints* y además el total de utilizar RADIC completo.

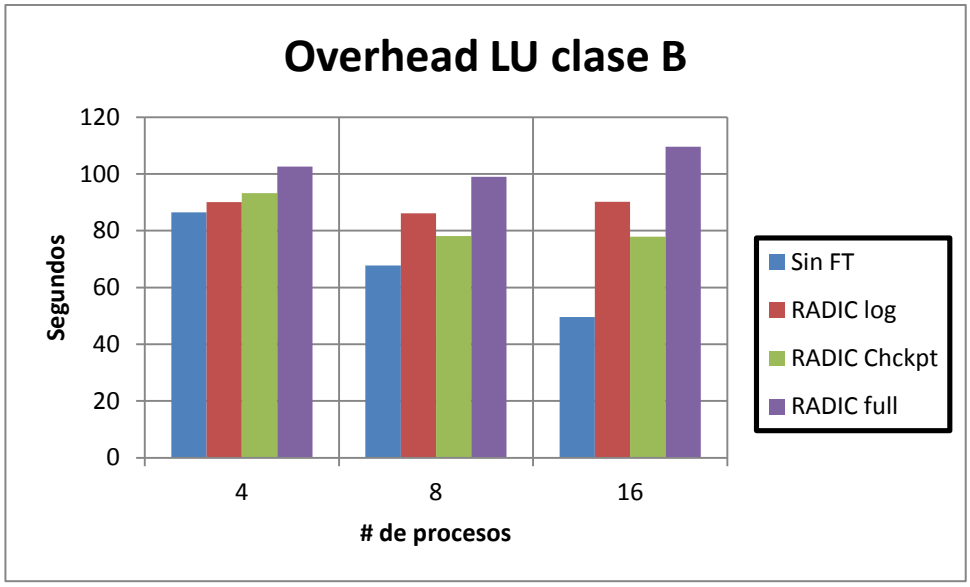

Figura 6.4. *Overhead* introducido en el *benchmark* LU clase B.

En la Figura 6.4 puede verse que el impacto de la tolerancia a fallos es mayor a causa de los *checkpoints* cuando los procesos son pocos, y cada uno de estos *checkpoints* tiene aproximadamente un tamaño de 150 MB. El crecimiento en el *overhead* cuando se utilizan 8 procesos o más, se debe a la realización de logs. Cuando se utilizan 16 procesos la comunicación predomina y se hace mayor cantidad de accesos a disco para guardar los logs, y por lo tanto los tiempos de RADIC aumentan, ya que desintoniza bastante la aplicación que de por sí ya no escala correctamente.

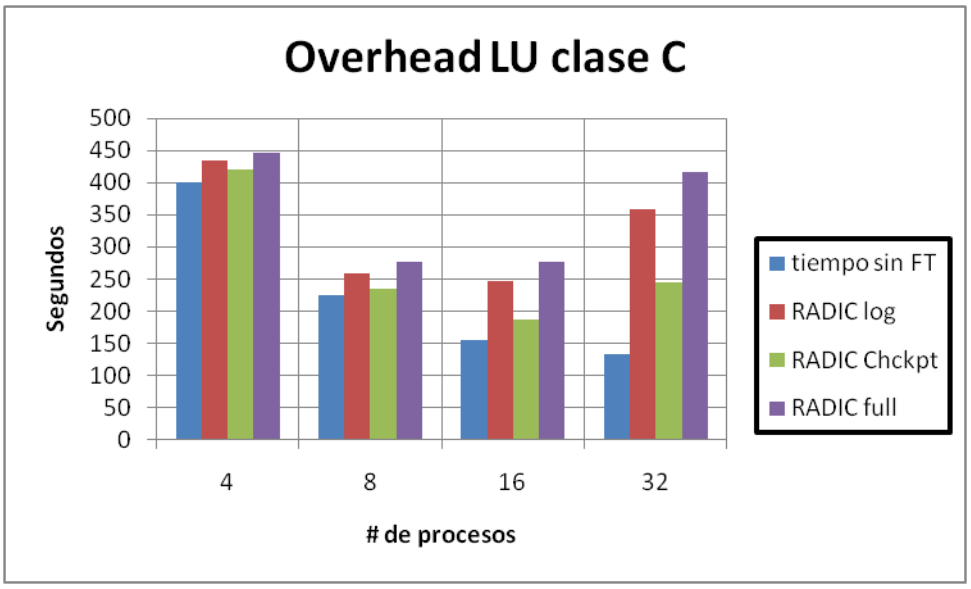

Figura 6.5. *Overhead* introducido en el benchmark LU clase C.

En la Figura 6.5 puede verse que la mayor sobrecarga es causada por la realización del log pesimista basado en el receptor, en la figura puede verse que a medida que se aumenta la cantidad de procesos que se utilizan con la misma carga de trabajo, la aplicación cambia su relación cómputo- comunicación y afecta más el *overhead* introducido por RADIC, dado que utilizando tolerancia a fallos, el punto de inflexión en el cual la aplicación deja de escalar se adelanta, ya que se introduce *overhead* y la aplicación va más lenta.

Considerando los *overheads* mostrados en esta sección puede concluirse que con una aplicación que realiza más cómputo que comunicación la mayor sobrecarga es debida a la realización y transferencia de *checkpoints*. Sin embargo, en una aplicación como el *benchmark* LU donde la comunicación es constante, y además los *checkpoints* son bastante grandes, el log cobra mayor importancia a medida que crece el número de procesos utilizado dado que cada mensaje tiene un retraso mayor porque solo se da por recibido un mensaje una vez que es almacenado en disco, por lo cual, todas las comunicaciones funcionan más lentamente, pudiendo llegar a ser más del doble el tiempo de cada comunicación.

6.3.2. Impacto en las prestaciones: *overhead* y degradación

En esta sección los resultados obtenidos muestran como impacta RADIC en las prestaciones de las aplicaciones paralelas, tanto en presencia, como en ausencia de fallos.

Se han realizado experimentos que muestran el comportamiento de las prestaciones que ofrecen los *benchmarks* LU cuando se introduce RADIC y como se degradan las mismas cuando ocurre un fallo.

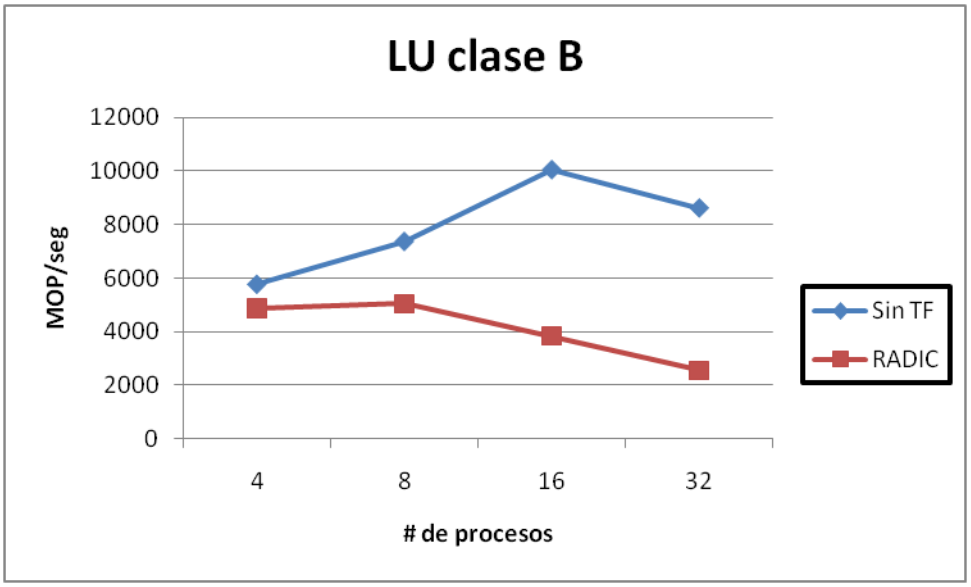

Figura 6.6. Prestaciones del *benchmark* LU clase B en ausencia de fallos.

En la Figura 6.6 y en la Figura 6.7 se observa la degradación en prestaciones medida en operaciones/segundo (como un indicador de *throughput*) del *benchmark* LU clase B y C respectivamente. Puede verse que el impacto en las prestaciones es mayor a más procesos, pero esto no quiere decir que la arquitectura de tolerancia a fallos no escale correctamente, dado que si se observa el comportamiento de la aplicación sin tolerancia a fallos puede verse que para una misma carga de trabajo, tiene un límite para el máximo número de procesadores que pueden utilizarse para lograr un aumento en las prestaciones.

También debe considerarse que debido a la configuración (*mapping*) utilizada de más de un proceso por nodo, cuando se desea escribir a disco existen competencias dentro del mismo nodo, y lo mismo ocurre cuando se desea utilizar la red para transferir los *checkpoints* a los protectores, por lo cual la configuración también colabora con los *overheads*.

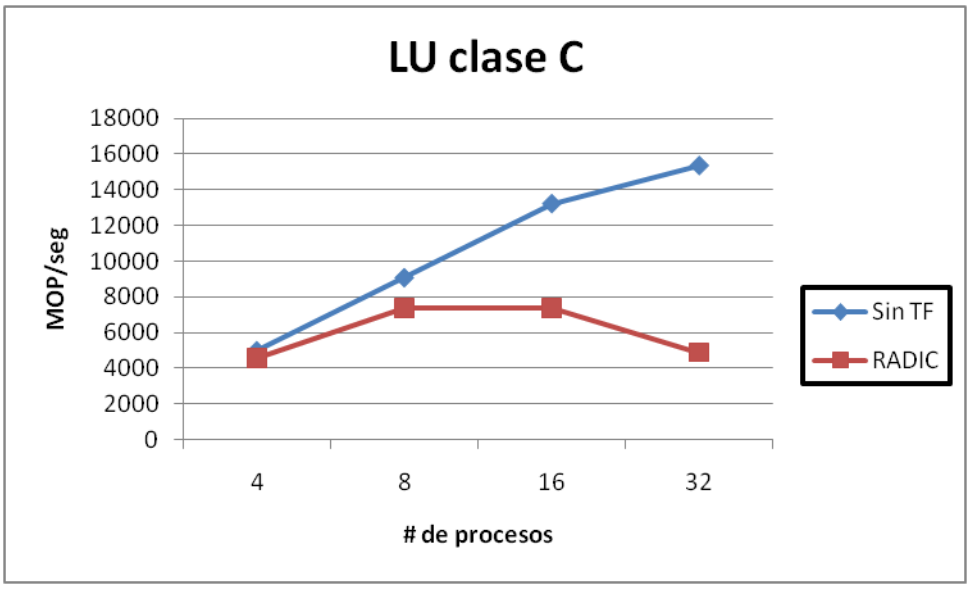

Figura 6.7. Prestaciones del *benchmark* LU clase C en ausencia de fallos.

Uno de los principales objetivos de este trabajo es la introducción de nodos *spare* dentro de la arquitectura RADIC y demostrar que las prestaciones del sistema se degradan considerablemente dado que hay una pérdida de capacidad computacional y el protector en el cual se restaura el proceso fallado queda sobrecargado.

En la Figura 6.8 se observan dos ejecuciones del algoritmo de multiplicación de matrices. En una de las ejecuciones se utilizan nodos *spare* para restaurar al proceso que falla, y en la otra ejecución el proceso fallado es restaurado en su protector dejándolo sobrecargado. En la ejecución con *spares* se cuenta con 2 *spares*, y se inyectan 2 fallos para utilizar ambos *spares*. En la ejecución sin nodos *spare* se inyecta un solo fallo y se restaura al proceso en su protector.

Debe aclararse que para obtener las operaciones por segundo (OP/seg) en la Figura 6.8 se ha calculado el tiempo que se tarda la aplicación en cada iteración interna y se ha dividido el tamaño de sub-matriz que computa cada proceso por el tiempo calculado. En esta ejecución se utilizaron 32 procesos en 4 nodos (un proceso por *core*) y un intervalo de *checkpoint* de 30 segundos además de un tamaño de matriz de 3000.

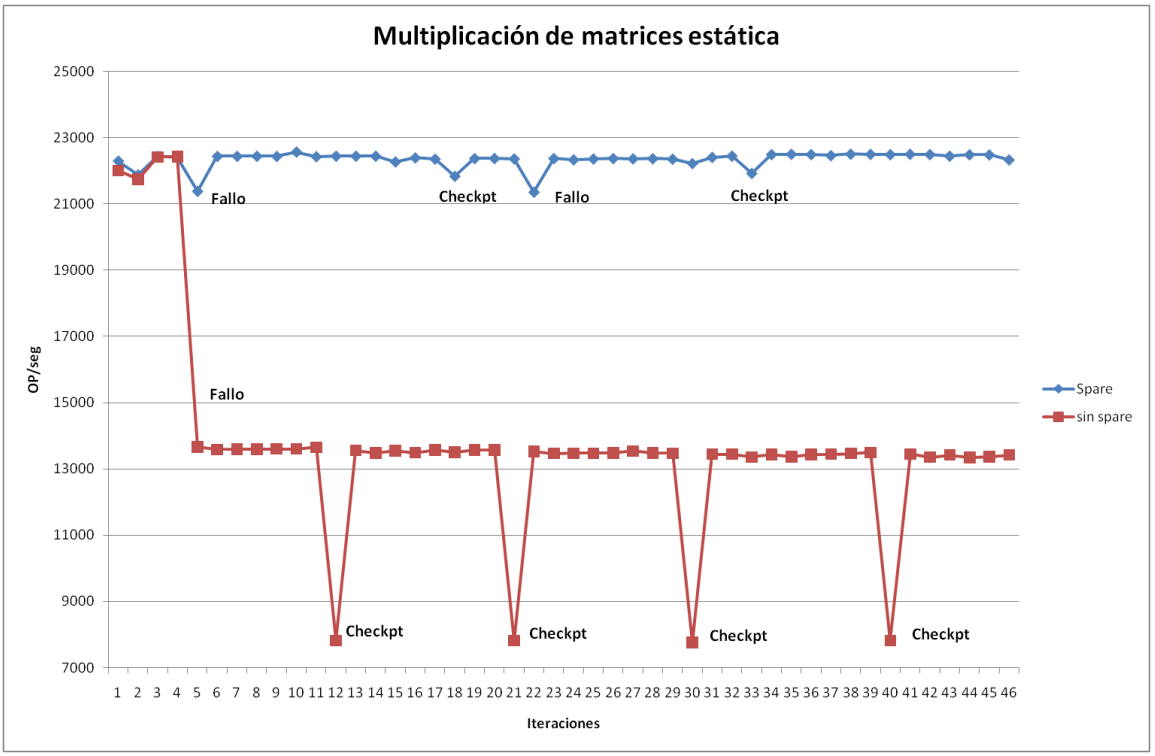

Figura 6.8. Prestaciones de la multiplicación de matrices en presencia de fallos con tamaño 3000.

En la Figura 6.9 se observan las prestaciones del *benchmark* LU cuando ocurren fallos y se cuenta con nodos *spare*. Esta ejecución se ha hecho utilizando 8 procesos en 4 nodos (2 procesos por nodo) y se ha utilizado la C del *benchmark* LU, además se realiza un solo *checkpoint* aproximadamente a los 30 segundos de la ejecución. Se muestran dos ejecuciones, una sin *spares* en la cual el proceso fallado se restaura en su protector, y otra en la cual ocurre un fallo y se restaura en un nodo *spare* sin dejar sobrecargado ningún nodo. Para obtener las operaciones por segundo en esta aplicación se ha medido el tiempo de cada una de las iteraciones internas del *benchmark* y el tamaño de matriz que le toca a cada proceso es dividido por este tiempo.

La aplicación tiene un comportamiento medianamente irregular ya que es dependiente de los datos y trabaja con matrices dispersas. De todas formas la aplicación ejecutándose sin nodos *spare* presenta menores prestaciones cuando ocurre un fallo en un proceso, ya que competirá con otros procesos por los recursos al restaurarse en su protector, y sin embargo la ejecución con *spares* pierde solamente un poco las prestaciones mientras se restaura, pero esto queda disfrazado con el comportamiento irregular de la aplicación.

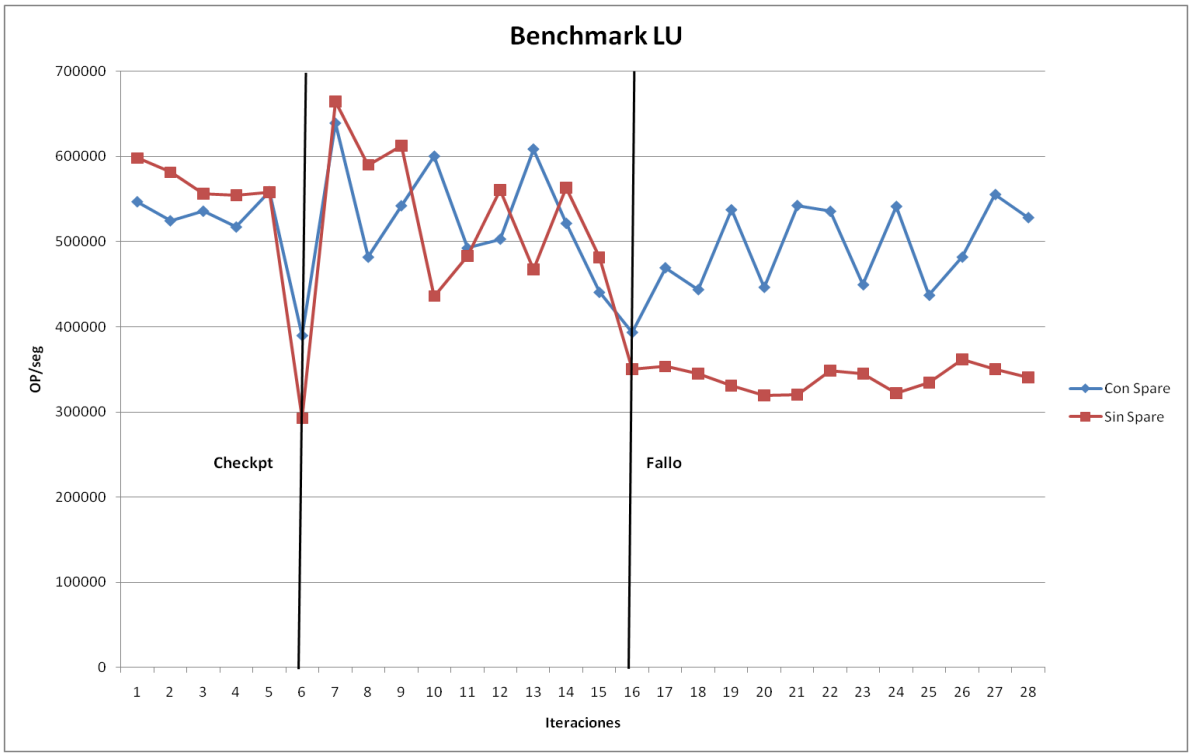

Figura 6.9. Prestaciones del *benchmark* LU clase C con 8 procesos en presencia de fallos.

Considerando la Figura 6.8 y la Figura 6.9, puede concluirse que el comportamiento de una aplicación con nodos *spare* luego de un fallo es bastante similar al comportamiento de la aplicación antes de la ocurrencia de fallos.

Con los resultados obtenidos puede decirse que los objetivos que fueron inicialmente trazados han sido alcanzados. Aunque se debe comentar que en estos experimentos no se considera el costo del traslado del log al protector y de la utilización del protocolo de *heartbeat/watchdog*.

Capítulo 7.
Conclusiones y Trabajos Futuros

7.1. Conclusiones

La propuesta presentada en este trabajo ha probado como la inclusión de nodos *spare* permite mantener las prestaciones en las aplicaciones paralelas cuando ocurre un fallo con un *overhead* mínimo para el tratamiento del fallo. Esto es bastante importante dado que generalmente las aplicaciones se configuran para funcionar con un número de nodos, y si se van perdiendo componentes durante la ejecución, la aplicación se desintoniza y la ejecución se vuelve mucho más lenta. Inclusive si consideramos una aplicación que utiliza 1000 nodos, y se pierde uno de estos nodos, las prestaciones de la aplicación disminuyen drásticamente aproximándose al 50%, lo cual puede ser fatal para una aplicación científica.

Al tener como objetivo la escalabilidad se utiliza una estrategia de tolerancia a fallos descentralizada pero obviamente la tolerancia a fallos introduce *overheads*, pero debe tenerse en cuenta el impacto que produce un fallo en una aplicación de gran escala y considerar la tolerancia a fallos como una buena opción de equilibrio prestaciones-costos. Una de las ventajas es poder acotar el impacto del fallo, además de garantizar la disponibilidad del sistema en caso de fallos. En un sistema sin tolerancia a fallos es difícil prever (predecir o analizar) el impacto en caso de fallos.

La implementación de RADIC en Open MPI trae grandes ventajas, dado que es una librería de amplia utilización en el mundo científico y permitirá la utilización de RADIC en aplicaciones del mundo científico, además la implementación facilitará la utilización de la migración en caliente dentro de Open MPI y se complementará correctamente con el estándar MPI 3.

El estándar MPI 3(que aún se encuentra en definición) ha sido analizado, y se ha visto que el mismo facilitará la gestión de errores por parte de los usuarios, añadiendo más información acerca de los fallos ocurridos y dará posibilidades de tomar acciones frente a

los mismos. RADIC se complementará correctamente con esta propuesta, dado que se beneficiará de la información extra obtenida y se encargará de la gestión automática de fallos, lo cual no será incluido dentro del estándar.

Una arquitectura de tolerancia a fallos con las características de RADIC es deseable ya que no requiere intervención alguna del usuario y es automática, además de ser configurable de acuerdo a los componentes disponibles. El usuario puede analizar la relación coste prestaciones y decidir cuándo es útil o no usar *spares.*

7.2. Líneas abiertas

La integración dentro de Open MPI así como ofrecer una interfaz a Open MPI para la migración en caliente e incluir el trabajo realizado dentro del código abierto de Open MPI es también una tarea pendiente.

Seguir los pasos al estándar MPI 3 y adaptar la actual implementación para aprovechar las ventajas ofrecidas por éste estándar.

Como un trabajo futuro queda la adaptación de RADIC a aplicaciones paralelas que cuenten con eventos de entrada/salida, aplicaciones del mundo transaccional, etc.

Además es necesario desarrollar políticas que consideren los nodos *multi-core* al momento de trasladar procesos. Estas políticas aún no han sido definidas por lo cual también requerirán ser implementadas dentro de la versión actual.

Bibliografía

[En línea] // MPI Forum: mpi3. - 2011. - http://lists.mpi-forum.org/mpi3-p2p/.

Agbaria Adnan y Friedman Roy Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations [Conferencia]. - [s.l.] : IEEE Computer Society, 1999. - págs. 31--39.

Ansel Jason, Arya Kapil y Cooperman Gene DMTCP: Transparent checkpointing for cluster computations and the desktop [Conferencia]. - 2009. - págs. 1--12.

ARM, Auburn University, Bull, Chelsio Communications, Cisco Systems Inc., Computer Architecture Group, Computer Science Department, University of British Columbia, EverGrid, IBM, Oracle [En línea] // Open MPI: Open Source High Performance Computing. - 2011. - http://www.open-mpi.org.

Avizienis Algirdas [y otros] Basic Concepts and Taxonomy of Dependable and Secure Computing [Publicación periódica] // IEEE Trans. Dependable Secur. Comput.. - [s.l.] : IEEE Computer Society Press, January de 2004. - 1 : Vol. 1. - págs. 11--33. - 1545-5971.

Bailey D [y otros] The Nas Parallel Benchmarks [Publicación periódica] // International Journal of High Performance Computing Applications. - 1994.

Bhargava Bharat K y Lian Shy-renn Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - An Optimistic Approach [Conferencia]. - 1988. - págs. 3--12.

Bosilca G [y otros] MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes [Conferencia] // Supercomputing, ACM/IEEE 2002 Conference. - 2002. - págs. 29 - 29.

Bouteiller Aurélien y Hérault Thomas MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging [Conferencia] // Supercomputing, 2003 ACM/IEEE Conference. - 2003. - págs. 25 - 25.

Buntinas Darius [y otros] Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols [Publicación periódica] // Future Generation Comp. Syst.. - 2008. - Vol. 24. - págs. 73-84.

Cappello Franck Fault Tolerance for PetaScale Systems: Current Knowledge, Challenges and Opportunities [Publicación periódica] // Recent Advances in Parallel Virtual Machine and Message Passing Interface. - [s.l.] : Lastovetsky, Alexey and Kechadi, Tahar and Dongarra, Jack, 2008. - Vol. 5205.

Chandy K M y Lamport Leslie Distributed snapshots: determining global states of distributed systems [Publicación periódica] // ACM Trans. Comput. Syst.. - [s.l.] : ACM, February de 1985. - 1 : Vol. 3. - págs. 63--75. - 0734-2071.

Duarte Angelo Amancio RADIC: a powerful fault-tolerant architecture [Libro]. - Bellaterra : Universitat Autònoma de Barcelona, 2007. - págs. 21 - 95.

Duarte Angelo, Rexachs Dolores y Luque Emilio An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI [Publicación periódica]. - 2006. - págs. 150--157.

Duell Jason The design and implementation of Berkeley Labs linux Checkpoint/Restart [Publicación periódica]. - 2003.

Elnozahy E. [y otros] A survey of rollback-recovery protocols in message-passing systems [Publicación periódica] // ACM Comput. Surv.. - [s.l.] : ACM, September de 2002. - 3 : Vol. 34. - págs. 375--408. - 0360-0300.

Fagg Graham E. y Dongarra Jack J. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world [Publicación periódica] // Recent Advances in Parallel Virtual Machine and Message Passing Interface. - 2000. - Vol. 1908. - págs. 346-353.

Fialho Leonardo Applying RADIC in Open MPI [Libro] / ed. Barcelona Universitat Autònoma de. - Bellaterra : [s.n.], 2008. - págs. 31 - 162.

Forum MPI MPI: A Message-Passing Interface Standard Version 2.2 [Libro]. - 2009.

Graham Richard L. [y otros] A Network-Failure-tolerant Message-Passing system for Terascale Clusters [Publicación periódica] // Int. J. Parallel Program.. - August de 2003. - Vol. 31. - págs. 285-303.

Hursey Joshua [y otros] The design and implementation of checkpoint/restart process fault tolerance for Open MPI [Publicación periódica] // In Workshop on Dependable Parallel, Distributed and Network-Centric Systems(DPDNS), in conjunction with IPDPS. - 2007. - págs. 1-8.

Indiana University and Partners [En línea] // Open MPI: Open Source High Performance Computing.. - 2011. - http://www.open-mpi.org/.

Intel Corporation; IBM; Cray Inc.; Microsoft; Ohio State University; Myricom; University of British Columbia [En línea] // MPICH-A Portable Implementation of MPI. - 2011. - http://www.mcs.anl.gov/research/projects/mpich2/.

Jalote P. Reliable, Atomic and Causal Broadcast [Sección del libro] // Fault Tolerance in Distributed Systems. - [s.l.] : P T R Prentice Hall, 1994.

Johnson David B. Distributed system fault tolerance using message logging and checkpointing [Libro]. - [s.l.] : Rice University, 1989.

Kalaiselvi S. y Rajaraman V. A survey of checkpointing algorithms for parallel and distributed computers. [Publicación periódica] // Sadhana. - 2000. - 5 : Vol. 25. - págs. 489-510..

Koren Israel y Krishna C. M. Fault Tolerant Systems [Libro]. - [s.l.] : Morgan Kaufmann Publishers Inc., 2007. págs. 310-353.

Lamport Leslie Ti clocks, and the ordering of events in a distributed system [Publicación periódica] // Commun. ACM. - [s.l.] : ACM, July de 1978. - 7 : Vol. 21. - págs. 558--565. - 0001-0782.

Litzkow Michael [y otros] Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System [Informe] : techreport. - 1997.

Plank J S [y otros] Libckpt: Transparent Checkpointing under Unix [Publicación periódica] // Usenix Winter Technical Conference. - New Orleans, LA : [s.n.], January de 1995. - págs. 213--223.

Plank J S [y otros] Libckpt: Transparent Checkpointing under Unix [Conferencia]. - New Orleans, LA : [s.n.], 1995. - págs. 213--223.

Randell B. System structure for software fault tolerance [Publicación periódica] // SIGPLAN Not.. - [s.l.] : ACM, April de 1975. - 6 : Vol. 10. - págs. 437--449. - 0362-1340.

Rao Sriram, Alvisi Lorenzo y Vin Harrick M Egida: An Extensible Toolkit for Low-Overhead Fault-Tolerance [Publicación periódica] // In Symposium on Fault-Tolerant Computing. - 1999. - págs. 48--55.

Rodrigues de Souza Josemar FTDR: Tolerancia a fallos, en clusters de computadores geográficamente distribuidos, basada en Replicación de Datos [Libro] / ed. Barcelona Universitat Autònoma de. - Bellaterra : [s.n.], 2006. - págs. 22 - 38.

Sankaran Sriram [y otros] The LAM/MPI checkpoint/restart framework: System-initiated checkpointing [Publicación periódica] // In Proceedings, LACSI Symposium, Sante Fe. - 2003. - págs. 479--493.

Santos Guna [y otros] Providing Non-stop Service for Message-Passing Based Parallel Applications with RADIC [Publicación periódica] // Proceedings of the 14th international Euro-Par conference on Parallel Processing. - [s.l.] : Springer-Verlag, 2008. - págs. 58--67.

Santos Guna y Duarte Angelo Increasing the Performability of Computer Clusters Using RADIC II [Publicación periódica] // International Conference on Availability, Reliability and Security.. - [s.l.] : IEEE Computer Society, 2008. - págs. 653--658.

Schroeder Bianca y Gibson Garth A. Understanding failures in petascale computers [Publicación periódica] // Journal of Physics: Conference Series. - 2007. - Vol. 78.

Stellner Georg. CoCheck: Checkpointing and Process Migration for MPI [Publicación periódica] // In Proceedings of the 10th International Parallel Processing Symposium (IPPS ’96). - 1996. - págs. 526--531.

Strom Rob y Yemini Shaula Optimistic recovery in distributed systems [Publicación periódica] // ACM Trans. Comput. Syst.. - [s.l.] : ACM, August de 1985. - 3 : Vol. 3. - págs. 204--226. - 0734-2071.

Tamir Yuval y Sequin Carlo H. Error Recovery in Multicomputers Using Global Checkpoints [Publicación periódica] // In 1984 International Conference on Parallel Processing.. - 1984. - págs. 32--41.

TOP 500 Project Top 500 - Supercomputer Sites [En línea]. - 2011. - http://www.top500.org/.

Wechsler Elisabeth NAS Parallel Benchmarks Set The Industry Standard for MPP Perfomance [Publicación periódica]. - Enero de 1995. - Vol. 2.

Weissman Jon B. Fault Tolerant Computing on the Grid: What are My Options? [Publicación periódica] // Proceedings in The Eighth International Symposium on High Performance Distributed Computing.. - 1998. - págs. 351-352.

Wong Frederick C. [y otros] Architectural requirements and scalability of the NAS parallel benchmarks [Publicación periódica] // Proceedings of the 1999 ACM/IEEE conference on Supercomputing. - [s.l.] : ACM, 1999.