Universitat Autònoma
de Barcelona

Escola d'Enginyeria

Departament d'Arquitectura de Computadors i Sistemes Operatius

Tolerancia a fallos en la capa de sistema basada en la arquitectura RADIC

Tesis doctoral presentada por **Marcela Castro León** para optar por el grado de Doctor por la Universitat Autònoma de Barcelona, bajo la dirección de la Dr. Dolores Rexachs

Barcelona, Julio 2013

Tolerancia a fallos en la capa de sistema basada en la arquitectura RADIC

Tesis doctoral presentada por Marcela Castro León para optar al grado de Doctor por la Universitat Autónoma de Barcelona. Trabajo realizado en el Departament d Arquitectura de Computadors i Sistemes Operatius de la Escola d Enginyeria de la Universitat de Barcelona, dentro del Programa de Computación de Altas Prestaciones bajo la dirección de la Dr. Dolores Rexachs.

Barcelona, Julio 2013

Supervisor

Dr. Dolores Rexachs

Resumen

Castellano

La demanda de mayor rendimiento de las aplicaciones científicas se satisface incrementando la cantidad de componentes. Sin embargo, un mayor número de componentes implica una mayor probabilidad de fallo. La abrupta caída de los tiempos medios entre fallos en los sistemas actuales de altas prestaciones impulsa la investigación de mecanismos de tolerancia a fallos para garantizar la ejecución de una aplicación a un coste razonable.

Message-Passing Interface (MPI), el estándar de programación más utilizado por las aplicaciones científicas, tiene un comportamiento *fail-stop*, realizando una parada segura de todos los procesos si se detecta un fallo en un nodo del clúster. Como consecuencia, se pierde la ejecución que se hubiera hecho en todos los nodos de procesamiento.

Los sistemas de cómputo de altas prestaciones han implementado mecanismos para garantizar el servicio, normalmente basados en técnicas de *rollback-recovery* mediante uso de *Checkpoint/Restart*. Estas soluciones se han implementado a nivel de aplicación lo cual no es transparente, o bien, a nivel de librería, lo cual no es generalizable a otras librerías y dejan fuera del campo de solución a un número diverso de aplicaciones.

Se propone un sistema de tolerancia a fallos transparente y automático de modo que pueda utilizarse sin modificar la aplicación y con la librería de paso de mensaje que prefiera el usuario. Se basa en detectar los errores en las comunicaciones de *socket* causados por fallos de nodos y reconfigurarlos en forma automática para comunicarse con la nueva dirección a donde se migra el proceso. Funciona en conjunto con un sistema que protege el estado de cómputo de los procesos y en caso de fallos, los recupera en otro nodo de cómputo por medio de técnicas de *rollback-recovery*.

Se ha realizado una validación experimental utilizando aplicaciones Master/Worker y *Single Program Multipla Data (SPMD)*, con comunicaciones basadas en sockets y en paso de mensajes *Message Passing Interface* (MPI). Las ejecuciones se realizaron en un cluster multicore, obteniendo los niveles deseados de funcionalidad y de prestaciones.

Palabras clave: Aplicaciones Paralelas, tolerancia a fallos, paso de mensajes, RADIC, socket, Alta Disponibilidad.

Català

La demanda de major rendiment de les aplicacions científiques es satisfà incrementant la quantitat de components. No obstant això, un major nombre de components implica una major probabilitat de fallada. L abrupta caiguda dels temps mitjans entre fallades en els sistemes actuals impulsa la investigació de mecanismes de tolerància a fallades per garantir l execució d una aplicació a un cost raonable.

Message-Passing Interface (MPI), l estàndard de programació més utilitzat per les aplicacions científiques, té un comportament *fail-stop*, realitzant una parada segura de tots els processos en cas de detectar una fallada en qualsevol dels nodes del clúster. Com a consequència, es perd l execució que s hagués fet en tots els nodes de processament.

Els sistemes de còmput d altes prestacions, han anat implementat mecanismes per a garantir el servei, normalment basades en técniques de *rollback-recovery* mitjançant l ús de *Checkpoint/Restart*. Aquestes solucions s han implementat a nivell d aplicació, la qual cosa no és transparent, o bé, a nivell de llibreria, la qual cosa no és generalitzable a altres llibreries i es deixen fora del camp de solució a un divers nombre d aplicacions.

Es proposa un sistema de tolerància a fallades transparent i automàtic per l aplicació paral·lela de manera que pugui utilitzar-se sense modificar l aplicació i amb la llibrería de pas de missatge que prefereixi l usuari. Es basa en detectar els errors en las comunicacions de *sockets* causats per les fallades de nodes i reconfigurar-los en forma automàtica per a comunicar-se amb la nova adreça a on es migra el procés. Funciona en conjunt amb un sistema que protegeix l estat de còmput dels processos i, en cas de fallades, els recupera en un altre node de còmput mitjançant tècniques de *rollback-recovery*.

S ha realitzat una validació experimental utilitzant aplicacions *Master/Worker* i *Single Program Multipla Data (SPMD)* amb comunicacions basades en sockets i en pas de missatges *Message Passing Interface* (MPI).

Les execucions es van realitzar en un clúster multicore, obtenint els nivells desitjats de funcionalitat i prestacions.

Paraules clau: Aplicacions paral·leles, tolerància a fallades, passa de missatges, RADIC, socket, Alta Disponibilitat.

English

The demand of more performance of scientific applications is achieved by increasing the amount of components. However, a growing number of components implies that the probability of failure increases as well. The remarkable decrease of average times between failures in the current High Performance Computing systems encourages the investigation of mechanisms of fault tolerance suitable for new architectures which allow to guarantee the execution of an application at a reasonable cost.

Message Passing Interface (MPI), the standard of programming more used by scientific application, has a fail-stop behavior, by carrying out a safe stop of all the processes in case of detecting a failure in any of the nodes of the cluster. As a consequence, the execution which could have been done in all the processing nodes until that moment is lost.

High Performance Computing has implemented mechanisms in order to guarantee service, usually based on techniques of rollback-recovery by using the Checkpoint/Restart. Those solutions have been implemented at an application level which is not transparent, or, at library level, which is not extended to other libraries and leave out several applications.

A transparent and automatic fault tolerance system in proposed in this thesis, in such a way that the application can be used without being modified and with the message passing library preferred by the user. It is based on detecting failures in the communications of the *socket* caused by failures of nodes and reconfigure them in an automatic way to communicate with the new direction where the process is migrated. This method works along with a system which protects the status of computation of the processes and in the case of failure, they are recovered in other node of computation by using techniques of rollback-recovery.

An experimental validation has been carried out by using applications Master/Worker and Single Program Multipla Data (SPMD), with communications based on sockets and on Message Passing Interface (MPI). The executions were made in a multicore cluster, obtaining the desirable levels of functionality and performance.

Keywords: Parallel applications, fault tolerance, message passing, RADIC, socket, High Availability.

Agradecimientos

En primer lugar quiero agradecer especialmente al Dr. Emilio Luque y a mi directora Dr. Dolores Rexachs por haber confiado en mi y por darme la oportunidad de realizar este doctorado. Ambos son mi guía y ejemplo a seguir por su quehacer profesional y personal.

A Nicolás, por brindarme su amor, confianza y acompanarme sin condicionantes en todo momento.

A mis padres y hermanos por darme apoyo, carino, y por animarme a superarme.

A mis amigos por haberme comprendido, apoyado y alentado a iniciar y seguir en este proyecto. Especialmente quiero agradecer a Maria Jesús, que me ha brindado un soporte en este trabajo que sólo personas de una altísima calidad personal y profesional son capaces de ofrecer. Me siento orgullosa de las personas que forman mi círculo íntimo, y a todos ellos les dedico mi trabajo.

Quiero expresar mi agradecimiento a todos los miembros del Departamento de Arquitectura y Sistemas Operativos, por darme soporte para realizar este trabajo.

A la escuela Gimbernat, quiero agradecerle por haberme brindado la oportunidad de dedicarme a la docencia y a la investigación.

El desarrollo de este trabajo no hubiera sido posible sin los aportes del grupo de investigación, muchas gracias por vuestro interés, por la participación en las discusiones y todas vuestras sugerencias.

Y por último, agradezco a todos y a cada uno de los companeros de ambos laboratorios, por haberme dado consejos, ánimos y soporte durante todo este tiempo.

Índice general

Índice de figuras

Capítulo 1

Introducción

> *"La mayoría de las ideas de la ciencia son esencialmente sencillas y, por regla general pueden ser expresadas en un lenguaje comprensible para todos."*
>
> Albert Einstein

La computación de altas prestaciones está actualmente soportada por clústeres de cientos de procesadores. La demanda de mayor rendimiento de las aplicaciones científicas normalmente se satisface incrementando aún más el número de componentes. Sin embargo, el riesgo de tener un fallo aumenta. Si bien el tiempo medio entre fallos (*Mean Time Between Failures MTBF*) de cada una de las partes, sean estas fuentes de alimentación, ventiladores, placas de red, por mencionar algunos, es alto, el clúster puede fallar frecuentemente debido a tener un gran número de dichos componentes. [1].

Cuando uno de los nodos del clúster, donde se ejecuta una aplicación paralela de paso de mensaje, tiene un fallo, además de afectar a los procesos de ese nodo, se caen las comunicaciones establecidas con otros nodos. El resto de procesos paralelos sufren un error de comunicaciones, y si no tienen una estrategia para tolerar dicho fallo, en general, se cancela la ejecución, con la consecuente pérdida del trabajo realizado y la espera de un nuevo lanzamiento de la aplicación. Este comportamiento puede significar muchas pérdidas de horas de cómputo y esto se agrava si consideramos que las aplicaciones tienden a crecer en número de procesos y en nodos de clúster utilizados.

Message-Passing Interface MPI [2] es el estándar de programación más utilizado por las aplicaciones científicas. Sin embargo, este no especifica mecanismos de tolerancia a fallos y tiene un comportamiento *fail-stop*, explicado por Schlichting [3], realizando una parada segura de todos los procesos en caso de detectar un fallo en cualquiera de los nodos del clúster. Como consecuencia, se pierde la ejecución que se hubiera hecho hasta el

momento, se ha de realizar un tratamiento manual para solucionar la causa del problema y finalmente, reejecutar la aplicación.

Para evitar este problema se utilizan estrategias de tolerancia a fallos por medio de las cuales las aplicaciones paralelas finalizan su ejecución correctamente a pesar de que ocurran fallos en alguno de sus nodos.

En los últimos anos, proyectos de investigación como MPICH-V [4], FT-MPI [5]. RADIC [6], Open-MPI [7], MPI-FT [8], MPI/FT TM [9] [10], Egida [11]. LA-MPI [12], LAM-MPI [13] o *Fault tolerance in Message Passing Interface programs* [14] han desarrollado distintas aproximaciones, basándose en el paradigma de *rollback-recovery* que es el más utilizado en aplicaciones paralelas. Los protocolos propuestos por este paradigma anaden una sobrecarga durante la ejecución de la aplicación cuando se realiza la protección, y en la recuperación en caso de fallos.

La lógica de estos protocolos puede incluirse a nivel de la aplicación, donde es posible conseguir una mejor adaptación, anadiendo una menor sobrecarga, debido al conocimiento que se tiene, en este nivel, del estado de procesamiento y de las estructuras de datos en el momento de detectar un fallo. Sin embargo, el coste de desarrollo, de test y la necesidad de los códigos fuentes, hace que esta alternativa no se utilice frecuentemente y que la mayoría de aplicaciones paralelas no incorporen estrategias de tolerancia a fallos.

Para dar respuesta a estos casos, líneas de investigación proponen incluir los protocolos de tolerancia a fallos en la librerías de paso de mensaje, con un funcionamiento transparente a la aplicación para que no tenga que ser modificada. Sin embargo, con el desarrollo de la programación paralela una variedad de fabricantes de software ofrecen librerías de paso de mensajes. Si consideramos *Message-Passing Interface MPI* [2], existen múltiples versiones que dan respuesta a la evolución de cada implementación, Open-MPI [7], MPICH-V [4], MVAPICH [15], entre otras, y, a la del estándar que ya ha aprobado su tercera versión. Cuando el *rollback-recovery* se ha incorporado en este nivel, necesita ser mantenido en cada implementación y en cada versión. Otra vez, esto puede ser costoso. Pero además, la variedad de librerías hace que en cada sistema de cómputo paralelo coexistan múltiples versiones para satisfacer preferencias de usuarios de aplicaciones y administradores. En este escenario, no es factible aplicar una política general de tolerancia a fallos para el sistema de cómputo si cada librería define un comportamiento diferente de tolerancia.

En cambio, si la gestión de tolerancia a fallos se encuentra a nivel del sistema operativo, el servicio puede ser transparente para la librería y para la aplicación, entonces ambas actúan como si se estuviesen ejecutando en un computador virtual paralelo libre de fallos.

En realidad, cuando ocurre un fallo en uno de los nodos del clúster, dónde se ejecuta una aplicación paralela, el resto de procesos que estuvieran conectados con procesos en dicho

nodo manifiestan un error. La figura 1.1 muestra un diagrama de capas de comunicaciones de una aplicación de paso de mensajes. Un fallo en el nivel físico o en los niveles de red propaga el error a los niveles superiores causando una parada de ejecución. *Socket* [16] es una interfaz de aplicación *Application Program Interface (API)* estándar *de facto* de los sistemas *Portable Operating System Interface* (*POSIX*) para realizar el transporte de comunicaciones entre dos procesos utilizando protocolos del tipo *Transmission Control Protocol (TCP)* o *User Datagram Protocol (UDP)* [17]. Los sockets constituyen una interfaz básica para las aplicaciones que funcionan en red. Cuando una de las partes de la comunicación tiene un fallo, la funciones de *socket* devuelven un error.

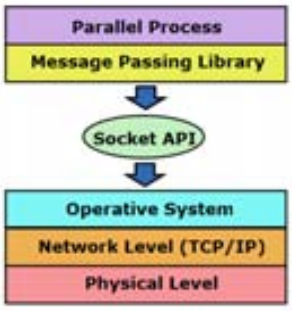


Figura 1.1: Socket Level

Nuestra propuesta consiste en cambiar el comportamiento de la interfaz de *socket* de modo que exista un sistema de tolerancia que detecte el fallo y que cuando éste se produzca, se reconfigure la comunicación utilizando una nueva dirección IP (*Internet Protocol*), a donde se migra el proceso que ha fallado, sin interrumpir la ejecución de la aplicación. Utilizando esta nueva dirección IP, se restablece la comunicación con el proceso remoto ya recuperado. Si además, dentro de la API de *socket*, incluimos las tareas de protección y de recuperación de procesos manteniendo la interfaz estándar, obtenemos una solución transparente y automática de tolerancia a fallos para aplicaciones y librerías de paso de mensajes que utilicen *socket*.

El diagrama de estados de la API *socket* se cambia para que automáticamente detecte fallos en la comunicación, incluyendo los causados por fallos en el vínculo, largos períodos de desconexión o migración de procesos. Cuando este tipo de errores ocurre, en lugar de cerrar el *socket* de forma inesperada, se averigua la nueva dirección a donde se migrarán los procesos, se reemplaza la dirección IP y la comunicación se recupera, sin pérdida de datos. A este tipo de funcionamiento capaz de recuperarse de un fallo, ya sea local o remoto, lo denominamos *socket seguro*.

Los principales protocolos de *rollback-recovery* para aplicaciones de paso de mensaje están explicados, analizados y clasificados por Elnozahy [18]. El análisis de cuál es el

protocolo más adecuado es una tarea muy compleja, ya que depende de los objetivos que se quieran conseguir, como por ejemplo, menor sobrecarga en tiempo de ejecución, menor sobrecarga de uso de ancho de banda o escalabilidad, y, para conseguir estos objetivos, múltiples factores tienen que ser tenidos en cuenta, como las características de la aplicación, las del sistema, la tasa de fallos y los parámetros del mismo protocolo a aplicar, como el intervalo de checkpoint.

RADIC, *Redundant Array of Distributed Controllers* [19] [20] [21], es una arquitectura de tolerancia a fallos para aplicaciones de paso de mensajes que define los modelos de protección de estado de ejecución, detección de fallos, recuperación de procesos y reconfiguración del sistema. Estas gestiones son llevadas a cabo por componentes distribuidos en el clúster de un modo descentralizado. Utiliza un protocolo *rollback-recovery* de log de mensajes pesimista basado en receptor, en conjunto con checkpoints no coordinados, manteniendo así el diseno distribuido. Esta propiedad es clave para no afectar la escalabilidad de la aplicación que se protege.

Los sistemas actuales de altas prestaciones están compuestos en general por procesadores *multicore* y *manycore*. Cuando los procesos paralelos que se comunican por mensajes se ejecutan en un mismo nodo se ven beneficiados por la disminución de la latencia y del ancho de banda de la comunicación intra-nodo. Los procesos que residen en un mismo nodo no fallan en forma independiente, teniendo que ser reiniciados y reejecutados hasta el mismo punto antes del fallo. Esta dependencia ya fue estudiada en la literatura proponiendo utilizar protocolos híbridos para adaptarse y evolucionar a estas nuevas arquitecturas [22] [23] [24] [25] [26] [27]. Nuestro trabajo disena e implementa en la capa de *socket seguro* un protocolo semi-coordinado por el cual los procesos en un nodo se coordinan para realizar checkpoint evitando así el log de mensajes entre estos procesos. En comunicaciones de procesos que residen en nodos diferentes se aplica el protocolo pesimista basado en receptor.

En síntesis, la estrategia de tolerancia a fallos que propone y desarrolla este trabajo es un modelo para dar soporte de tolerancia a fallos independiente y transparente a la librería de paso de mensajes y la aplicación. Se basa en modificar el comportamiento de los sockets usados por las aplicaciones paralelas de paso de mensaje para la comunicación entre procesos y las librerías de niveles superiores que se indican en la figura 1.1.

El nuevo modelo combina el uso de sockets seguros con los modelos de protección, detección y recuperación de la arquitectura RADIC, adaptada a clústeres multicore, donde varios procesos se ejecutan en el mismo nodo. Al ubicarse en esta capa, se consigue una solución independiente y transparente a la librería de paso de mensajes y a la aplicación.

Se incorpora toda la gestión para dar soporte a un protocolo de checkpoint semi-

coordinado que disminuye la sobrecarga de la tolerancia en sistemas multicore y la inclusión de las primitivas utilizadas por las librerías MPI.

La validación experimental se ha realizado utilizando los patrones de aplicaciones científicas más comunes como son los de *Master/Worker* (M/W) y *Single Program Multiple Data* (SPMD).

1.1. Motivación y Objetivos

El aumento de incidencias debido a fallos de componentes en el hardware con la consecuente pérdidas de horas de ejecución ha provocado un creciente interés en el área de tolerancia a fallos en los últimos anos. Las estrategias se han ido desarrollando ya sea ampliando las funciones en las aplicaciones o mediante uso de librerías especializadas en conjunto con las librerías de paso de mensaje. El primer tipo de estrategia es costoso y se necesita la aplicación, y el segundo no puede generalizarse al resto de librerías, de las cuales hay cada vez más.

Una solución de tolerancia a fallos que sea independiente de la librería de paso de mensajes, transparente a la aplicación y escalable es la principal motivación de este trabajo. De este modo, un conjunto más amplio de aplicaciones paralelas de paso de mensaje puede alcanzar el uso de mecanismos de tolerancia a fallos, debido a que esta estrategia no les requiere cambios.

El principal objetivo de esta tesis es proponer un modelo de tolerancia a fallos que permita que una aplicación de paso de mensajes continúe y finalice correctamente su ejecución cuando se presenten fallos en los nodos de procesamiento, sin realizar cambios en la aplicación así como tampoco en la librería de paso de mensaje que se este utilizando. El modelo es para aplicaciones paralelas de paso de mensaje que utilicen el estándar MPI ejecutadas en clústeres multicore, debido a que este es el entorno actual en el contexto de nuestra investigación.

Las contribuciones de esta tesis están relacionadas con la consecución de los objetivos planteados en la sección anterior. En esta tesis se propone:

- Un modelo de *socket seguro* que permite enmascarar los fallos de comunicaciones a aplicaciones y librerías basadas en *socket* en forma transparente.
- Un sistema de tolerancia a fallos distribuido, descentralizado y transparente en capa de socket seguro que utiliza un protocolo pesimista basado en receptor y checkpoint no coordinado.

- Un protocolo semicoordinado para sistemas multicore integrado en forma híbrida en el sistema de tolerancia a fallos. Se coordinan los checkpoints de los procesos que se ejecutan en el mismo nodo, evitando así hacer el log de las comunicaciones entre dichos procesos y se mantiene el log de mensajes basando en receptor entre procesos en distintos nodos.

- Extensión del modelo de tolerancia a fallos para el uso de aplicaciones paralelas basadas en MPI y evaluación experimental utilizando la implementación Open-MPI, una de las de mayor grado de aceptación en este campo. Se integran al modelo un conjunto adicional de funciones de sockets que son aplicadas por las librerías MPI para hacer una gestión eficiente de las comunicaciones múltiples.

1.2. Organización de la tesis

La tesis está organizada en los siguientes capítulos:

Capítulo 2: Estado del Arte.

Este capítulo introduce conceptos básicos sobre la tolerancia a fallos en sistemas de altas prestaciones y las técnicas de *rollback-recovery* que se utilizan para las aplicaciones paralelas de paso de mensaje. Luego, se presentan los trabajos relacionados, describiendo y comparando nuestra propuesta con otras arquitecturas y herramientas de tolerancia a fallos.

Capítulo 3: Tolerancia a fallos en la capa de socket basada en RADIC.

Este capítulo explica en detalle el modelo de *socket-seguro* y la adaptación de los modelos de protección, detección, recuperación y reconfiguración de RADIC a esta capa.

Capítulo 4: Protocolo de Checkpoint Semi-Coordinado.

En este capítulo, desarrollamos el protocolo de checkpoint semi-coordinado, propuesto para disminuir la sobrecarga que produce el protocolo pesimista en los sistemas multicore durante la fase de protección.

Capítulo 5: Computador Paralelo libre de fallos para aplicaciones MPI.

Este capítulo analiza y describe las adaptaciones que se han realizado al sistema de tolerancia a fallos en capa de socket para que pueda utilizarse con aplicaciones de paso de mensaje basadas en Message Passing Interface.

Capítulo 6: Conclusiones.

Se enumeran las conclusiones de la tesis y se presentan las líneas futuras.

Capítulo 2

Estado del Arte

En la primera parte de este capítulo, se presentan conceptos básicos sobre la tolerancia a fallos en sistemas de altas prestaciones y los protocolos de *rollback recovery* definidos y utilizados en la literatura. Se continúa con una discusión acerca de la capa a donde se anaden dichos protocolos. Por último, se explica la arquitectura de tolerancia a fallos para aplicaciones de paso de mensaje RADIC, cuyos modelos de protección, detección y recuperación han sido utilizados en el presente trabajo de investigación para ser incorporados en la capa de socket para conseguir una solución de tolerancia a fallos independiente de la librería de comunicaciones de paso de mensajes.

2.1. Tolerancia a Fallos en sistemas de Altas Prestaciones

2.1.1. Introducción

Las tasas de fallos en componentes de hardware de los sistemas actuales aumentan, tal como lo refleja el estudio realizado por Schoeder y Gibson [28] en base a las incidencias ocurridas durante nueve anos en el centro de investigación de Los Álamos. Entre el 40 % y 80 % de las incidencias, dependiendo del tamano del cluster, son causadas por un problema del nodo.

El estudio de la tolerancia a fallos no es un tema nuevo, sino que se viene investigando desde hace ya muchos anos, y desde la década del 70, [29], se le ha ido dando cada vez más importancia. Sin embargo, la evolución de los computadores y sus aplicaciones hace que estas técnicas se mantengan en constante estudio para tener mejoras evolutivas y adaptativas a los sistemas emergentes.

En los últimos anos, varios investigadores alertan acerca de la necesidad de desarrollar nuevos mecanismos de tolerancia a fallos más eficientes y adecuados a las nuevas arquitecturas.

Elnozahy [30] advierte que la práctica actual de checkpoint no será posible en los en los sistemas *peta-scale*, y, como los fallos serán más frecuentes, propone que se asignen recursos del sistema para las tareas de *rollback-recovery*.

Capello [31] indica la necesidad de mejorar los mecanismos actuales de *rollback-recovery* para reducir los tiempos y mejorar la escalabilidad y propone que se utilicen nuevos enfoques como los algoritmos basados en tolerancia a fallos, *forward-recovery*, operación pro-activa, ejecución especulativa y memoria transaccional.

Chakravorty [32] propone un esquema de tolerancia a fallos con una sobrecarga reducida durante la ejecución sin fallos y evitando la recuperación de procesos que no están directamente afectados por un fallo. Utiliza un protocolo de *rollback-recovery* de log de mensajes basado en emisor. Esta basado en Charm++ [33] y en Adaptive MPI [34].

La estrategias de checkpoint sin disco y tolerancia a fallos basada en algoritmos es la propuesta de Kuanching [35] para solucionar los problemas de escalabilidad en sistemas multicores.

El aumento de incidencias causadas por fallos de componentes de hardware en los nuevos sistemas, hace que un mayor número de aplicaciones requieran ser dotadas de mecanismos de tolerancia de fallos.

Los sistemas actuales ofrecen soluciones parciales a las necesidades de automatización de los mecanismos de tolerancia en las aplicaciones de altas prestaciones. Normalmente, un servicio de alta disponibilidad, se soporta mediante redundancia en los sistemas o checkpoint y soporte de operaciones 24/7. Esto implica la presencia de operadores que pueden detectar y reparar fallos y reanudar las ejecuciones en forma manual. Este servicio tiene un alto coste y, por ello, no factible en muchos casos [36].

La distribución de recursos es la causa principal de la complejidad de la gestión de la tolerancia a fallos, considerando que se necesitan operaciones remotas, transferencias, detección de fallos y mecanismos de recuperación mediante consenso entre las partes involucradas.

2.1.2. Alta Disponibilidad por redundancia

Un sistema robusto y con alta disponibilidad posee a la vez, tres propiedades conocidas como *RAS* por su acrónimo en inglés, de acuerdo a lo que indica Hwang et.al en [37].

1. Fiabilidad *(Reliability)*: es la capacidad de un sistema de funcionar correctamente

y representa el tiempo que opera sin caerse .

2. Disponibilidad *(Availability)*: es el porcentaje del tiempo que un sistema esta disponible para el usuario.

3. Capacidad de Servicio*(Serviceability)*: se refiere a cuan fácil es el servicio, incluyendo hardware, software, mantenimiento, reparaciones, y actualizaciones.

La fiabilidad se mide por el tiempo medio entre fallos *Mean time to failure MTTF*, mientras que la capacidad de servicio se mide por el tiempo medio de reparación *Mean Time to Repair MTTR*. La disponibilidad de un sistema es definida por:

$$Disponibilidad = MTTF/(MTTF + MTTR) \tag{2.1}$$

Se deduce que hay dos maneras de aumentar la disponibilidad. La primera es incrementar el tiempo medio hasta el fallo $MTTF$, y la segunda, es disminuir el tiempo medio de reparación $MTTR$.

Estudios recientes muestran que el tiempo medio hasta el fallo $MTTF$ en los clústeres disminuye debido al aumento de componentes. Por lo tanto, la estrategia para aumentar la disponibilidad de los computadores paralelos consiste en disminuir el tiempo de reparación $MTTR$.

La propuesta que presentamos en este trabajo de investigación es transparente al usuario y al administrador del sistema, por lo que no necesita intervención humana, y se activa en forma automática, reduciendo de este modo el tiempo de reparación $MTTR$.

2.1.3. Tolerancia a Fallos en Hardware

La tolerancia a fallos es la habilidad de un sistema de realizar su función correctamente aún en presencia de fallos [36]. Un fallo es un evento que ocurre cuando el servicio se desvía del correcto funcionamiento, y la causa puede ser desde un problema de implementación del software, un mal funcionamiento del hardware o acciones del usuario. Un error es la consecuencia del fallo, su manifestación cuando llega a la interfaz del usuario.

El presente trabajo está relacionado con fallos permanentes de hardware, como por ejemplo, de la fuente de energía, de la placa de red o del disco local. La técnica más usada para mejorar la disponibilidad del sistema en estos casos consiste en usar redundancia de componentes. Así, cuando uno de estos falla, el control es tomado por un sustituto. La condición es que estos componentes sean independientes, en el sentido que no pueden

estar sujetos a la misma causa del fallo. El inconveniente del uso de la redundancia de componentes independientes es el coste de los mismos y el consumo de energía [38].

Configuración de clústeres tolerantes a fallos Los modelos de alta disponibilidad usados en clústeres entre dos nodos ordenados en forma ascendiente por su grado de disponibilidad son [37]:

1. *Hot Standby server cluster*: En este caso, sólo el nodo primario está activo realizando todo el trabajo. El nodo secundario *standby* ejecuta programas de monitorización pero no tiene carga de trabajo y necesita una copia de los datos para comenzar su actividad.

2. *Active-takeover clusters*: Ambos servidores son primarios y realizan el trabajo normalmente, y soportan las operaciones de *failover* para pasar el servicio al componente redundante en caso de fallos y de *failback* cuando el componente está reparado. Durante la ejecución de estas operaciones se puede observar una demora y en algunos casos, una pérdida de datos que no se hayan podido guardar.

3. *Failover cluster*: Cuando un componente falla, esta técnica permite que los demás retomen el servicio que proveía el componente que ha fallado. El mecanismo de *failover* requiere funciones de *diagnóstico, notificación y recuperación*. El diagnóstico se refiere a la detección y localización del fallo, y la técnica más usada es el *heartbeat*. Cuando no se recibe la senal desde un nodo, se concluye que nodo o la red han fallado.

Esquemas de recuperación La recuperación se refiere a las tareas que se han de realizar para retomar el trabajo de un componente que ha fallado. Hay dos tipos de técnicas [37]:

1. *Backward recovery*: Los procesos salvan periódicamente un estado consistente normalmente llamado *checkpoint* en un almacenamiento estable. Luego de un fallo, el sistema es reconfigurado para aislar el componente que ha fallado, y se realiza un *rollback*, recuperando el checkpoint previamente guardado y reanudando la ejecución. Esta técnica es relativamente fácil de implementar sobre todo en aplicaciones independientes pero implica una pérdida de ejecución.

2. *Forward recovery*: Cuando el tiempo de ejecución es crucial, el *rollback* no es un opción posible. En esta técnica, se utiliza la información de diagnóstico para recuperar un estado válido y continuar la ejecución.

2.2. Rollback Recovery Protocols

Las aplicaciones paralelas de paso de mensaje en su mayoria estan desarrolladas utilizando el estándar *de facto Message-Passing Interface MPI* [2] para la programación de paso de mensaje. El comportamiento estático definido por este estándar es que todos los procesos paran su ejecución aunque sólo uno de sus procesos esté afectado por un fallo. Este comportamiento se conoce como *(fail-stop)* [3].

Las estrategias propuestas por la literatura para este tipo de aplicaciones se basan en utilizar protocolos de *rollback-recovery*, debido a que por el momento han demostrado ser los más adecuados.

Estos protocolos han sido estudiados [18], [39] y comparados en la literatura, según la sobrecarga que tienen durante la protección y la recuperación. Constituyen una técnica usada para reducir el tiempo de cómputo que se pierde cuando hay un fallo y se recupera a un estado previo de la computación. Las aplicaciones realizan checkpoint durante la operación libre de fallos, denominada fase de protección, y lo graban en un almacenamiento estable. Un almacenamiento estable es un dispositivo lógico que sobrevive a los fallos, normalmente representado como un servidor de ficheros centralizado, aunque puede considerarse otro dispositivo con una probabilidad de fallo independiente al que se está protegiendo. Las técnicas de tolerancia a fallos tienen un coste de rendimiento debido a que anaden tareas a las de su ejecución normal. El tiempo de ejecución adicional se lo denomina sobrecarga. La latencia del checkpoint es el tiempo que se necesita para crear y guardar el checkpoint en el almacenamiento estable.

En esta sección, describiremos los protocolos relacionados con el presente trabajo, checkpoint coordinado 2.2.1, checkpoint no coordinado 2.2.2, log de mensajes 2.2.3 y protocolos híbridos 2.2.4.

2.2.1. Checkpoint Coordinado

En un checkpoint coordinado todos los procesos de la aplicación paralela guardan su estado en forma simultánea. Para asegurar una línea de recuperación válida, es necesario eliminar los mensajes en tránsito y con ello, se evitan los mensajes huérfanos durante la recuperación. Varios algoritmos se han propuesto, siendo los más usuales los de Chandy-Lamport [40] y el checkpoint coordinado bloqueante [41]. La ventaja de esta técnica es que el checkpoint es la única sobrecarga por la protección durante la ejecución libre de fallos, mientras que la desventaja es que requiere que todos los procesos realicen *rollback* a su último checkpoint, aunque no estén afectados por el fallo.

DMTCP [42] (*Distributed Multithreaded Checkpointing*) es una herramienta de check-

point y restart a nivel de usuario para ser utilizada con aplicaciones distribuidas y paralelas. El estado global de la aplicación es guardado incluyendo sus sockets de comunicaciones. El protocolo utilizado es del tipo checkpoint coordinado.

2.2.2. Checkpoint No Coordinado

Al utilizar checkpoint no coordinado cada proceso paralelo elige el momento más conveniente para esta operación. De este modo, la sobrecarga de la ejecución libre de fallos podría solaparse con la ejecución de otros procesos. Se evitan las tareas y el tiempo de coordinación, aunque, si se utiliza como única solución, la recuperación puede requerir que los diversos procesos realicen *rollback* en efecto dominó pudiendo así llegar incluso a la pérdida de todo el cómputo realizado. Además, es necesario guardar todos los checkpoints de todos los procesos.

Para evitar el efecto dominó de *rollback* que puede ocasionar las mensajes huérfanos, Chandy [40] y Koo [43] han propuesto metodologías para determinar el estado global consistente.

Nuestra propuesta utiliza checkpoint no coordinado de modo que el estado de cada proceso se guarda y se recupera en forma autónoma, evitando las tareas de coordinación de las técnicas anteriores.

En la próxima sección se explica como se combina esta técnica con log de mensajes para establecer un estado consistente global y sin pérdida de mensajes.

Utilizamos la librería de Berkeley Lab Checkpoint/Restart (BLCR) [44], una implementación de checkpoint/restart a nivel de sistema para clústeres Linux realizada para aplicaciones de altas prestaciones incluyendo las de paso de mensaje.

Las librerías de paso de mensaje más utilizadas como Open-MPI [7], MPICH-V [4], MVAPICH [15] y LAM/MPI [13] han seleccionado BLCR para integrar sus propuestas de tolerancia a fallos. Funciona a nivel de kernel, opción que proporciona ventajas de integración y mejor funcionamiento que las ubicadas a nivel de usuario [45]. Hay que tener en cuenta que BLCR no considera las comunicaciones que el proceso tenga abiertas en el momento de guardar el estado, siendo responsabilidad del usuario restablecerlas en caso de *restart*.

2.2.3. Log de Mensajes

Los protocolos de log de mensajes se basan en modelar la ejecución de los procesos como una secuencia de eventos deterministas que ocurren entre eventos no deterministas [46]. Un evento no determinista, es, por ejemplo, la recepción de un mensaje. Estos eventos

se identifican y guardan en un almacenamiento estable durante la ejecución libre de fallos. Luego, el estado del proceso puede ser recuperado mediante la reejecución de los eventos en el mismo orden [47]. Normalmente, se utiliza en combinación con checkpoint no coordinado de cada proceso paralelo para reducir la duración de la recuperación. En caso de fallos, sólo los procesos afectados realizan *rollback recovery*, y luego se reejecutan realizando un *forward* hasta el momento del fallo. Estos protocolos se clasifican en [18]:

1. **Pesimistas:** Asume que el fallo puede ocurrir inmediatamente después de un evento no determinista y por lo tanto, ningún proceso puede depender de este evento hasta que no este guardado en almacenamiento estable. Se guardan los eventos en forma sincrónica, aumentando la sobrecarga durante la ejecución libre de fallos. Así, se garantiza que no se generan mensajes huérfanos, es decir, en el momento de la recuperación, se disponen de todos los mensajes que el resto de procesos le hubieran enviado. La figura 2.1 muestra un diagrama de tres procesos P0,P1,P2. P0 tiene los determinantes m0,m4,m7, P1 a m1,m3,m6 y P2 a m2,m5. Estos determinantes garantizan que los procesos P1 y P2 se puedan reproducir en caso de fallos desde los checkpoint B y C respectivamente.

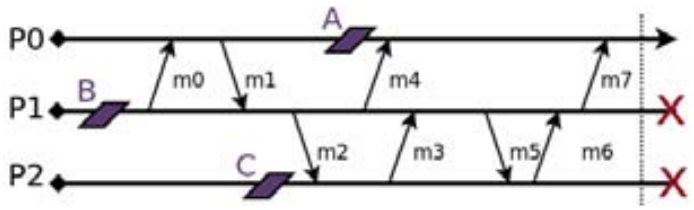


Figura 2.1: Log de Mensajes Pesimista

2. **Optimistas:** Se guardan los eventos no deterministas en forma asincrónica, asumiendo que el log se finaliza antes de un posible fallo. Reducen la sobrecarga durante la ejecución libre de fallos, pero pueden aparecer mensajes huérfanos. Esto complica el protocolo de recuperación y el borrado de mensajes. La figura 2.2 muestra un diagrama de tres procesos P0,P1,P2. Si P2 falla antes que m5 se guarde en almacenamiento estable, se necesita hacer rollback de P1 además de P1 por el mensaje huérfano. Esto puede provocar un efecto dominó.

3. **Causales:** Evitan el acceso sincrónico al guardar los eventos en almacenamiento estable, pero se generan copias de mensaje adicionales para evitar los mensajes huérfanos. Combinan las ventajas de los anteriores, pero mantienen las desventajas de los protocolos optimistas en el sentido que requiere un algoritmo de recuperación más complejo.

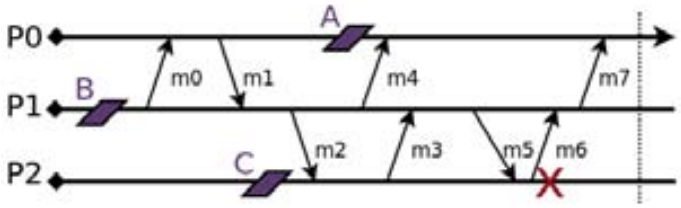


Figura 2.2: Log de Mensajes Optimista

Estos protocolos pueden ser basados en receptor o en emisor según si el log del evento no determinista es considerado en recepción o en emisión respectivamente. En el presente trabajo se utiliza un protocolo pesimista basado en receptor.

Aunque estos protocolos permiten que cada proceso salve su estado en forma autónoma evitando tiempos adicionales de coordinación, el hecho de guardar cada mensaje anade una sobrecarga que puede incrementar el tiempo de ejecución de la aplicación.

Hay trabajos de investigación que estudian y comparan el funcionamiento y sobrecarga de estos protocolos como en [48] [49], [50]. En [51] se compara el protocolo coordinado con un log de mensaje pesimista basado en emisor implementado en MPICH-V [4]. Se pueden conseguir mejores tiempos durante la operación libre de fallos utilizando protocolos pesimistas basado en emisor [52] pero aumentando la latencia y complejidad de la recuperación, o bien, combinar ambos como propone Rao [50]. Sin embargo, el rendimiento finalmente depende de los diferentes sistemas y características de la aplicación así como de los parámetros utilizados en el protocolo. Por lo cual, es importante elegir un protocolo que mejor se adapte al sistema, aplicación y necesidades de fiabilidad [53].

2.2.4. Protocolos Híbridos

La evolución de las arquitecturas de los sistemas de altas prestaciones a *multicore* y *manycore* hace que los protocolos de *rollback-recovery* de aplicaciones de paso de mensajes también tengan que evolucionar para adaptarse a estas tecnologías.

Por un lado, los protocolos pesimistas requieren que se realice log en forma sincrónica durante la operación libre de fallos, siendo esto un inconveniente muy difícil de aceptar en comunicaciones entre procesos de un mismo nodo *multicore*, debido a que la latencia efectiva de cada mensaje será la velocidad entre nodos en lugar de la interna.

Aunque los checkpoints coordinados evitan la sobrecarga del log, no son escalables. Cuando se crece en número de procesos, la coordinación global durante la ejecución libre de fallos y el rollback en caso de fallos aumenta a niveles no aceptables [54] [30].

Yang [24] propone dividir el sistema en *clusters* de modo que se minimice la sobrecarga

del log y de la coordinación de checkpoints, teniendo también en cuenta una recuperación eficiente.

Por otra parte, Boutellier en [22] realiza una coordinación entre procesos que se ejecutan en el mismo nodo *multicore* mientras que aplica un protocolo pesimista basado en emisor para proteger mensajes entre procesos ubicados en diferentes nodos. Se fundamente en que los procesos en un mismo nodo se relacionan ya que cuando hay fallos se reinician y reejecutan al mismo tiempo.

En nuestro trabajo, proponemos utilizar un protocolo de checkpoint semi-coordinado para la ejecución en arquitecturas *multicore*, el cual también coordina a los procesos que ejecutan en un mismo nodo para hacer checkpoint combinando con un protocolo pesimista, pero basado en recepción. Al grabar el evento en recepción, la recuperación de cada proceso funciona en forma independiente, disponiendo de todos los mensajes recibidos sin ser necesario solicitar rollback de otros procesos. Otra diferencia es la validación en el trabajo de Boutellier se realiza utilizando protocolos *rollback-recovery* en la capa de Open-MPI [7] mientras que en nuestra propuesta se usa la capa de socket [55] [56].

HOPE [23] es un protocolo híbrido que combina checkpoint inducido por comunicación CIC [18] con un protocolo pesimista. Como esta orientado a ambientes GRID, el criterio de agrupación utilizado está basado en la red y en el patrón de comunicación. El checkpoint inducido por comunicaciones no es eficiente ni escalable entre procesos con un alto grado de acoplamiento en la comunicación debido a que el número de checkpoints crece en forma no controlada.

Como una estrategia para reducir el tiempo de coordinación y de checkpoint, Gao [25] propone crear grupos de coordinación y solapar los checkpoint entre ellos. Sin embargo, se necesitará un rollback de todos ellos en caso de fallos. Se valida utilizando MVAPICH2 [15].

Rao [26] propone un protocolo de checkpoint coordinado y no coordinado con log de mensajes pesimista basado en emisor durante los intervalos de la coordinación de un estado global consistente. Si bien se disminuye la sobrecarga de la protección, no soluciona los problemas de la escalabilidad del checkpoint coordinado durante la recuperación y puede ser ineficiente en casos de procesos con comunicaciones intensas en cantidad o volumen.

Otra estrategia de agrupar procesos para hacer *checkpoint* combinada con log de mensajes es la utilizada en [27]. Los procesos son asociados de acuerdo al patrón de comunicaciones. Se ofrece una herramienta de trazado para la creación de grupos. En nuestro caso, los procesos se agrupan y coordinan si están en el mismo nodo, por la correlación de los procesos en caso de fallos. Normalmente, en los sistemas multicore, los procesos de paso de mensaje se agrupan por nodo, en la medida de lo posible, de acuerdo

al patrón de comunicaciones por medio de la distribución o *mapping* de ejecución.

2.3. Middleware para dar soporte a la tolerancia a fallos

En los últimos anos, se han propuesto varias estrategias de tolerancia a fallos para las aplicaciones paralelas científicas de paso de mensaje, que, de acuerdo a Gropp [14] se pueden categorizar en tres grupos según sea la capa en la cual el protocolo de *rollback-recovery* es incorporado.

Aplicación: En este caso los mecanismos de tolerancia a fallos forman parte de la lógica del programa. Por ejemplo, la propuesta de Gropp [14], la de Rao [26], la herramienta propuesta por Rodríguez [57] para realizar checkpoints portables, o las técnicas descritas por Florio en [58], son ejemplos que entran en esta clasificación. Estas estrategias se basan en modelar o bien facilitar la tarea de diseno y desarrollo de aplicaciones tolerantes a fallos ya sea definiendo patrones de programación o anadiendo llamadas a nuevas librerías. Estas soluciones tienen la capacidad de ser las más eficientes debido al grado de conocimiento que tienen de la lógica y de las estructuras de datos de la aplicación. Sin embargo, se necesita disponer de los programas fuentes y además tienen un alto coste en desarrollo y pruebas, y la complejidad de la tolerancia a fallos se anade como un problema más a resolver por los programadores de aplicaciones.

Starfish [59] opera a este nivel para ofrecer su funcionalidad completa, aunque también puede utilizarse en forma transparente sin modificar la aplicación, ubicándose entonces en la capa siguiente de librería de paso de mensajes. Permite realizar cambios dinámicos en las aplicaciones paralelas para que no finalicen su ejecución en caso de caída de un nodo, pero no posee un manejo automático de reconexión de los clientes en caso de fallos de nodos ni de reinicio de los procesos a partir del checkpoint. Usa checkpoint coordinado y no coordinado pero no dispone de protocolo de log de mensajes. FT-MPI [5] está disenado para que dar a la aplicación control sobre como tolerar los fallos, tomando decisiones acerca de qué realizar con el proceso que ha caído o qué tipo de protocolo utilizar. Sin embargo, tiene una modalidad por defecto que permite funcionar en modo transparente para la aplicación.

Librería de Paso de Mensaje: Los algoritmos de tolerancia a fallos se anaden en la capa de la librería de paso de mensaje, ofreciendo en muchos casos una solución transparente a la aplicación. La mayoría de las propuestas en la literatura se integran en

implementaciones de Message Passing Interface (MPI) [2], que es el estándar más utilizado en la programación paralela de paso de mensajes. En estos casos podemos ver la tolerancia a fallos como una funcionalidad extendida de cada implementación. El desarrollo de las aplicaciones de altas prestaciones, hace que aumente la variedad de librerías MPI. Si bien todas responden a las versiones del estándar MPI-1, MPI-2 o, en breve, a MPI-3 [60], ofrecen diferencias en prestaciones y funcionalidades, como soporte a distintos tipos de redes y protocolos de tolerancia a fallos. Cuando la función de protección ante fallos está acoplada a la librería de paso de mensajes, encontramos las limitaciones de que si el usuario quiere cambiar de librería, probablemente ya no disponga de los mismos protocolos de tolerancia a fallos, y además, los cambios de versiones de las librerias, generalmente impactan en la tolerancia a fallos requiriendo desarrollo y test. Ejemplo de estos casos son MPICH-V [4] FT-MPI [5]. RADIC para Open-MPI [19], Open-MPI [7], MPI-FT [8], MPI/FT TM [9] [10], Egida [11]. LA-MPI [12], LAM-MPI [13] proponen tolerancia a fallos en la librería, pero ambos proyectos han convergido con el de Open-MPI. Si bien MPI es el lenguaje más utilizado para aplicaciones paralelas de paso de mensaje, y nuestro estudio se ha centrado en esta área, tambien podemos mencionar el trabajo de Chakravorty [32] quién propone un protocolo de tolerancia a fallos transparente para sistemas paralelos masivos basado en el lenguaje de objetos Charm++ [33].

Sistema Operativo: Cuando la tolerancia a fallos se ubica en esta capa, ofrece una solución transparente para la aplicación y para la librería. Permite que se aplique una política de tolerancia a fallos a nivel de sistema de la que pueden beneficiarse todas las aplicaciones paralelas que se ejecuten en el mismo, independientemente de la librería de paso de mensajes y versión que estuvieran utilizando. Las políticas de tolerancia a fallos que se pueden aplicar a este nivel podrían determinar, por ejemplo, cuando tolerar fallos y qué protocolo se adecúa más a la ejecución de la aplicación en dicho sistema, teniendo en cuenta la criticidad de finalización. Los inconvenientes de este tipo de solución es que generalmente son más complejas de desarrollar e integrar con el resto de componentes y que son sensibles a cambios en el sistema operativo, como por ejemplo la versión de kernel. La propuesta de este trabajo pertenece a esta categoría. A continuación, explicamos otros proyectos que ofrecen funciones de tolerancia a fallos en esta capa.

Como se ha visto en la sección anterior, Berkeley Lab Checkpoint/Restart (BLCR) [44] es una conocida herramienta que se utiliza para hacer *checkpoint* y *restart* de procesos a nivel de kernel. Puede usarse en programas *multithreading*, y en caso de aplicaciones distribuidas y paralelas el usuario es responsable de cerrar y abrir los sockets de comunicaciones.

DMTCP [42] (*Distributed Multithreaded Checkpointing*) es una herramienta de check-

point y restart a nivel de usuario para ser utilizada con aplicaciones distribuidas y paralelas. El estado global de la aplicación es guardado incluyendo sus sockets de comunicaciones. El protocolo utilizado es del tipo checkpoint coordinado, y no incluye log de mensajes, por lo cual no es posible la recuperación de un proceso independiente hasta el momento del fallo, y fuerza a todos los procesos a hacer rollback utilizado el último checkpoint. Esto es un inconveniente para el caso de aplicaciones con un número alto de procesos y para las débilmente acopladas.

DejaVu [61] es un sistema de tolerancia a fallos que permite la migración y la recuperación de aplicaciones paralelas y distribuidas. Provee los requerimientos de protección, detección y recuperación de una solución de tolerancia a fallos e implementa un mecanismo para guardar un estado global. Aunque implementa un protocolo de checkpoint no coordinado, lo hace mediante un algoritmo de coordinación para asegurar el estado consistente. Esto es un inconveniente para escalar a un mayor número de procesos.

Kerrighed [62] implementa un mecanismo jerárquico de checkpoint coordinado para aplicaciones de paso de mensajes desarrollado a nivel de sistema operativo para tener independencia de la librería.

2.3.1. Arquitecturas de Tolerancia a Fallos

Las arquitecturas de tolerancia a fallos para aplicaciones de paso de mensaje que se basan en estrategias de *rollback-recovery*, proveen las funcionalidades de protección, detección, recuperación y reconfiguración. La protección consiste en salvar el estado de los procesos y/o los mensajes de acuerdo al tipo de protocolo que se esté usando. La detección, corresponde a determinar si ha ocurrido un fallo, en cuyo caso, se da lugar la recuperación de los procesos afectados y la reconfiguración del sistema para aislar el componente que ha fallado y enmascarar el error.

Algunos proyectos son incompletos con respecto a estos requisitos, en general la detección de fallos se deja a cargo del usuario. También, la recuperación del estado de ejecución no suele ofrecerse en forma automática, dejando que la aplicación o el operador reanude la ejecución interrumpida y reconfigure al nuevo sistema aislando al nodo fallido hasta su reparación. Así, por ejemplo, en esta fase manual se indica dónde se recuperan los procesos y a dónde se han de conectar los procesos clientes. Consideramos que la recuperación y la reconfiguración manual no colaboran a aumentar la disponibilidad del sistema. Nuestra propuesta, en cambio, incluye un sistema de detección y un sistema de reconfiguración y recuperación automático.

La tabla 2.1 muestra las características que ofrecen las principales arquitecturas de tolerancia a fallos que se encuentran en la literatura. Se comparan las siguientes

características:

	Open -MPI	MPICH	MPI -FT	FT MPI	Star fish	Egida	RADIC OMPI	RADIC Sistema
Detección	No	No	Sí	Sí	Sí	Sí	Sí	Sí
Rec.M/A	Manual	Manual	Manual	Manual	Manual	Manual	Autom	Autom
Tr-Ap	Sí	Sí	Sí	Semi	Sí	Sí	Sí	Sí
Tr-Lib	No	No	No	No	Sí	No	No	Sí
Central	File	Chk	Sí	-	No	Sí	No	No
	Mgr	Server	Sí	-	No	Sí	No	No
Compilar	Sí	Sí	Sí	Sí	Sí	Sí	Sí	No
Chk-Coord	Sí	Sí	Sí	No	Sí	Sí	No	No
Log-Msg	Sí	Sí	Sí	No	Ext	Ext	Sí	Sí

Cuadro 2.1: Tabla comparativa de arquitecturas de Tolerancia a Fallos

1. **Detección:** Si la arquitectura detecta o no los tipos de fallos que protege.
2. **Recuperación Manual o Automática:** Si la recuperación y reconfiguración se realiza en forma automática o ha de ser provista por el usuario o programador de la aplicación.
3. **Transparente a la Aplicación:** Se indica si para utilizar la arquitectura es necesario o no modificar la aplicación.
4. **Transparente a la Librería:** Se indica si para utilizar la arquitectura es necesario o no modificar la librería de paso de mensaje MPI.
5. **Centralizado:** Se indica si la arquitectura incluye algún componente centralizado. Este análisis se realiza porque los elementos centralizados pueden afectar a la escalabilidad de la aplicación.
6. **Compilar:** Se indica si para utilizar la solución es necesario disponer de los fuentes de la aplicación para compilar y/o enlazar. Creemos importante que un servicio en la capa de infraestructura orientado no los necesite.
7. **Chk.Coordinado:** Se indica si ofrece el protocolo de checkpoint coordinado.
8. **Log.Mensajes:** Se indica si ofrece algún protocolo de log de mensajes ya sea pesimista, optimista o causal o si se puede extender por parte del usuario.

2.4. Arquitectura RADIC

RADIC [6] [21] [20] es una arquitectura de tolerancia a fallos para aplicaciones de paso de mensajes que define los modelos requeridos de protección de estado de ejecución, detección de fallos de nodos y recuperación de estado de ejecución. Tolera fallos de nodos y utiliza la redundancia del computador paralelo para recuperar procesos paralelos que estuvieren ejecutando en un nodo que cae. En la configuración básica, sólo tolera un fallo a la vez. Es decir, se asume que un fallo ocurre luego de que el sistema se haya recuperado de un fallo anterior.

Define dos componentes de software los cuales trabajan en un modo colaborativo para conseguir la tolerancia a fallos:

- **Observador (O_i):** Hay uno por cada uno de los procesos paralelos (P_i) de la aplicación. Monitoriza el estado de la aplicación, realiza los checkpoints no coordinados y el log de los mensajes recibidos. Esta información, llamada datos críticos, es enviada a otro nodo que los almacena, considerado almacenamiento estable por la independencia de los fallos entre ambos. Enmascara los errores generados por los fallos de comunicación y colabora en la detección de errores, avisando al protector en caso de tener un error de comunicación.

- **Protector (T_i):** Hay un protector en cada uno de los nodos de procesamiento (N_i). Guarda los datos críticos enviados por los observadores. Es responsable de la recuperación de los procesos utilizando el último checkpoint recibido. Utiliza un protocolo de heartbeat / watchdog para la detección de fallos con sus nodos vecinos.

La figura 2.3 muestra una aplicación paralela en ejecución en un cluster utilizando RADIC. Las flechas en diagonal representan el flujo de datos críticos y las horizontales, los *heartbeats* enviados entre los protectores.

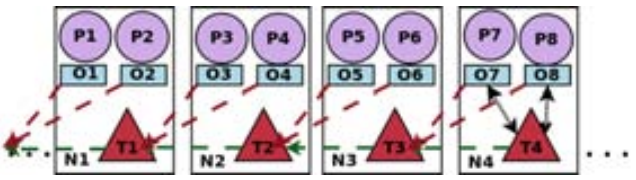


Figura 2.3: Diagrama de RADIC - Cada observador O_i envia los datos críticos a su protector T_{i-1}. Cada protector T_i envía senales de *heartbeat* al protector T_{i-1}

RADIC proporciona los modelos de protección, recuperación y detección los cuales serán explicados en los siguientes capítulos. Las actividades relacionadas al protocolo de

rollback recovery utilizado de log de mensaje pesimista basado en receptor, así como el resto de tareas realizadas, son llevadas a cabo de modo distribuido, sin incorporar ningún elemento centralizado para mantener la escalabilidad de la aplicación que se protege. Estas propiedades son esenciales en las arquitecturas actuales donde las aplicaciones y el número de procesadores se mantienen en constante aumento. Es por este motivo que se han seguido los modelos de RADIC en el diseno de la solución del sistema de tolerancia a fallos que se presenta en este trabajo.

Capítulo 3

Tolerancia a fallos en la capa de socket basada en RADIC

3.1. Introducción

Socket [16] es una interfaz de aplicación *Application Program Interface (API)* estándar *de facto* de los sistemas *POSIX Portable Operating System Interface* para realizar el transporte de comunicaciones entre dos procesos utilizando protocolos del tipo *Transmission Control Protocol (TCP)* o *User Datagram Protocol (UDP)* [17]. Los sockets constituyen una interfaz básica para las aplicaciones que funcionan en red. Cuando una de las partes de la comunicación tiene un fallo, la funciones de *socket* devuelven un error.

Nuestra propuesta consiste en cambiar el comportamiento de la interfaz de *socket* de modo que se detecte el fallo y que cuando éste se produzca, se reconfigure la comunicación utilizando una nueva dirección IP (*Internet Protocol*), a donde se migra al proceso que ha fallado, sin interrumpir la ejecución de la aplicación. Utilizando esta nueva dirección IP, se restablece la comunicación con el proceso remoto ya recuperado. Si además, dentro de la API de *socket*, incluimos las tareas de protección y recuperación de procesos manteniendo la interfaz estándar, obtenemos una solución transparente y automática de tolerancia a fallos para aplicaciones y librerías de paso de mensajes que utilicen *socket*.

La idea de establecer conexiones seguras modificando las funciones del socket fue propuesta por Zandy [63], pero en dicho caso no se utiliza en conjunto con un sistema de tolerancia a fallos que garantice que no hay pérdida de datos, sino como un sistema de localización de servidores en forma automática y transparente en caso se fallos.

En la primera parte de este capítulo, se presenta el modelo propuesto de *socket seguro* que permite enmascarar los fallos de comunicaciones a aplicaciones y librerías basadas en *socket* en forma transparente. La segunda parte desarrolla la propuesta describiendo

los modelos funcionales de protección, detección, recuperación y reconfiguración. En la tercera parte se analiza la sobrecarga del protocolo de log de mensajes durante las fases de protección y recuperación y, por último, se realiza la validación experimental y explican las conclusiones.

3.2. Socket Seguro

3.2.1. ¿Qué es un Socket?

Un socket permite establecer una conexión y un intercambio de datos mediante diferentes protocolos de transporte como TCP. Son utilizados por las aplicaciones a través de un conjunto de operaciones *Application programm interface API* que constituyen un estándar *de facto* soportadas por librerías del sistema operativo [16].

La figura 3.1 muestra el flujo de operaciones que se necesita para establecer una conexión cliente/servidor y luego para comunicar datos mediante las funciones de socket *send* y *recv*. Los parámetros utilizados durante la creación del socket permite establecer las características del protocolo que se va a utilizar para el intercambio de datos.

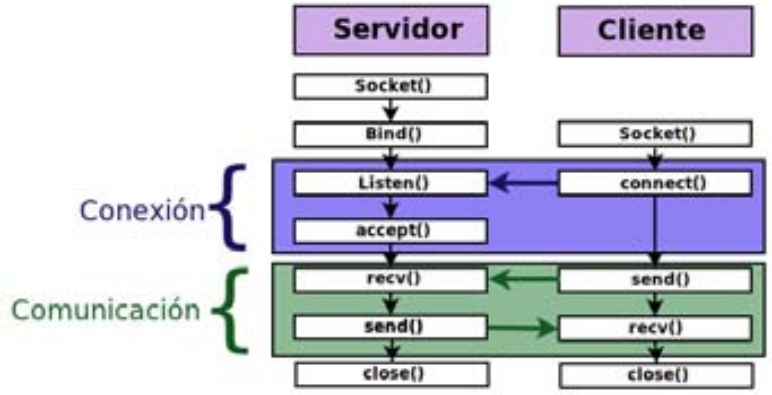


Figura 3.1: Socket API básica

Un socket es utilizado por dos partes, cliente y servidor, dependiendo del tipo de operación que se haya realizado durante el establecimiento de la conexión. Sin embargo, durante la comunicación, ambas partes se comportan de igual manera, enviando y recibiendo datos, por lo que utilizaremos el término **local** para referirnos al ámbito de un nodo determinado y **remoto** para lo que sucede en uno diferente.

Cuando el sitio **remoto** de una conexión tiene un fallo durante la comunicación, la operación de *send* o *recv* del sitio **local** devuelve un error. A su vez, si la aplicación que utiliza dicho socket no tiene previsto una alternativa ante dicha situación, probablemente

cancele. En el caso particular del estándar MPI, el comportamiento establecido es el de *fail-stop* [3].

El modelo de socket es utilizado por los protocolos de transporte como, por ejemplo TCP, que garantizan fiabilidad en el sentido de que cada paquete enviado es recibido respetando el orden, pero no proveen un mecanismo para recuperar la conexión cuando hay un fallo permanente en una de sus partes, debido a que esta clase de situación se ha dejado fuera del alcance del transporte de datos entre los dos procesos.

3.2.2. ¿Qué es un *socket seguro*?

Un *socket seguro* es un socket que tolera fallos. Se cambia el comportamiento habitual de las operaciones de comunicaciones de socket. Cuando el socket detecta que el sitio remoto tiene un problema, mediante los errores que recibe durante una operación de comunicaciones de envío o recepción de datos, en lugar de devolver el error a la aplicación, se realiza una función de diagnóstico y recuperación representada en la figura 3.2.

Figura 3.2: Diagnóstico y recuperación del *socket seguro*

El procedimiento consiste en:

- Se detecta un error durante el envío o recepción de datos.
- Se intenta restablecer la conexión, repitiendo la operación utilizada al inicio de la comunicación, *accept o connect*.
- Si eso no tuviera éxito, se obtiene, mediante una función provista por el sistema de tolerancia a fallos, la dirección en la que se recupera el proceso remoto.

- Se restablece la conexión utilizando esta nueva dirección y se realiza la comunicación de envío o recepción de datos fallida en donde se detectó el fallo.

- Si no se ha podido restablecer la conexión, se repite el procedimiento, reintentando reconectar un número configurable de veces. Si falla, se cancela la ejecución por no poder garantizar la conexión segura.

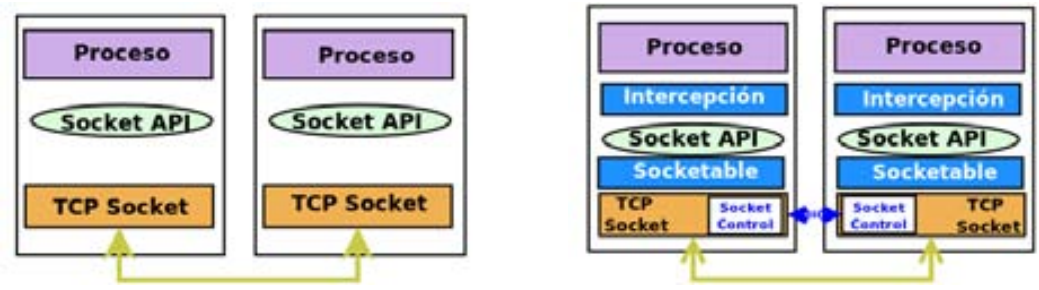
El *socket seguro* sólo puede conectarse con otro *socket seguro* y asume que no hay pérdida de datos por parte del proceso remoto cuando se migra de nodo.

3.2.3. ¿Cómo se consigue un *socket seguro*?

Dado que nuestro objetivo es que el sistema de tolerancia a fallos funcione sin realizar cambios en la aplicación así como tampoco en la librería de paso de mensaje, el *socket seguro* que se propone mantiene la misma interfaz API que la del socket. De este modo, las capas superiores utilizan al socket seguro en forma transparente.

Los sistemas operativos POSIX disponen de un mecanismo denominado interposición de funciones por el cuál las llamadas a operaciones de librerías son atendidas en primer lugar por otra librería, quién, a su vez, invoca a la función original como parte de su lógica. De esta manera se realizan, por ejemplo, tareas de trazado o monitorización. El socket seguro opera bajo la forma de una librería dinámica que intercepta las llamadas a operaciones de la API de socket que se realicen durante la ejecución de la aplicación paralela.

La figura 3.3(a) representa una conexión realizada entre dos procesos mediante un socket basado en protocolo TPC. En la figura 3.3(b) muestra la misma conexión utilizando un *socket seguro*, que anade una intercepción para controlar los errores en comunicaciones y un socket de control.



(a) Conexión por socket (b) Conexión por socket seguro

Figura 3.3: Comparación de conexión por socket normal y por *socket seguro*

Cuando se establece una conexión por un socket, la librería del *socket seguro* guarda la información relativa al tipo de operación realizada, es decir, si es un *connect* o un *accept*, los datos de la dirección remota y el identificador del socket en una estructura de datos llamada **socketable**.

Durante la comunicación, se monitoriza el resultado de las operaciones de *send* y *recv*. Si se detecta un error del sitio remoto, se inicia la función de diagnóstico y recuperación que se ha explicado en el apartado anterior, representada en el diagrama 3.2, utilizando los datos de conexión guardados y cambiando la dirección IP en caso de un fallo remoto. Los sockets del tipo *accept* se realizan a partir de otro socket denominado *servidor o listen*, que es el socket que espera o escucha solicitudes de conexión de los clientes. Los datos de los sockets servidores creados por el proceso, también se guardan en la **socketable** porque necesitan ser recreados en la nueva dirección en caso de fallos.

3.2.4. Restableciendo la conexión

Cuando un socket servidor acepta la conexión de un cliente durante una reconexión, necesita determinar si el proceso remoto efectivamente es el que estaba conectado previamente y no otro cliente, ya que múltiples clientes solicitan una conexión con un servidor llamando a la misma dirección, determinada por una dirección IP *(Internet Protocol)* y puerto.

Para ello, durante la primera conexión, los dos procesos involucrados intercambian la siguiente información de identificación:

- Nodo: Número único del nodo donde se ejecuta el proceso. Los nodos son numerados al inicio en forma secuencial a partir de una tabla que contiene los nodos del clúster y es conocida por todos.
- Proceso: Número único que identifica al proceso (PID).
- Socket: Identificador del socket. El sistema operativo asigna un número al socket creado durante la primera conexión, conocido por la capa de aplicación y que se utiliza para el resto de operaciones. Se le denomina **socket virtual**.

Para el intercambio de este tipo de información se establece un nuevo socket denominado **control** entre las dos partes involucradas en una conexión. Esto permite que los mensajes de identificación se transmitan por este canal sin interferir en la comunicación de la aplicación. Hay un socket de control por cada socket virtual.

Cuando se restablece la conexión del socket, también se conecta un nuevo socket de control y se intercambian los datos de identificación del nodo, proceso y socket virtual. Con

dichos datos, el servidor puede determinar a cuál de las conexiones clientes corresponde comparando con los valores guardados en la socketable durante la primera conexión. Al final de la sección se incluye un ejemplo para facilitar la comprensión de este proceso.

El sistema operativo asigna nuevos identificadores a estos sockets creados. Por lo tanto, cuando se realizan reconexiones, el socket virtual conocido por la aplicación ya no es válido para operar con el proceso remoto, sino que ha de utilizarse este nuevo identificador al que lo denominamos **socket real**.

Cuando se interceptan las operaciones de socket de envío y recepción de datos, se cambia el identificador del socket virtual por el socket real antes de efectuar la comunicación, asegurando así que los datos llegan al proceso remoto sin error.

Por ejemplo, un proceso master que acepta conexiones de dos workers conectados por los sockets 4 y 6. Cada uno de estos utiliza los sockets 5 y 7 para su información de control respectivamente. La figura 3.4(a) muestra la tabla socketable del master en este momento. Si se pierde la conexión con uno de los worker por un fallo en el nodo N_i, se vuelve a realizar un *accept* del socket de la aplicación y el de control, identificados con 8 y 9 respectivamente. Se recibe la identificación por el socket de control 9. Si los datos de identificación corresponden al proceso remoto worker pid-w1 y su correspondiente socket virtual, entonces este nuevo par de socket serán los socket real y de control asociados al socket virtual 4. La figura 3.4(b) muestra los cambios en la tabla.

Socket	Remoto			Socket		Tipo	Datos
Virtual	Pid	Socket V.	Nodo	Real	Control	Socket	Conexión
4	pid-w1	3	N_i	4	5	Accept	ip-N_i /puerto
6	pid-w2	3	N_j	6	7	Accept	ip-N_j /puerto

(a) Socketable Master inicial

Socket	Remoto			Socket		Tipo	Datos
Virtual	Pid	Socket V.	Nodo	Real	Control	Socket	Conexión
4	pid-w1	3	N_z	8	9	Accept	ip-Nz /puerto
6	pid-w2	3	N_j	6	7	Accept	ip-N_j /puerto

(b) Socketable Master despues del fallo N_i

Figura 3.4: Tabla Socketable - Cambios realizados después de un fallo

3.2.5. Estructura de datos Socketable

Para cada uno de los socket creados por la aplicación, la librería mantiene los siguientes datos en la estructura socketable para disponer de un socket seguro:

- Socket virtual: Identificador del socket conocido por la aplicación.

- Socket real: Identificador del socket que está conectado con el proceso remoto.
- Socket control: Identificador del socket que está conectado con la librería del proceso remoto.
- PID remoto: Identificador del proceso remoto.
- Socket virtual remoto: Identificador del socket que se utiliza en la capa de aplicación el proceso remoto.
- Nodo remoto: Número de nodo del proceso remoto.
- Tipo de socket: Operación de conexión *accept o connect* o *listen* para socket servidores.
- Datos Conexión: Datos para realizar la conexión: IP, protocolo, puerto.

3.3. Tolerancia a Fallos usando sockets seguros

3.3.1. Funciones de Tolerancia a Fallos

Un sistema de tolerancia a fallos de aplicaciones de paso de mensaje que utilizan estrategias de *rollback-recovery*, han de proveer funcionalidades para la protección del estado de ejecución, detección de fallos, recuperación de procesos y reconfiguración del sistema para aislar al componente que ha fallado y enmascarar el error.

Los modelos de funcionamiento que hemos adoptados, se basan en la arquitectura de tolerancia a fallos RADIC [19] [20] [21], debido a sus características de distribución, descentralización, flexibilidad y escalabilidad. Las siguientes secciones explican como se han adaptado dichos modelos a la capa de socket.

3.3.2. Modelo de Protección

El modelo de protección define como se llevan a cabo las tareas relativas a salvar el estado de la aplicación para que pueda ser recuperado en caso de fallos. El componente de RADIC responsable de estas tareas es el **observador**, que se asocia a cada uno de los procesos paralelos para realizar el log de mensajes y los checkpoints de acuerdo al protocolo de *rollback-recovery* pesimista basado en receptor.

La librería del socket seguro presentada en la sección anterior se asocia a cada uno de los procesos y se integra con el rol del **observador**, por lo que la denominamos de esta manera a partir de aquí.

La figura 3.5 muestra la conexión entre cada observador O_i con su protector T_{i-1} usada para enviar la información crítica, compuesta por el log de los mensajes recibidos y los ficheros de checkpoint. A continuación se explica como el **observador** sigue un protocolo de protección de log de mensajes y checkpoint no coordinado mediante la interposición de funciones de socket.

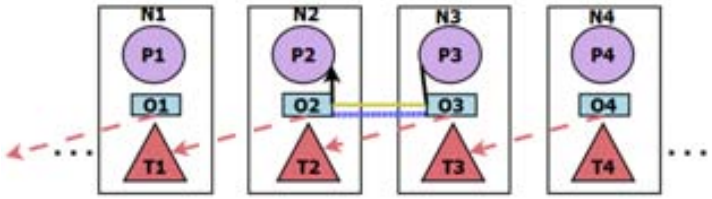


Figura 3.5: Modelo de Protección: socket real: línea continua amarilla - socket control: línea punteada azul - socket protector: línea discontinua roja

Cada componente distribuido de RADIC, observadores y protectores, gestiona una tabla denominada radictable, que tiene un registro para cada uno de los nodos del clúster con los siguientes campos.

- Nodo: Número único asignado al inicio en forma secuencial a partir de una tabla que contiene los nodos del clúster y es conocida por todos ellos.
- Nombre: Nombre del nodo.
- IP: Dirección del nodo.
- Protector: Identificador del nodo protector. Inicialmente se asigna al nodo anterior en la secuencia.

Log de Mensajes

El protocolo pesimista basando en receptor indica que los eventos no deterministas han de ser identificados y sus correspondientes determinantes se han de registrar en un almacenamiento estable [18].

La recepción de un mensaje es considerado un evento no determinista que ha de ser grabado. En RADIC, el almacenamiento estable es en otro nodo, el del protector, por lo que en principio, realizar el log de mensajes recibidos parece tan simple como interponer cada mensaje recibido por cada socket y enviar una copia al protector. Sin embargo, el protocolo pesimista asume que los fallos pueden ocurrir inmediatamente después de un evento no determinista, siendo pesimista porque en la realidad los fallos no son tan

frecuentes. Esta propiedad estipula que si un evento no esta registrado en almacenamiento estable, ningún proceso puede depender de él.

Por lo cual, el emisor del mensaje ha de esperar a que el mensaje se guarde en el protector antes de continuar con la operación. Una vez que el mensaje recibido se ha salvado en el protector, se envía una confirmación al emisor.

El envío de esta confirmación se realiza a través del socket de control utilizado por el *socket seguro* ya que es una comunicación del protocolo de protección entre los dos observadores involucrados en la conexión y no debe utilizarse el canal de la aplicación para no afectar la integridad de sus mensajes.

La figura 3.6 muestra como se interceptan las operaciones de envío y recepción, correspondientes a las operaciones *send y recv* respectivamente. Se siguen los siguientes pasos:

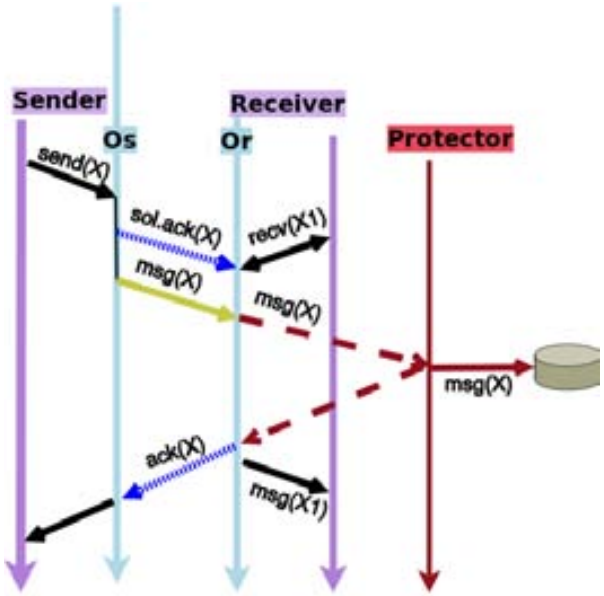


Figura 3.6: Log de Mensajes: Socket real: Línea continua amarilla - Socket control: Línea punteada azul - Socket protector: Línea discontinua roja

1. El observador del proceso *sender* O_s interpone la operación **send(X)**, siendo X la longitud del mensaje. O_s envía una solicitud numerada de confirmación **sol-ack(X)** a través del canal de control incluyendo la longitud del mensaje enviado.

2. El observador del proceso *receiver* O_r interpone la operación **recv(X1)**, siendo X1 la longitud del mensaje esperado. De acuerdo al estándar de la función *recv* de socket, cuando se solicita una cantidad de bytes superior a la cantidad recibida,

solo se entrega dicha cantidad. Por lo tanto, consideramos que X1 es menor o igual a X, ya que si fuera mayor, se entrega X.

3. O_r recibe sol-ack(X) por el canal de control y el mensaje de **X** bytes de longitud por el socket real. Se guarda el mensaje en una zona de memoria reservada para cada socket virtual.

4. El mensaje es enviado al protector y se espera la confirmación de que este es guardado.

5. O_r envía la confirmación a O_s a través del socket de control y finaliza la operación de *recv* devolviendo los primeros X1 bytes de los X recibidos.

6. O_s recibe la confirmación y finaliza la interposición de la operación de *send(X)*.

7. Si X1 es menor a X, las siguientes operaciones de *recv* que se intercepten hasta completar la cantidad de X, se responden copiando los siguientes bytes desde el buffer interno previamente grabado.

Checkpoint

El estado de cada proceso paralelo es guardado periódicamente en forma no coordinada. Una mayor frecuencia de checkpoint permite que el rollback que se realiza en caso de un fallo dure menos tiempo, y, en contrapartida se anade una sobrecarga en el tiempo de ejecución equivalente al tiempo que se necesita para hacer el checkpoint.

Durante el checkpoint todas las comunicaciones activas del proceso tienen que cerrarse. La librería BLCR que utilizamos recomienda este procedimiento por dos razones. La primera es para evitar pérdida de información en tránsito y, en segundo lugar, porque en caso de fallo, dichos sockets se han de restablecer luego de la reejecución del proceso en un nuevo nodo del clúster.

Por lo tanto, todos los sockets se cierran antes de un checkpoint y luego se restablecen. Esta operación se realiza aplicando el procedimiento para restablecer un fallo del *socket seguro* explicado previamente en 3.2.2, a todos los socket registrados en la socketable.

El evento de checkpoint puede iniciarse por:

- **Tiempo:** El protector local verifica en cada operación interceptada si se ha cumplido el intervalo de tiempo del checkpoint. Cuando esto ocurre, llama a una función de la librería de checkpoint/restart BLCR para hacer un salvado del estado de ejecución.

- **Evento:** El protector solicita al observador un evento de checkpoint al finalizar la reejecución de un proceso o cuando se están por completar las tablas internas de log de mensajes. Por otro lado, un usuario o administrador puede iniciar el checkpoint de un proceso mediante un utilitario que provee BLCR para realizar checkpoint/restart y esta llamada es reconocida por el observador y por la librería de BLCR. Sin embargo, nuestra propuesta automática recomienda que se utilicen las otras opciones para que el checkpoint sea reconocido por el protector en caso de fallos.

Cuando se realiza un checkpoint/restart, BLCR invoca a una función denominada *callback* provista por el observador.

La figura 3.7 representa el diagrama de flujo de la función de *callback*.

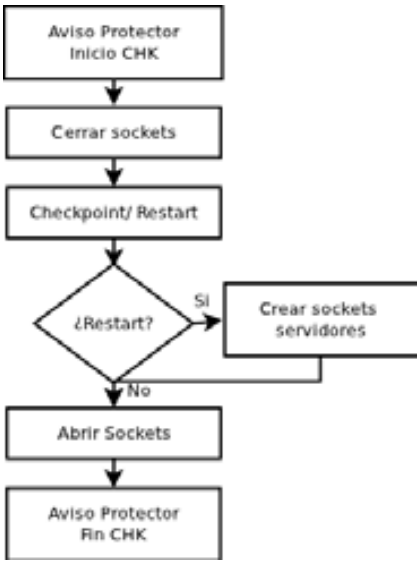


Figura 3.7: Checkpoint/Restart: Función de callback

3.3.3. Modelo de Detección

La detección de fallos de nodos es realizada por los protectores mediante envío y recepción de senales de *heartbeat*. Cada uno de los nodos del clúster tiene un protector. Los nodos son numerados al inicio secuencialmente 1 a N, siendo N el total de nodos que intervienen en la ejecución. Cada protector T_i, con i entre 1 y N, es responsable de detectar los fallos del nodo sucesor T_{i+1}, así como la de guardar los datos críticos de los procesos paralelos que este ejecuta. Cuando el protector T_{i+1} no responde a la senal de *heartbeat*, se espera confirmación del fallo del protector T_{i+2} que contacta con T_i al no recibir tampoco

respuesta de T_{i+1}, y se inicia la recuperación. La figura 3.8 muestra como el observador O_4 y protector T_4 contactan con T_2 al detectar el fallo de N_3

Los observadores participan en la detección de fallos cuando un *socket seguro* recibe un error en la comunicación con el proceso remoto y se conectan con el protector de dicho proceso. La localización del protector es determinada por la tabla radictable. El observador conoce el identificador del nodo remoto porque es un dato intercambiado durante el inicio de la conexión y es guardado en socketable. A partir de este dato, recupera la dirección del nodo protector de radictable y le pide un estado de diagnóstico.

Hay tres posibles respuestas que el protector contesta al pedido de diagnóstico:

1. El estado del nodo es correcto. En este caso, el observador sigue intentando conectar con el proceso remoto durante un número configurable de veces. Si continua la misma situación de no poder conectarse y el protector no detecta problemas, se finaliza la ejecución asumiendo que es un problema del nodo local con el resto del sistema.

2. El proceso está haciendo checkpoint. El observador espera un tiempo configurable, por defecto 1 segundo, y sigue reintentando hasta que el proceso finalice el checkpoint.

3. El proceso se ha recuperado en otro nodo. El protector envía la nueva dirección del proceso. El observador cambia su radictable anulando al nodo que ha fallado o bien cambiando su dirección si se ha utilizado un nodo de repuesto. Se cambia la dirección del socket remoto y vuelve a intentar la conexión con el proceso remoto, siguiendo el procedimiento de diagnóstico del socket seguro 3.2.

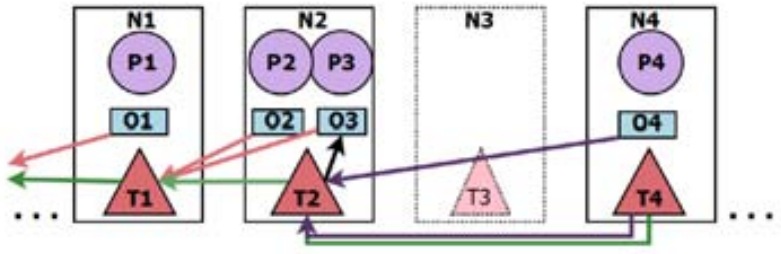


Figura 3.8: Modelo de Detección y Recuperación

3.3.4. Modelo de Recuperacion

Cuando un nodo falla, el protector que se ejecuta en el nodo anterior, detecta y confirma dicho fallo al recibir el pedido de diagnóstico de protector siguiente al del fallo. Si hubiera

un nodo de repuesto disponible, el protector le envía los datos críticos (checkpoints y log de mensajes) para que los procesos se recuperen en dicho nodo, si no existe, se utiliza el mismo nodo del protector, donde se reinician los procesos que han caído utilizando el último checkpoint. tal como se representa en la figura 3.8, donde el proceso P_3 es reiniciado en N_2.

Los procesos se recuperan utilizando la librería BLCR, quién invoca a la función de *callback* detallada en la figura 3.7. El proceso se ejecuta nuevamente a partir de ese punto. El observador detecta el estado de recuperación, y como primera acción, se vuelven a crear los sockets del tipo servidor del proceso que habían sido creados en el nodo anterior. Estos sockets son requeridos para escuchar las conexiones que luego serán aceptadas por el comando *accept.*

El observador se comporta de un modo diferente durante la reejecución del proceso, que se inicia con el último checkpoint y finaliza cuando se procesa el último de los mensajes guardados en el log. La figura 3.9 muestra el proceso de reejecución realizado por el observador para cada uno de los i socket virtuales del proceso paralelo.

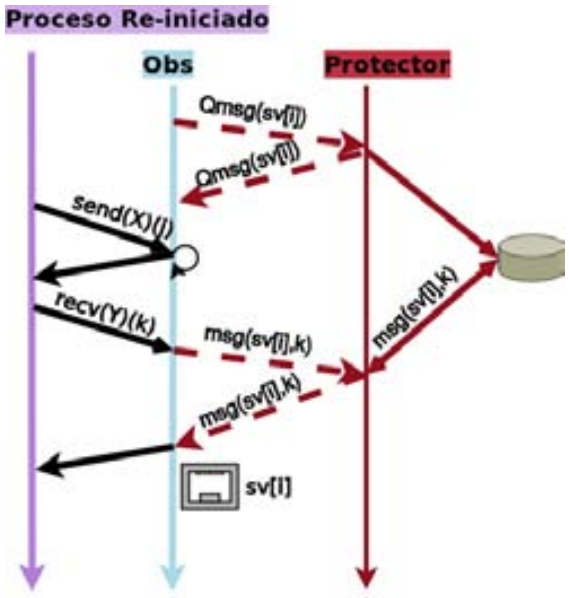


Figura 3.9: Reejecución de un proceso

1. El observador establece una conexión con el protector local y consulta cuantos mensajes hay guardados pendientes de reprocesar para el socket virtual i. El protector devuelve dicha cantidad **Qmsg(sv[i])**.

2. Durante la reejecución, las operaciones send(X) se interceptan y no se realizan nuevamente porque corresponden a reenvíos. $\boldsymbol{j}$ es un entero mayor o igual que

0 que representa a cada unos a las operaciones *send* ya realizadas por el socket virtual i antes del fallo. El diagrama representa sólo una de ellas. X es la longitud del mensaje.

3. Cada operación *recv(Y)(k)* se interpone y se pide al protector local. El mensaje se entrega en orden FIFO para cada uno de los socket virtuales. k es un número entero de 0 a **Qmsg(sv[i])**. Y es la longitud del mensaje.

4. Luego de interponer **Qmsg(sv[i])** funciones *recv*, se reconectan los sockets real y de control con el proceso remoto siguiendo el procedimiento detallado en la sección de socket seguro 3.2.

3.3.5. Modelo de Reconfiguración

Los observadores y protectores distribuidos en los N nodos del clúster cuando necesitan comunicarse con un proceso que ha cambiado de nodo, deben reconfigurar su sistema para comunicarse con el nuevo nodo. Esta tarea se realiza una vez confirmado el fallo por parte del protector de dicho nodo.

Se enumeran a continuación las diferentes situaciones y las tareas que realiza cada uno de los componentes para conseguir una correcta reconfiguración del sistema para aislar al nodo que ha fallado y enmascarar los errores del fallo.

- Al detectar un corte de comunicaciones en el nodo remoto, el observador reintenta abrir y restablecer la comunicación tal como se ha indicado en el procedimiento de *socket seguro*. Si el proceso remoto está haciendo checkpoint, luego de una serie de reintentos, se consigue establecer la conexión, aunque el sistema operativo asigna otro identificador al socket. Se guarda este identificador como socket real y se cambia por el virtual en cada una de las operaciones de la aplicación, enmascarando así el fallo de las comunicaciones.

- Los observadores y protectores cambian la radictable una vez confirmado el fallo por parte del protector. Si el nodo que ha fallado se ha remplazado por un nodo de repuesto, se cambia el nombre y la dirección IP correspondiente a dicho nodo. Si en cambio, los procesos han sido recuperados en el nodo del protector, se elimina el registro del nodo que ha caído. El campo protector del nodo siguiente se cambia por el identificador del nuevo protector que es el anterior al eliminado. De esta manera, la radictable representa la nueva secuencia de nodos y protectores luego del fallo que se utiliza para la detección de fallos por *heartbeat*.

- Los observadores de los procesos recuperados cambian las direcciones de los socket que estaban previamente conectados. Se utiliza la dirección que el protector indica cuando confirma el fallo, ya sea la del nodo de repuesto o bien su misma dirección. De este modo, se establece un nuevo socket real que es utilizado para transmitir la información que la aplicación solicite por el socket virtual.

3.4. Análisis de Sobrecargas

Esta sección analiza la sobrecarga del protocolo de *rollback-recovery* utilizado por RADIC, log de mensaje pesimista basando en receptor y checkpoint no coordinado. Se analiza la sobrecarga durante las fases de la protección y de la recuperación.

3.4.1. Log de mensaje pesimista basado en receptor en fase de protección

El uso de log de mensajes permite recuperar el estado de los procesos en el punto de fallo. La modalidad pesimista basado en receptor presenta más sobrecarga que otros como el optimista o causal durante la protección, pero, el proceso se recuperación es más simple y efectivo y no afecta a otros procesos, manteniendo el comportamiento distribuido [11] [18].

La figura 3.10 muestra el diagrama que representa como el observador interpone las operaciones de comunicación del socket de modo que se trate siguiendo un protocolo de este tipo, explicado previamente en 3.3.2. Cada paso anade una sobrecarga la cual se identifica anteponiendo un prefijo **Ts-** o **Tr-** dependiendo si la sobrecarga es relativa a un envío o recepción respectivamente.

A continuación, se explica y se realiza una discusión acerca de lo que ocasiona cada sobrecarga.

1. O_s interpone la operación **send(X)**, y envía una solicitud de confirmación **sol-ack(X)** a través del canal de control. La sobrecarga se denomina **Ts-ack-req.** El tiempo utilizado por el **Ts-msg** no es considerado sobrecarga ya que corresponde al tiempo de envío del mensaje del proceso. Los tiempos de envíos son los más cortos porque el sistema operativo devuelve el control al proceso en forma inmediata solapando el de la comunicación con la ejecución.

2. O_r interpone la operación **recv(X1)**, siendo X1 la longitud del mensaje esperado. Tal como se ha explicado en el capítulo anterior, consideramos que X1 es menor

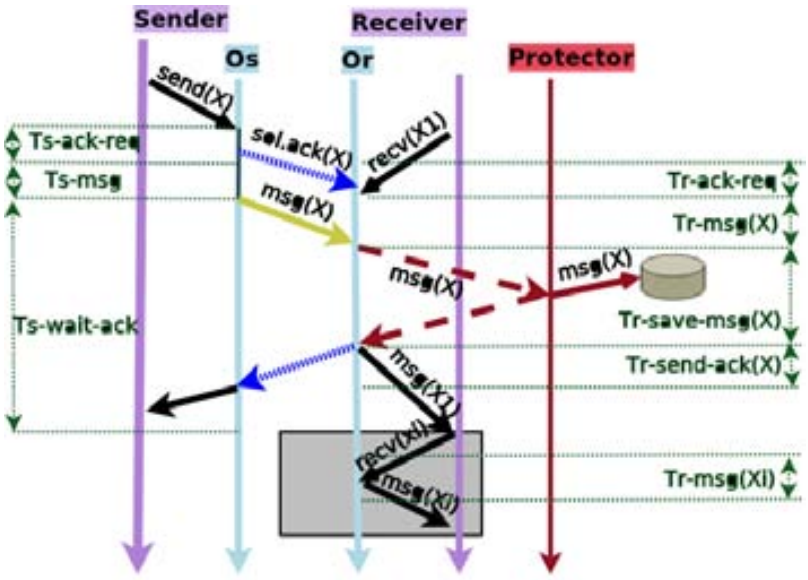


Figura 3.10: Log de mensaje pesimista basado en receptor en fase de Protección: Sockets Virtual/Real: Líneas continuas - Control: Líneas punteadas - Protector: Líneas discontinuas

o igual a X, ya que si fuera mayor, se entrega X.

3. O_r recibe sol-ack(X) por el canal de control en un tiempo **Tr-ack-req** y al mensaje de **X** bytes de longitud por el socket real en **Tr-msg(X)**. Este último no es una sobrecarga, sino que corresponde con la recepción del mensaje por parte del proceso. **Tr-ack-req** representa la espera del proceso receptor a los datos que necesita para continuar. Aumenta cuando el emisor demora este envío, y tiene una mayor dependencia con el patrón de la aplicación que con el sistema de protección.

4. O_r envía el mensaje al protector para guardarlo, y espera la confirmación en un tiempo **Tr-save-msg(X)**. Esta es la porción más considerable del total de sobrecarga del protocolo.

5. O_r envía la confirmación a O_s a través del socket de control anadiendo **Tr-send-ack**. y finaliza la operación de *recv* devolviendo los primeros X1 bytes de los X recibidos.

6. O_s recibe la confirmación y finaliza la interposición de la operación de *send(X)* en un tiempo **Ts-wait-ack**. Este tiempo está directamente relacionado con el tiempo que demoró el receptor en grabar el mensaje **Tr-save-msg(X)**.

7. Si X1 es menor a X, las siguientes operaciones de *recv* que se intercepten hasta

completar la cantidad de X, se responden copiando los siguientes bytes desde el buffer interno previamente grabado, considerando un tiempo **Tr-msg(Xi)**.

3.4.2. Log de mensaje pesimista basado en receptor en fase de recuperación

Cuando un nodo falla, el protector que se ejecuta en el nodo anterior, detecta, confirma dicho fallo y luego reinicia los procesos que se estaban ejecutando en ese nodo utilizando el último checkpoint de cada uno.

Una vez reiniciado el proceso paralelo, se empieza la reejecución desde el último checkpoint hasta que todos los mensajes previamente grabados sean utilizados por las operaciones de recepción de datos *recv*. Durante la reejecución, las operaciones de envío no se realizan porque dichos mensajes ya fueron enviados durante la primera ejecución. Sin embargo, cada envío tiene asociado una solicitud numerada de confirmación, por lo que los observadores receptores pueden detectar y descartar envíos duplicados. Esta funcionalidad es útil porque el proceso en recuperación puede realizar envíos de mensajes duplicados debido a que el procedimiento considera que la última operación antes del fallo es una recepción y esto puede no ser así, y que en la primera ejecución se hayan efectuados envíos de mensajes como últimas operaciones antes del fallo del nodo.

La figura 3.11 muestra el procedimiento de recuperación previamente explicado en el capítulo anterior en 3.3.4 ahora anadiendo las sobrecargas nombradas con el prefijo **Trcv-**. A continuación se explica y se discute sobre cada una de las sobrecargas de cada paso realizados para cada uno de los socket virtuales que tiene el proceso.

1. El observador consulta al protector local cuantos mensajes tiene guardados para reprocesar para el socket virtual *i*. El protector devuelve dicha cantidad **Qmsg(sv[i])**. El tiempo es **Trcv-qmsg**, correspondiente a una operación local, y es un tiempo despreciable.

2. Durante la reejecución, las operaciones send(X) se interceptan y no se realizan nuevamente porque corresponden a reenvíos. El tiempo **Trcv-send(X)(j)** es considerado despreciable ya que no se realiza operación alguna.

3. Cada operación *recv(Y)(k)* se interpone y se pide al protector local. El mensaje se entrega en orden FIFO para cada uno de los socket virtuales. k es un número entero de 0 a **Qmsg(sv[i])**. Se demora un tiempo **Trcv-recv(Y)(k)**. Corresponde a una operación de paso de mensaje realizada en un mismo nodo con una búsqueda en el disco de un mensaje de longitud Y.

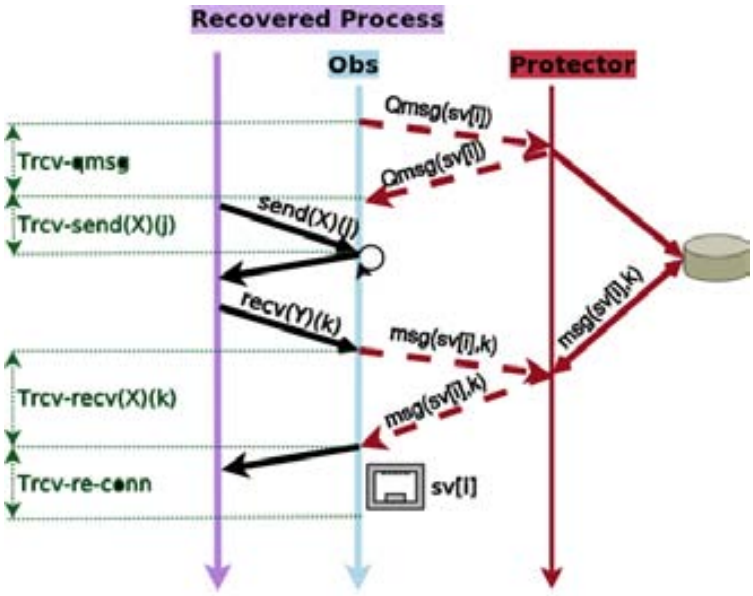


Figura 3.11: Receiver-based Pessimistic Protocol in Recovery Phase

4. Luego de interponer **Qmsg(sv[i])** funciones *recv*, se reconectan cada uno de los sockets real y de control de la socketable con el proceso remoto siguiendo el procedimiento detallado en la sección de *socket seguro*. La sobrecarga es de **Trcv-re-conn** y usualmente el proceso remoto se encuentra en fase de reintentos de conexión, por lo que esta tarea se realiza de inmediato.

A través de este detalle del procedimiento puede observarse la eficiencia del protocolo pesimista basado en receptor durante la recuperación, ya que no tiene dependencias con otros procesos, realiza operaciones locales y se evitan los reenvíos. La duración de la recuperación depende sobre todo de la cantidad de tiempo transcurrido desde el último checkpoint hasta el momento del fallo, decreciendo cuando dicho intervalo es menor.

3.5. Validación Experimental

Se ha desarrollado un prototipo de los módulos observador y de protector en capa de socket siguiendo las especificaciones de diseno definidas en este capítulo. De este modo, el observador intercepta las funciones de socket para mantener un nivel de fiabilidad ante fallos de nodos remotos y para seguir un protocolo de log de mensajes pesimista basado en receptor combinado con checkpoint no coordinado. El protector detecta fallos de sus nodos vecinos mediante senales de *heartbeat*, protege el estado de los procesos que se ejecutan en el nodo sucesor guardando su información crítica, atiende y contesta los pedidos de

diagnóstico, reinicia los procesos en caso de confirmar un fallo, y devuelve los mensajes guardados a un proceso en reejecución.

En esta sección se explica la evaluación experimental realizada con el objetivo principal de asegurar que el mecanismo propuesto para tolerar fallos mediante *socket seguro* y la aplicación de un protocolo de log de mensajes funcionan en esta capa. El segundo objetivo es conocer la sobrecarga del sistema durante la ejecución y la recuperación, para lo cual medimos los tiempos de ejecución y el ancho de banda de los modelos definidos y los comparamos con la ejecución de la aplicación sin protección.

Los experimentos fueron ejecutados en un cluster de 4 nodos con procesador *Intel*® Core™ *i5-650* con 6GB RAM, y red Gigabit *Ethernet*. El sistema operativo es Ubuntu 10.04 Kernel 2.6.32-33-server.

Los componentes observador y protector del prototipo se han desarrollado en C, compilados con GNU C versión *gcc (Ubuntu 4.4.3-4ubuntu5.1) 4.4.3*. El protector es un programa compilado con las opciones por defecto y linkado con la librería de BLCR (*-lcr*). El observador es una librería dinámica compilada con las cláusulas *-shared y -soname* y linkado con las librerias de BLCR (*-lcr*) y con la libreria de *dynamic link -ldl*).

3.5.1. Aplicaciones

Utilizamos dos aplicaciones que representan los patrones de aplicaciones y comunicaciones más utilizados en las aplicaciones paralelas en el ámbito científico. La primera es una aplicación de suma de matrices que sigue el modelo de *Master/Worker* y la otra de transferencia de calor del tipo *Single Program Multiple Data (SPMD)* ambas basadas en socket TCP.

La aplicación de suma de matrices Master/Worker es un proceso master que genera una matriz cuadrada entera y envía a cada worker las filas a sumar y recibe el resultado, el cual acumula. Se utiliza una sucesión entera para poder validar el resultado mediante una fórmula de cálculo. Se ha realizado la ejecución con una matriz de 1000x1500 y 4 workers, uno por cada uno de los nodos disponibles. La matriz hace 1000/4 ciclos, en cada uno de los cuales envía a cada worker 16k y luego espera las respuestas de 11 bytes cada una.

La aplicación de transferencia de calor SPMD, consiste en un proceso que genera dos matrices de números flotantes de 1000x1000 y luego distribuye la carga de trabajo entre los procesos. Se ejecuta con 4 procesos, uno por cada nodo. Dichas matrices se distribuyen entre los 4 procesos dividiendo 250 filas para cada uno. Cada uno de los procesos realiza un cálculo con sus datos durante 70000 iteraciones. Para cada uno de estos cálculos necesita conocer el resultado de las filas de borde que tiene el proceso siguiente y/o anterior, por lo que cada proceso envía y recibe dichas filas de borde. Al final, devuelve el resultado

del proceso al distribuidor, quien graba la imagen resultante. Cabe aclarar que en los experimentos realizados no se efectúan los intercambios de datos en todas las iteraciones, sino cuando la iteración es múltiplo de un parámetro adicional. En este caso es 100, por lo que se realizan 700 intercambios de bordes. Esta modificación se ha realizado para agilizar el proceso de desarrollo y pruebas, evitando una sobrecarga en las comunicaciones.

Estas aplicaciones nos permiten validar el comportamiento del socket seguro con dos topologías de comunicaciones diferentes. El master utiliza un socket servidor que acepta múltiples conexiones, los workers usan sockets clientes, mientras que los procesos de la aplicación SPMD utilizan a la vez dos sockets, uno del tipo servidor que acepta conexiones y otro del tipo cliente.

Se utiliza un proceso por nodo para probar la configuración básica de RADIC monocore. Si bien no se disponen de nodos de repuesto para utilizar en caso de fallos, al utilizar nodos con dos cores, no se espera degradación durante la ejecución después del fallo de dos procesos en un sólo nodo.

3.5.2. Metodología

Se han utilizado tres tipos de ejecuciones.

1. *No FT:* No se utiliza tolerancia a fallos, y se realiza para tener una referencia de la productividad de cada proceso y del tiempo de ejecución.

2. *FT Protección:* Se utiliza la tolerancia a fallos para proteger dicha ejecución mediante checkpoint y log de mensaje. Permite comprobar el funcionamiento de los mecanismos de protección de checkpoint y log, el procedimiento de diagnóstico del socket seguro cuando se detecta un corte de comunicación por el checkpoint y la correspondiente reconfiguración de los sockets si el sistema operativo cambia el identificador del socket real. Utilizamos este tipo de ejecución para medir la sobrecarga anadida por log y del checkpoint. En el caso del M/W, se realiza checkpoint cada 200 eventos y en el SPMD cada 600. Se considera un evento cada operación de socket interceptada por el observador.

3. *FT Recuperación:* Se utiliza la tolerancia a fallos para proteger la ejecución mediante checkpoint y log de mensaje y se inyecta un fallo en el nodo **N3** que provoca una caída del proceso P3 50 eventos después del primer checkpoint en el caso del M/W y 100 en el caso del SPMD. Este tipo de ejecución con fallos nos permite comprobar el funcionamiento del mecanismo de detección del sistema mediante heartbeat y watchdog, la recuperación del proceso por parte del

protector, la reejecución por parte del observador utilizando el log previamente grabado, la reconexión con el resto de procesos, y, por último, la reconfiguración de los protectores y observadores aislando el nodo que ha fallado y enmascarando las comunicaciones de la aplicación al proceso migrado.

Para realizar las gráficas se ha realizado un promedio de la actividad realizada de cada proceso durante un intervalo de tiempo medido en segundos. El eje X muestra los distintos intervalos y el eje Y, el trabajo realizado durante dicho intervalo. El trabajo de los procesos de SPMD se mide en iteraciones y la de los procesos Master y Worker se mide en filas.

Se realizaron al menos 10 ejecuciones de cada uno de los casos y se comprobó el funcionamiento esperado de protección, detección y recuperación. Se selecciona un caso para cada una de las aplicaciones con un valor medio de tiempo de ejecución y se muestra y explican los resultados obtenidos en el siguiente apartado.

3.5.3. Experimentos

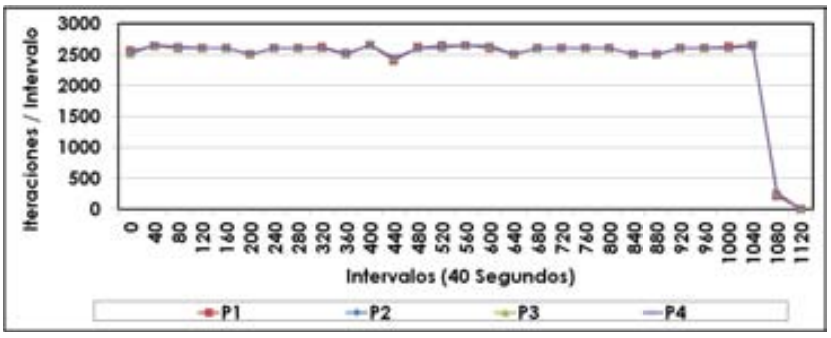
Aplicación SPMD de transferencia de calor

La figura 3.12 muestra el comportamiento de todos los procesos de la aplicación de transferencia en las ejecuciones realizadas con tolerancia a fallos.

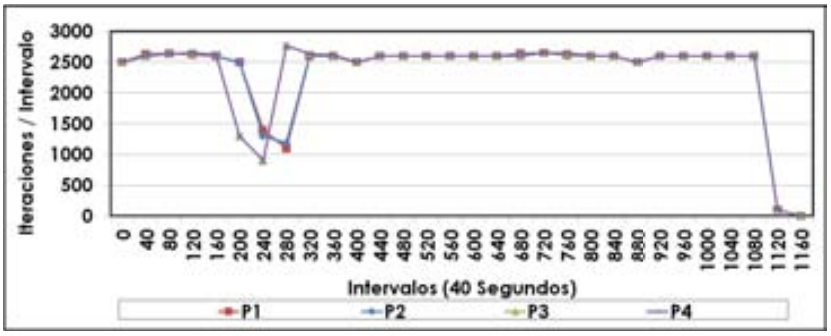
En la figura 3.12(a) se muestra la ejecución denominada *FT Protección*. Los procesos realizan checkpoint cerrando y abriendo los sockets. Este error de comunicaciones es detectado por el socket seguro de los procesos que se comunican con dicho proceso y se aplica el procedimiento de diagnóstico que permite recuperar la comunicación. Cada checkpoint ocupa 1400kb y las latencias son menores a 1 segundo.

La figure 3.12(b) muestra como en el momento del fallo de nodo, el rendimiento de todos los procesos caen por el acoplamiento de mensajes en este tipo de aplicaciones. Durante el tiempo de reejecución, en el cual se reprocesan 100 eventos con 50 mensajes de log, el resto de procesos aplica el procedimiento de diagnóstico hasta poder reconectar con el proceso en su nueva ubicación, el nodo N2. A partir de ese punto, la ejecución continua y finaliza correctamente. Como se utilizan nodos multicores, no hay degradación de rendimiento a pesar de que los procesos P2 y P3 se ejecutan en el mismo nodo N2 luego de la recuperación.

La figura 3.13 muestra el comportamiento del proceso P3 durante los tres tipos de ejecuciones. La diferencia de altura en la representación de No FT con *FT-Protección* es causado por la disminución del rendimiento al anadir checkpoints y log de los mensajes. En la representación de *FT-Recuperación* se ve la caída de rendimiento durante el fallo, y luego la subida durante la reejecución.



(a) Ejecucion FT-Proteccion



(b) Ejecucion FT-Recuperacion

Figura 3.12: Productividad de aplicación SPMD en Protección y Recuperación

El aumento en tiempo de ejecución fue de 3.24 % durante la protección y de 6.76 % en presencia de un fallo.

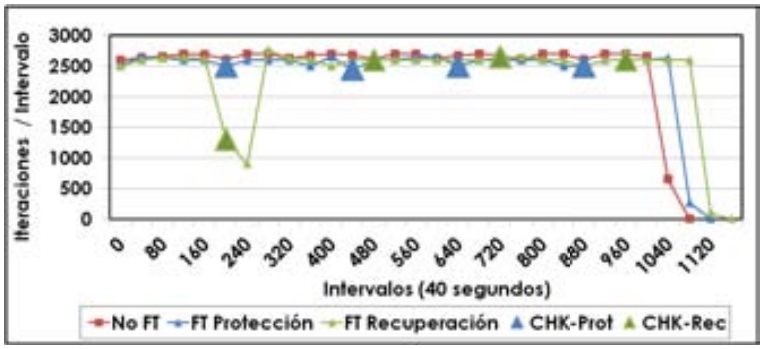


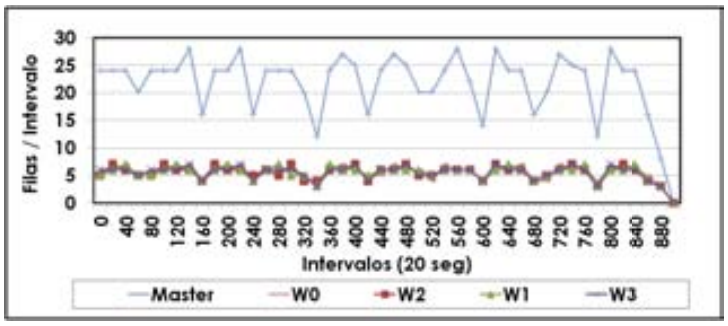
Figura 3.13: Comparación de ejecuciones proceso P3

Aplicacion Master / Worker

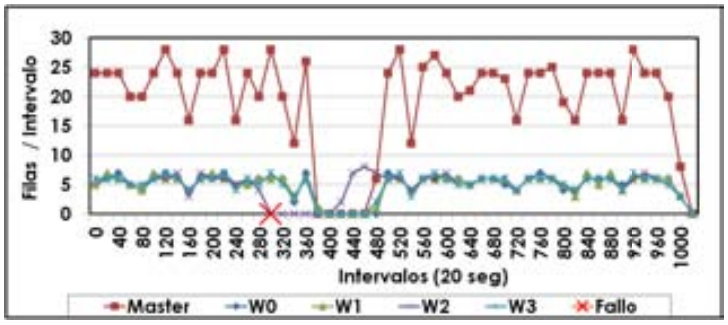
La figura 3.14 muestra el comportamiento del master y los workers de la aplicación de suma de matrices, durante la ejecución *FT Proteccion* sin fallos en la figura 3.14(a), y en recuperación, en la figura 3.14(b), muestra la caída de productividad cuando el worker cae

y se recupera en el nodo 2.

Los descensos en las filas por intervalo que se ven en el master, se deben a los 10 checkpoints que efectúa y a los dos checkpoints de cada worker. En recuperación, hay un lapso sin actividad por parte de toda la aplicación, correspondiente al tiempo de detección de fallo por parte de los protectores, reinicio y reejecución del worker desde el último checkpoint hasta el punto de fallo, y reconexión con el master. Luego, continua la ejecución y termina correctamente.



(a) Ejecucion FT-Proteccion

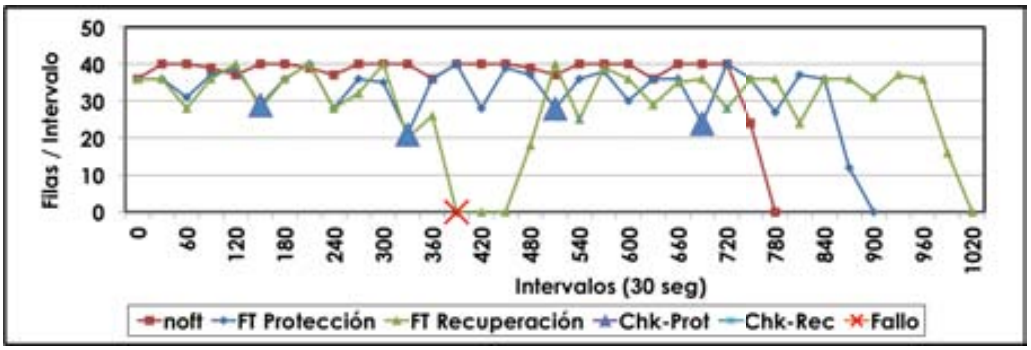


(b) Ejecucion FT-Recuperacion

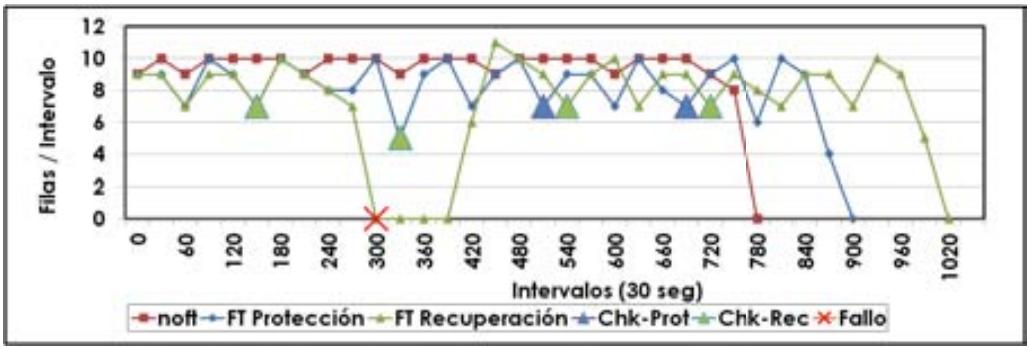
Figura 3.14: Productividad de aplicación Master/Worker en Protección y Recuperación

La figura 3.15 muestra el rendimiento de los procesos master 3.15(a) y del worker W3 3.15(b) durante los tres tipos de ejecuciones. Al igual que en el caso anterior, la diferencia de altura muestra como la tolerancia a fallos sobrecarga a la aplicación disminuyendo la productividad. También se observa una sobre-productividad del worker durante la reejecución. Las filas por segundo realizadas por del master disminuyen abruptamente segundos después de la caída del worker, debido a que se queda detenido su procesamiento hasta que logra reconectar con el worker recuperado.

El aumento en tiempo de ejecución es de 15.49 % durante la protección y de 31.25 % en recuperación. El intervalo de checkpoint pequeño fuerza la realización de múltiples checkpoint que impactan en la sobrecarga total del tiempo de ejecución.



(a) Proceso Master

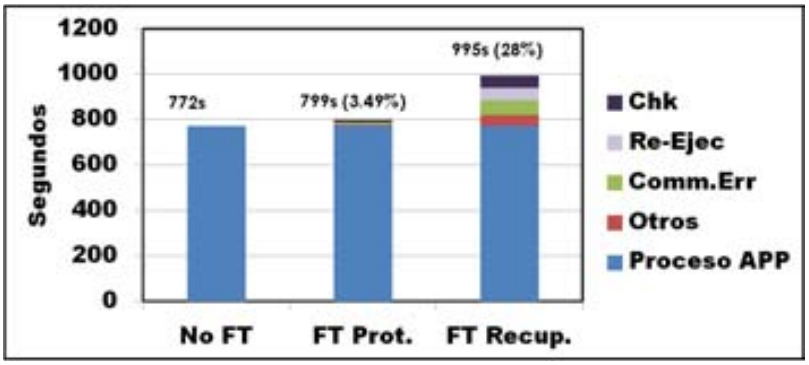


(b) Proceso Worker

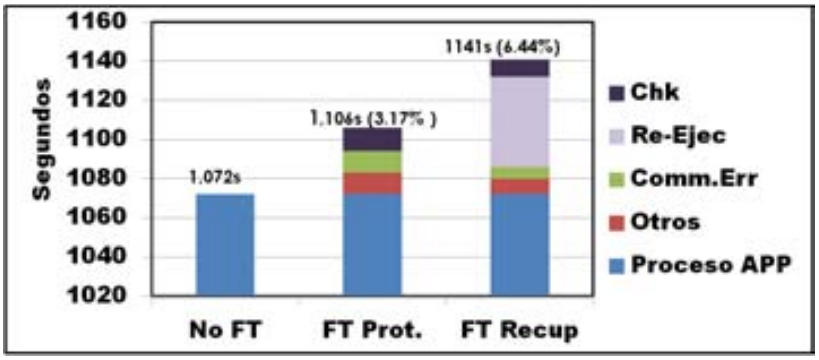
Figura 3.15: Comparación de ejecuciones de los procesos Master y Worker W3

Analisis de sobrecargas en tiempo y ancho de banda

En las figuras 3.16(a) y 3.16(b) se representan los diferentes tiempos de sobrecarga de ambas aplicaciones, comparando los tres modos de ejecución. Dichos tiempos fueron medidos en el proceso P3 ejecutando en el nodo **N3.** El tiempo se divide en: los segundos de interrupción y restablecimiento de las comunicaciones al realizar checkpoint y restart, el tiempo de reinicio y reejecución, tiempo de reconfiguración para recuperarse de fallos remotos o checkpoints, y otros tiempos no considerados, como la detección de errores, y por último, el tiempo de referencia de procesamiento de la aplicación sin tolerancia **(No FT).**



(a) M/W

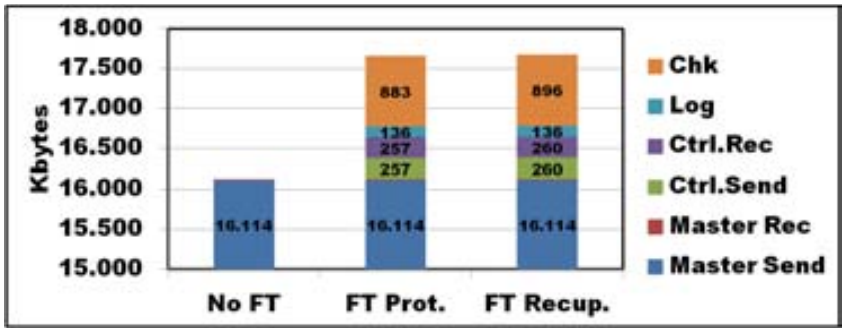


(b) SPMD

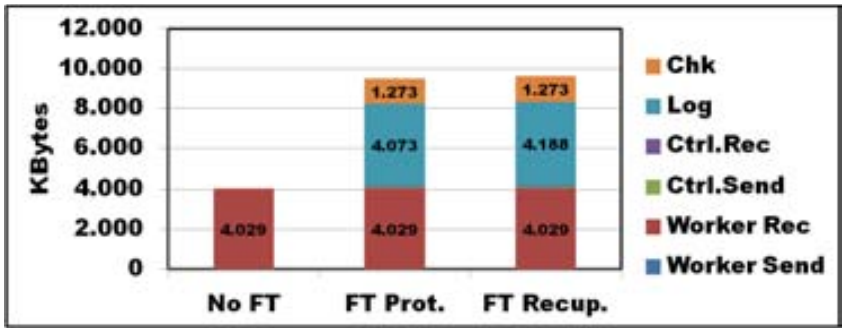
Figura 3.16: Tiempos de sobrecarga

El gráfico 3.17(a) y 3.17(b) muestra la cantidad de bytes enviados y recibidos por el proceso master y worker respectivamente en los tres modos de ejecuciones. Cabe aclarar que el experimento del M/W de este experimento se ha realizado con una matriz de 1000x1000 y que el intervalo de checkpoint del master se aumenta a 800 eventos, por lo que se realizan dos checkpoint en toda la ejecución en lugar de 10. El master envía dichos ficheros de checkpoints a su protector. Dado que recibe sólo 11 bytes por parte de los workers como resultado de la suma de la fila, también el log de dichos mensajes no es

considerable frente a los bytes que envía a los workers. La cantidad de datos enviados por el canal de control es muy baja, se utilizan paquetes de 128 bytes y el diálogo del protocolo entre observadores se limita al inicio de la comunicación y a la confirmación de log de cada mensaje. El worker recibe más mensajes, y entonces, envía más log a su protector.



(a) MW Proceso master



(b) MW Proceso worker P3

Figura 3.17: M/W Sobrecarga de ancho de banda de procesos Master y Worker

Por último, la figura 3.18 representa el tráfico del proceso SPMD que se ejecutó en **N3**. En forma similar al worker, la sobrecarga del canal de control es baja y el log de mensaje es igual a la cantidad de mensajes recibidos.

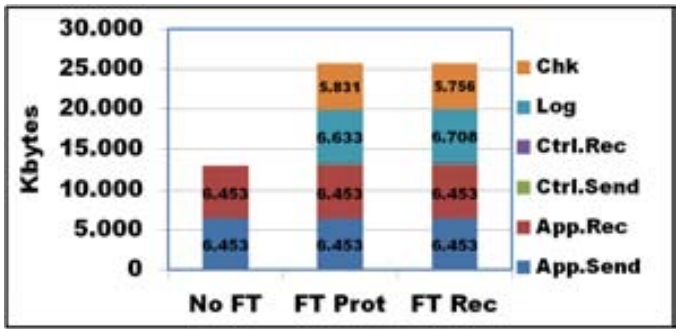


Figura 3.18: SPMD Sobrecarga de ancho de banda de un proceso SPMD

3.6. Conclusiones

Se ha definido un modelo de tolerancia a fallos para aplicaciones de paso de mensajes que funciona en la capa de socket. Se ha desarrollado un prototipo y se han realizado pruebas con dos modelos de aplicaciones que representan paradigmas de programación paralela muy utilizados en el ámbito científico. Los resultados nos permiten concluir que el modelo provee un mecanismo automático para finalizar una aplicación paralela cuando se presentan fallos en los nodos.

Este middleware permite incluir la tolerancia a fallos en la capa de socket, y de este modo, el usuario puede elegir la librería de paso de mensajes que mejor se adapte a sus necesidades y tolerar fallos de nodos durante la ejecución.

El estudio de tráfico nos muestra la sobrecarga que anade el protocolo pesimista de log de mensajes. Cada mensaje recibido por un proceso ha de ser enviado a otro nodo para ser protegido y dicha operación se realiza antes que el mensaje sea utilizado y que el proceso emisor pueda continuar. Esto puede ser causa de una alta sobrecarga en aplicaciones limitadas por comunicación, es decir, cuyas tareas que definen su camino crítico dependen de resultados de comunicaciones. Si consideramos además el caso de nodos *multicores*, un mensaje intercambiado entre procesos de un mismo nodo, tiene la misma latencia que un mensaje entre dos nodos, perdiendo la ventaja de rapidez en comunicaciones de esta arquitectura. Este problema es tratado en el capítulo siguiente.

Capítulo 4

Protocolo de Checkpoint Semi-Coordinado

4.1. Introducción

Las estrategias de tolerancia a fallos utilizadas para las aplicaciones de paso de mensaje necesitan adaptarse a los sistemas multicore.

El protocolo de *rollback-recovery* de log de mensaje pesimista basando en receptor combinado con checkpoint no coordinado fue adoptado por RADIC porque tiene un comportamiento distribuido y descentralizado. El diseno de esta arquitectura fue desarrollado para clústers de nodos *monocores*, donde se ejecutaba un sólo proceso paralelo por nodo [20].

Investigaciones que comparan las distintas alternativas de tolerancia a fallos para aplicaciones paralelas de paso de mensaje concluyen que los protocolos de *rollback-recovery* pesimistas basados en receptor son los más eficientes durante la recuperación, pero tienen un alto coste durante ejecución libre de fallos [48] [49] [50].

La sobrecarga anadida por este protocolo en la fase de protección se debe al tiempo de grabar cada mensaje recibido en almacenamiento estable, que es realizado en forma sincrónica con el envío del mensaje. La latencia de cada *send* es teóricamente duplicada ya que al menos dos saltos son necesarios. Uno, para llegar el proceso receptor y otro, para llegar al almacenamiento estable.

Cuando el proceso emisor y receptor residen en un mismo nodo, la demora que se anade para proteger el mensaje se incrementa a mucho más que el doble de la latencia del envío, debido a la diferencia de ancho de banda entre la red intra-node con la inter-node. Es por esto que la desventaja de rendimiento del protocolo pesimista es aún más notable en los sistemas multicores.

La garantía de finalizar correctamente la ejecución de una aplicación paralela de paso de mensaje aún en presencia de fallos tiene un coste en rendimiento de la aplicación. Este coste tiene dos partes. Por un lado, el coste de la protección, y, por otro, el de recuperación.

Los procesos que se ejecutan en un mismo nodo no son independientes con respecto al fallo tal como explica Bouteiller [22], y las tareas de recuperación están relacionadas ya que cuando hay fallos, los procesos se reinician y reejecutan al mismo tiempo.

Denominamos **grupo** a los procesos paralelos que se ejecutan en un mismo nodo. Si se utiliza checkpoint coordinado entre los miembros de un grupo, ya no es necesario que se guarden los mensajes intercambiados por el grupo, evitándose así la sobrecarga del log en estos casos. En contrapartida, se anade un tiempo de coordinación que evita mensajes en tránsito para conseguir una recuperación consistente, libre de mensajes huérfanos.

Este capítulo propone y desarrolla un nuevo protocolo denominado checkpoint semi-coordinado, que combina al log de mensaje pesimista basado en receptor con checkpoint coordinado, aprovechando las ventajas de ambos en los sistemas multicore.

Utilizamos la siguiente nomenclatura. Consideramos que el sistema está compuesto por N nodos, y cada uno de estos se representan con i de 1 a N. Hay un protector T_i en cada nodo. Cada grupo tiene de M_i miembros (procesos), siendo M_i la cantidad de miembros del grupo del nodo i. Se utiliza la variable j con valores de 1 a M_i para representar a cada uno de estos. Así, por ejemplo, P_{ij} corresponde al proceso paralelo que se ejecuta en el nodo i número j.

La siguiente sección explica el funcionamiento del protocolo semi-coordinado, y se muestra de que modo se decrementan las sobrecargas y el coste de coordinación que se anade. Por último, se presenta la evaluación experimental realizada para comparar el comportamiento de ambos protocolos en casos de aplicaciones que utilizan el paradigma de programación Master/Worker y SPMD.

4.2. Protocolo de checkpoint semi-coordinado

El protocolo de checkpoint semi-coordinado es una alternativa que se propone incorporar a RADIC para ser utilizado en clústers multicore.

Los procesos paralelos miembros de un grupo están relacionados cuando ocurre un fallo debido a que tienen que ser reiniciados y reejecutados hasta el mismo momento. Además, si están localizados en el mismo nodo, probablemente tengan una relación dada por la aplicación por la cuál se necesite una comunicación intensiva en volumen o en rapidez. En estos casos, utilizar una coordinación entre ellos puede ser una mejor opción que el log pesimista.

En los sistemas multicore, la latencia que anade el log pesimista en las comunicaciones entre procesos en el mismo nodo provoca una caída en el rendimiento de los procesos paralelos. Sin embargo, al utilizar una coordinación para realizar un checkpoint de cada uno de ellos se anade una nueva sobrecarga de esta tarea necesaria para evitar mensajes en tránsito. Existen en la literatura varios algoritmos para realizar esta coordinación, siendo los más conocidos el de Chandy-Lamport [40] y el de Coti [41]. La comunicación entre miembros del grupo tiene que detenerse antes de realizar el checkpoint para evitar mensajes en tránsito.

Esta sección explica la estrategia usada para coordinar los procesos de un grupo y como se combina con el protocolo pesimista basado en receptor que se continua utilizando para mensajes que provienen de procesos localizados en otros nodos del clúster.

En primer lugar se presenta el sistema de tolerancia a fallos en capa de socket presentado en capítulo anterior 3.3 a través del modelo de comunicaciones que establecen los observadores de los procesos paralelos con el con otros observadores y protectores, cuando se está ejecutando una aplicación en un sistema. Estas relaciones son las utilizadas para el intercambio de mensajes durante la gestión de protección, detección y recuperación. En segundo lugar, se especifican los cambios que se realizan en los modelos de RADIC explicados en el capítulo anterior, para incorporar al protocolo semi-coordinado. Se continua con el detalle del protocolo de coordinación entre miembros de un grupo que se lleva antes de un checkpoint y por último se describe al protocolo semi-coordinado durante la fase de protección y recuperación.

4.2.1. Tolerancia a fallos en la capa de socket basada en RADIC

En el capítulo anterior se explicaron los modelos de protección y recuperación, detallando los procedimientos que siguen los componentes distribuidos de RADIC, observadores y protectores. La figura 4.1 es un diagrama de una aplicación paralela de paso de mensaje en ejecución en un sistema multicore con un sistema de tolerancia a fallos basado en RADIC.

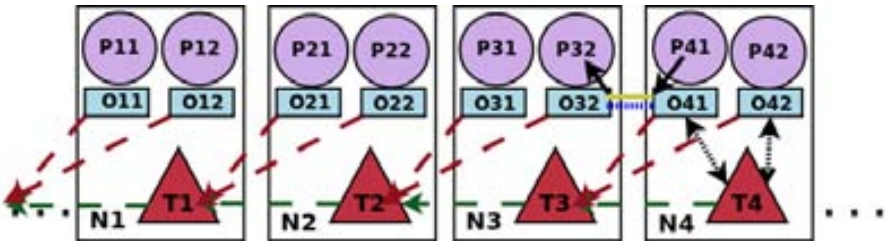


Figura 4.1: RADIC en Multicore. O_{ij} envía los datos críticos al protector T_{i-1}. T_i envía senal de heartbeat a T_{i-1}

Los observadores gestionan los siguientes cinco tipos de sockets para llevar a cabo sus funciones de salvar el estado de los procesos y mantener las comunicaciones seguras, también representados en la figura 2.3.

1. **Socket virtual:** identificador del socket conocido por el nivel de aplicación. Está representado por la línea negra que conecta a los procesos P6 y P7 con sus observadores.

2. **Socket real:** es el socket que realmente conecta con el proceso remoto, ya que la conexión original puede haberse cortado luego de un checkpoint o un fallo. Está representado por la línea amarilla.

3. **Socket de control:** es el socket que se utiliza para comunicar a dos observadores participantes de una comunicación de la aplicación y se utiliza para intercambiar información de control de mensajes e identificación. Representado por la línea punteada azul.

4. **Socket Protector:** Usado por cada observador O_{ij} para enviar los mensajes recibidos y el checkpoint al almacenamiento estable mantenido por su protector T_{i-1}, dibujados con líneas discontinuas rojas.

5. **Socket Protector Local:** Usado por cada O_{ij} para actualizar el estado del protector T_{i-1} en caso de fallos y durante la recuperación, para recibir los mensajes a reprocesar durante la reejecución y actualizar el estado del sistema. Se representa con una línea negra punteada.

4.2.2. Cambios en los modelos de RADIC

En RADIC, los protectores conocen los procesos que están en ejecución en su nodo y los observadores pueden identificar si el proceso remoto está ubicado o no en su mismo nodo por la información intercambiada por el socket seguro.

El componente apropiado para llevar a cabo el proceso de coordinación entre miembros de un grupo es el protector porque conoce a los procesos que se ejecutan en su nodo y para no anadir tareas adicionales al observador. Hasta ahora, el socket que establece cada observador con su protector local era utilizado sólo en caso de fallos y durante la recuperación. Ahora se anade esta nueva tarea de soporte a la coordinación.

La figura 4.2 representa a una aplicación paralela ejecutando en un clúster de N nodos multicore. Cada nodo i, con i de 1 a N, tiene un grupo G_i de M_i miembros, siendo M_i la cantidad de miembros del grupo del nodo N_i.

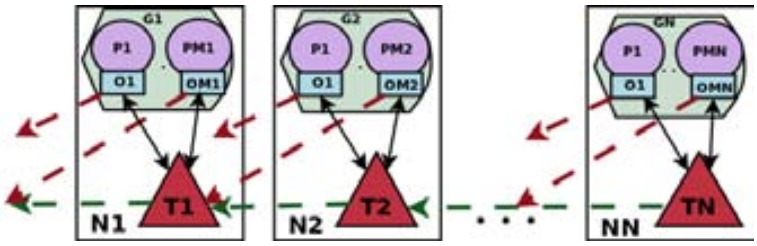


Figura 4.2: Grupos de procesos paralelos en un Clúster Multicore

El evento de checkpoint se inicia luego de cada intervalo de tiempo configurado para el grupo o bien, al finalizar una reejecución, o cuando se está por completar el espacio asignado para log de mensajes. El protector local T_i coordina a cada miembro del grupo G_i para silenciar las comunicaciones entre ellos antes de un checkpoint.

Inicialmente, los procesos que comparten un nodo pertenecen a un mismo grupo. Sin embargo, en caso de fallos, cuando no hay nodos de repuesto, RADIC recupera a los procesos en el nodo del protector que guarda los datos críticos, formado por el último checkpoint y los mensajes recibidos. En estos casos, dicho protector es capaz de reconocer la existencia de dos grupos diferentes, el que estaba ya ejecutando en su nodo y el nuevo que se recupera. Se mantienen no coordinados, realizando checkpoint en su frecuencia original y registrando en el log los mensajes intercambiados entre ellos hasta el final de la ejecución. Este comportamiento se debe a que consideramos que el nodo puede repararse o bien se puede anadir uno de repuesto, en cuyo caso, los procesos se mueven recuperando la configuración de despliegue de nodos original para evitar la degradación de rendimiento hasta el final de la ejecución.

El protector de RADIC podría generar más de un grupo con los procesos que ejecutan en su nodo cuando estos tienen un intervalo de checkpoints diferente. En este caso, los procesos se agruparían no sólo por estar en el mismo nodo, sino también por igualdad de intervalo. Esta funcionalidad podría ser útil cuando coexistan en un mismo nodo, procesos con un patrón de comunicación diferente que requiera un intervalo distinto para anadir menos sobrecarga de checkpoint. En un caso extremo, si cada uno de los procesos tiene un intervalo diferente, se vuelve a la configuración del protocolo no-coordinado que se disponía hasta el momento.

4.2.3. Checkpoint coordinado

El procedimiento que se utiliza para la coordinación de procesos de un grupo se muestra en la figura 4.3. Hay dos entidades participantes. La primera es el protector T_i con i de 1 a N, siendo N el número de nodos representados en la figura 4.2. En el nodo **i**, hay 1 a

M_i procesos paralelos a coordinar. Cada uno de estos, tiene asociado a un observador O_{ij}, donde j representa a cada uno de los procesos paralelos en el nodo i, con valores de 1 a M_i.

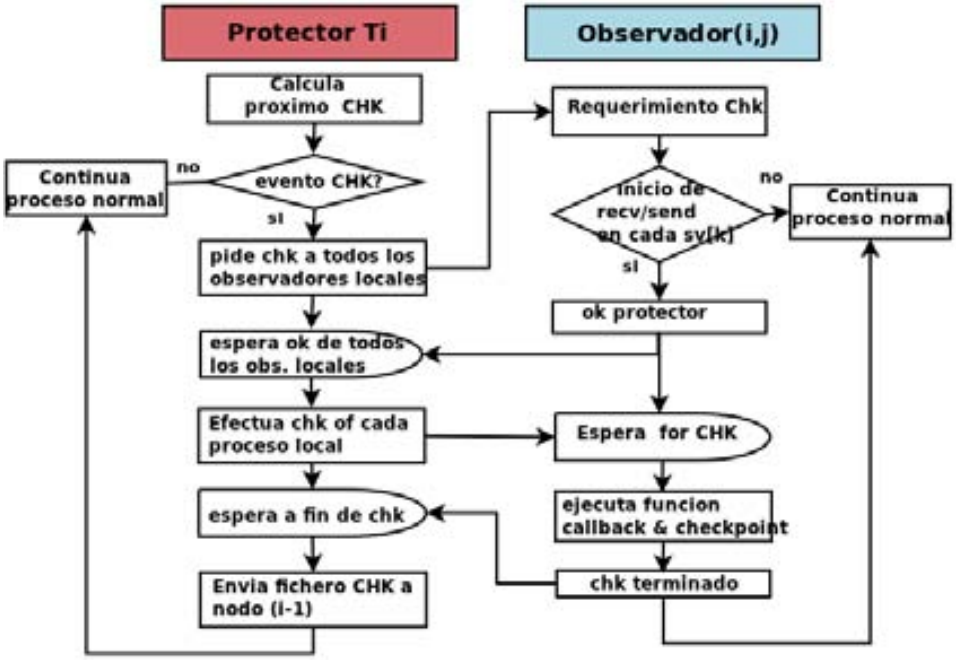


Figura 4.3: Protocolo de Checkpoint Coordinado

Cuando T_i detecta que es momento de realizar un checkpoint porque, por ejemplo, se ha cumplido el intervalo de checkpoint configurado para el grupo de procesos del nodo i, envía un mensaje a cada uno de los observadores locales O_{ij}

Luego de recibir el mensaje, O_{ij} interrumpe su actividad de comunicación al inicio de la primera operación de envío *send* o al final de una recepción *recv* y confirma al protector local que ya está preparado para que se realice el checkpoint. En este momento, no hay mensajes en tránsito porque todos los envíos están finalizados y confirmados. Una vez que cada O_{ij} ha confirmado al protector que está preparado para el checkpoint, el protector invoca a la función de BLCR, quien a su vez, invoca a la función de *callback* tal como se ha explicado en el capítulo anterior 3.3.2.

Cada observador O_{ij} avisa a su protector local cuando termina el checkpoint. Al finalizar todos, el protector T_i envía los ficheros de checkpoint al protector que se encuentra nodo anterior T_{i-1}.

4.2.4. Protocolo semi-coordinado en fase de protección

El observador mantiene en la estructura de datos socketable la identificación de cada uno de los procesos paralelos remotos de cada socket virtual. Tal como se explicó en la

sección 3.2.5 y se representó en la figura 3.4, esta información se recibe por el socket de control cuando se realiza la primer conexión del socket seguro, y esta formado por el identificador del nodo, el del proceso, y el del socket virtual.

Para dar soporte a este protocolo semi-coordinado, se anade un nuevo atributo a la identificación, que es el identificador del grupo.

Con este valor, el observador puede reconocer si el proceso remoto pertenece o no a su mismo grupo. El nodo no es suficiente porque procesos con diferente intervalo de checkpoint pueden estar localizados en un mismo nodo y no ser del mismo grupo, y, en una recuperación, los procesos que provienen de un nodo fallido no pertenecen al mismo grupo que los procesos que ya estaban ejecutándose en el nuevo anfitrión.

La identificación del grupo es asignada al inicio de la ejecución por cada protector local y se envía a cada observador en el momento de establecer la conexión. Por defecto, el valor inicial es 0. Cuando un protector reinicia un conjunto de procesos en su nodo, le asigna un nuevo identificador de grupo para diferenciarlo de los grupos que ya tuviera en ejecución.

La figura 4.4 representa al procedimiento utilizado por cada observador cuando intercambia mensajes con otro miembro de su mismo grupo.

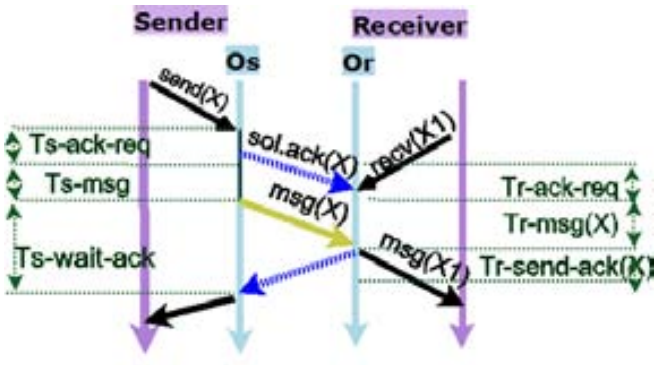


Figura 4.4: Protocolo de log de mensajes entre grupos

La diferencia con el procedimiento explicado previamente en 3.10 es que el observador receptor O_r no realiza el log del mensaje en el protector y devuelve la confirmación inmediatamente después de leer el mensaje. La sobrecarga de los procesos emisor y receptor se reduce en el tiempo denominado Tr-save-msg(X) en las comunicaciones de grupo.

Cuando los procesos no son del mismo grupo, los observadores mantienen el comportamiento definido hasta ahora representado en la figura 3.10.

Es importante tener en cuenta que aunque la sobrecarga se reduce al utilizar este protocolo semi-coordinado, puede ocurrir que el tiempo total de ejecución no se reduzca cuando se esté en cualquiera estas situaciones:

- Las comunicaciones entre miembros de grupos no sea considerable.
- La aplicación está limitada por cómputo, en cuyo caso, el camino crítico esta determinado por el tiempo de ejecución de los procesos paralelos, y un ahorro en comunicaciones no se traduce en una mejora en el tiempo total de la aplicación. En estos casos, la comunicación se solapa con el cómputo, y la disminución de sobrecarga en comunicaciones no implican una mejora en el tiempo total.
- El tiempo en que los procesos detienen su ejecución debido a la coordinación previa a los checkpoint es mayor a la reducción de log de los mensajes del grupo.

4.2.5. Protocolo semi-coordinado en fase de recuperación

El protocolo de checkpoint semi-coordinado cambia la recuperación que se ha explicado antes en 3.4.2 porque en caso de fallos, todos los procesos de un grupo se reejecutan en forma simultánea.

El procedimiento de recuperación dibujado en la figura 3.11 es aplicado de igual manera cuando el socket virtual conecta con un proceso remoto que no pertenece al mismo grupo. En cambio, si es del mismo grupo, significa que el proceso remoto está en reejecución en ese momento y que no hay mensajes grabados previos. Dichos sockets se reconectan al inicio del proceso de recuperación. Las operaciones de envío no se evitan porque aunque son reenvíos, el proceso remoto no los ha guardado. De igual modo, las operaciones de recepción de datos se vuelven a realizar utilizando lo que el proceso remoto reenvía.

El protocolo de checkpoint semi-coordinado no mejora al ya eficiente procedimiento de recuperación del protocolo pesimista. Las operaciones de envío y recepción son ahora efectuadas en lugar de evitarse y buscarlas en almacenamiento local. Se podrían observar mejoras en tiempo cuando se utiliza un nodo de repuesto, en cuyo caso, la transferencia de datos hacia el nuevo nodo será menor al tener un menor log en el almacenamiento estable.

4.3. Validación experimental

La librería del observador y el protector han sido modificados para que consideren el uso del protocolo semi-coordinado durante las fases de protección y recuperación. Cuando se recibe un mensaje proveniente de un miembro del grupo, no se envía al protector y se confirma la recepción en forma inmediata al emisor. Se cambia el proceso de recuperación para que las comunicaciones del grupo de reconecten al inicio y se vuelvan a repetir las operaciones. También se anade el atributo de grupo en la tabla socketable y su intercambio

durante la identificación con el proceso remoto. El cumplimiento del intervalo de checkpoint ya no está ahora controlado por el mismo observador, sino que se espera un mensaje del protector local para la coordinarse con el resto de procesos. Se modifica al protector para que envíe la identificación del grupo de procesos locales, ya sea al inicio de la ejecución de la aplicación paralela como al reiniciar un conjunto de procesos a partir del checkpoint. Se anade la gestión del protocolo coordinado y la realización del checkpoint coordinado.

Realizamos una validación experimental con el objetivo de comparar los protocolos no coordinado y semi-coordinados implementados con RADIC en capa de socket.

Los experimentos fueron ejecutados en un clúster de 4 nodos con procesador Intel® Core™ i5-650 con 6GB RAM, y red Network Gigabit Ethernet. Cada nodo cuenta con 2 cores. El sistema operativo es Ubuntu 10.04 Kernel 2.6.32-43-server.

4.3.1. Aplicaciones

Utilizamos dos aplicaciones que representan los patrones de comunicaciones más utilizados en las aplicaciones paralelas en el ámbito científico. La primera es una aplicación de suma de matrices con un patrón Master/Worker y la otra de transferencia de calor del tipo *SPMD Single Program Multiple Data* ambas basadas en socket TCP.

Estas aplicaciones nos permiten estudiar el comportamiento de ambos protocolos en situaciones diferentes. Por un lado, en la aplicación M/W, no hay comunicación entre workers, por lo que es de esperar que los grupos se vean afectados por la sobrecarga de la coordinación mientras que no tienen ganancia por evitar el log de mensajes.

En cambio, la aplicación SPMD ejecuta procesos fuertemente acoplados con comunicación entre todos sus procesos, y se espera que mejore la productividad al utilizar el protocolo semi-coordinado.

La aplicación de suma de matrices se ejecutó utilizando ocho workers, dos por nodo. Se ubican master y primer W1 en el N1, W2 y W3 en N2, y así sucesivamente.

La aplicación de transferencia de calor, se ejecutó con 8 procesos, también dos por nodo. Se ejecutan P1 y P2 en el nodo N1, P2 y P3 en el nodo N2 y así sucesivamente.

Se utilizan dos procesos por nodo por estar utilizando una plataforma dual-core.

4.3.2. Metodología

Se han realizado tres tipos de ejecución.

1. *No FT:* No se utiliza tolerancia a fallos, y se realiza para tener una referencia de la productividad de cada proceso y del tiempo de ejecución.

2. *FT Protección:* Se utiliza la tolerancia a fallos para proteger dicha ejecución mediante checkpoint y log de mensaje. Permite comprobar el funcionamiento de los mecanismos de protección de checkpoint semi-coordinado, valorar la sobrecarga durante la ejecución libre de fallos, y compararla con la del anterior protocolo pesimista no-coordinado basado en receptor.

3. *FT Recuperación:* Se utiliza la tolerancia a fallos para proteger la ejecución mediante checkpoint y log de mensaje y se inyecta un fallo en el nodo **N3** que provoca una caída de los dos procesos que se ejecutan en dicho nodo. Como no se dispone de nodos de repuesto, los procesos se reinician en el nodo **N2**. Este tipo de ejecución con fallos nos permite comprobar el funcionamiento del protocolo de checkpoint semi-coordinado en recuperación, valorar la sobrecarga y compararla con la del anterior protocolo pesimista no-coordinado basado en receptor.

Los checkpoints son realizados en los procesos que se ejecutan en el nodo N3, para evitar las reconexiones de sockets que provocan los checkpoints y así facilitar la comparación. El intervalo de checkpoint es gestionado por el protector, y se utiliza un modelo basado en intervalos de *heartbeat*, fijado en 9 segundos. El primer checkpoint que contiene un estado inicial de los procesos, es realizado al tercer *heartbeat*. Los siguientes checkpoints, se realizan cada 13 senales de *heartbeats*. Cabe aclarar que, en el caso del protocolo semi-coordinado, este es el momento en que el protector inicia el protocolo de coordinación con los observadores locales.

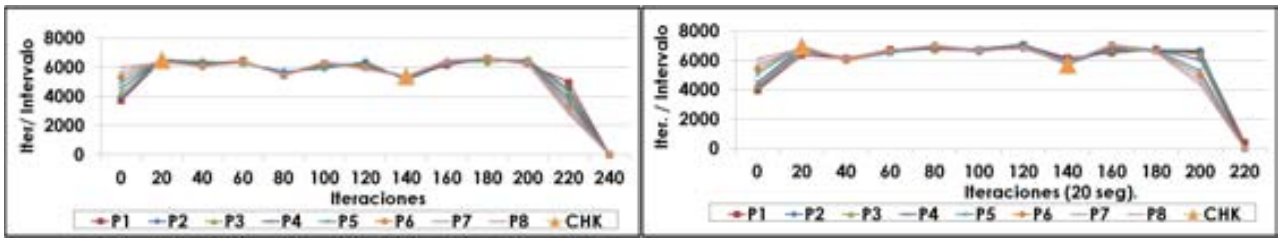
Se utiliza un diagrama de productividad que mide el trabajo por unidad de tiempo que realiza cada proceso. Esta métrica permite observar como impactan las sobrecargas en cada proceso a lo largo de su actividad. También se mide y se compara el tiempo de ejecución.

Para representar el gráfico de la productividad se agrupa el trabajo realizado por cada proceso durante un intervalo de tiempo medido en segundos. El eje X muestra los distintos intervalos y el eje Y, el trabajo realizado durante dicho intervalo. El trabajo de los procesos de SPMD se mide en iteraciones y la de los procesos Master y Worker se mide en filas. Cabe aclarar que los procesos SPMD realizan un intercambio de los bordes cada 50 iteraciones. Esta modificación se ha realizado para tener una aplicación con mayor carga de comunicaciones.

Se realizaron al menos 10 ejecuciones de cada uno de los casos y se comprobó el funcionamiento esperado de protección, detección y recuperación. Se selecciona un caso para cada una de las aplicaciones con un valor medio de tiempo de ejecución y se muestra y explican los resultados obtenidos en el siguiente apartado.

4.3.3. Experimentos

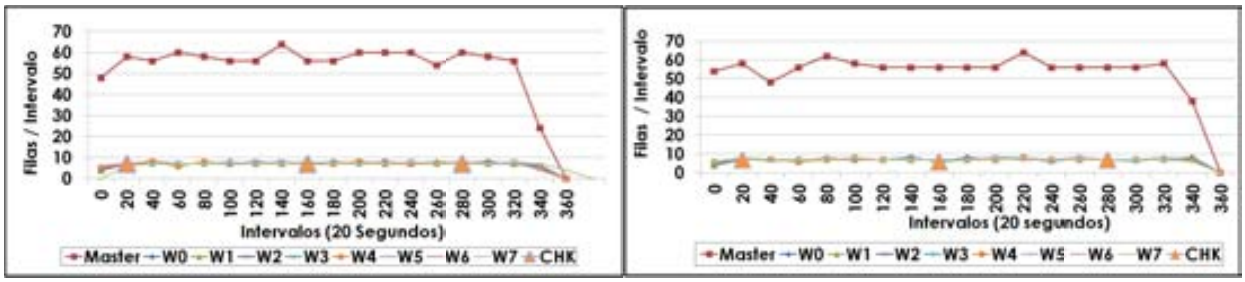
Las ejecuciones de la aplicación SPMD con protección sin fallos se muestran en la figura 4.5. Se refleja el fuerte acoplamiento de dichos procesos. Se observa una mejor productividad cuando se utiliza un protocolo semi-coordinado debido a que se añade menor sobrecarga al evitar el log en los grupos de procesos, y en consecuencia, el tiempo de ejecución es menor en este caso.



(a) Protocolo No Coordinado (b) Protocolo Semi-Coordinado

Figura 4.5: Ejecuciones de aplicación SPMD libre de fallos

La figura 4.6 muestra la ejecución libre de fallos de la aplicación M/W. Al no haber comunicación entre los workers, el uso del checkpoint semi-coordinado baja la productividad debido a que los workers paran su ejecución durante la coordinación antes de realizar cada checkpoint. No se percibe ventaja al utilizar el protocolo semi-coordinado en este caso.

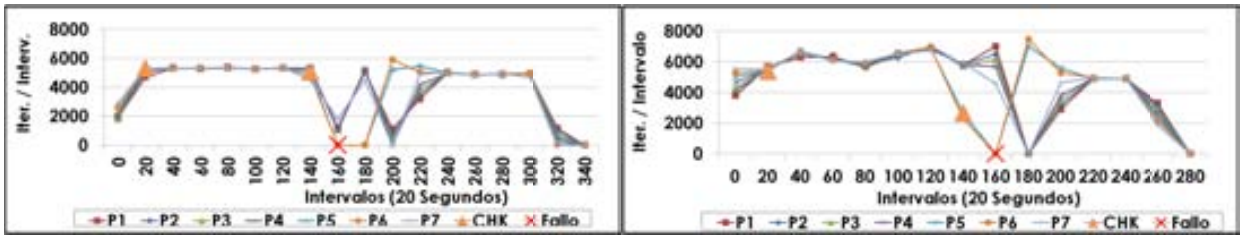


(a) Protocolo No Coordinado (b) Protocolo Semi-Coordinado

Figura 4.6: Ejecuciones de aplicación M/W libre de fallos

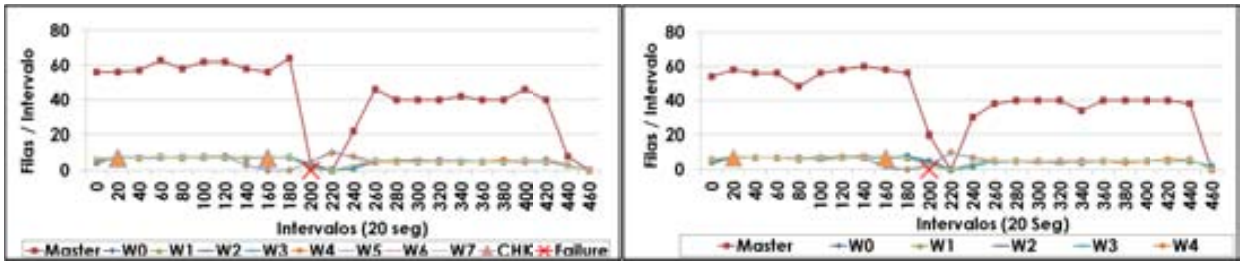
Las ejecuciones de la aplicación SPMD con recuperación se muestran en la figura 4.7. Se observa un mejor rendimiento cuando se utiliza el protocolo semi-coordinado también en la recuperación, ya que en la reejecución se alcanza un máximo de iteraciones.

La figura 4.8 muestra la ejecución con recuperación de la aplicación M/W. En este caso, el protocolo no coordinado muestra ligeramente un mejor comportamiento que el semi-coordinado durante la reejecución debido a que los datos están disponibles en forma local.



(a) Protocolo No Coordinado (b) Protocolo Semi-Coordinado

Figura 4.7: Ejecuciones de aplicación SPMD con recuperación

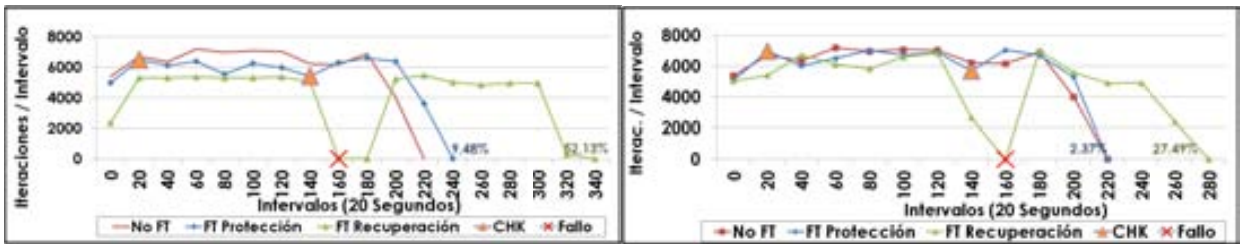


(a) Protocolo No Coordinado (b) Protocolo Semi-Coordinado

Figura 4.8: Ejecuciones de aplicación M/W con recuperación

La figura 4.9 compara las diferentes ejecuciones No FT, *FT Proteccion* y *FT Recuperacion* del proceso P5 ubicado en el nodo N3.

Durante la protección, se observa una caída mayor de productividad cuando se utiliza el protocolo no coordinado que con el semi-coordinado. El tiempo de ejecución sólo se incrementa en un 2.37 % con respecto al tiempo de referencia sin utilizar tolerancia a fallos. En cambio, usando el protocolo no coordinado la ejecución necesita un 9.48 % más de tiempo para terminar. La recuperación usando semi-coordinado muestra también un mejor funcionamiento, y se añade un 27.49 % en lugar de 52.13 % de la ejecución con protocolo no coordinado.



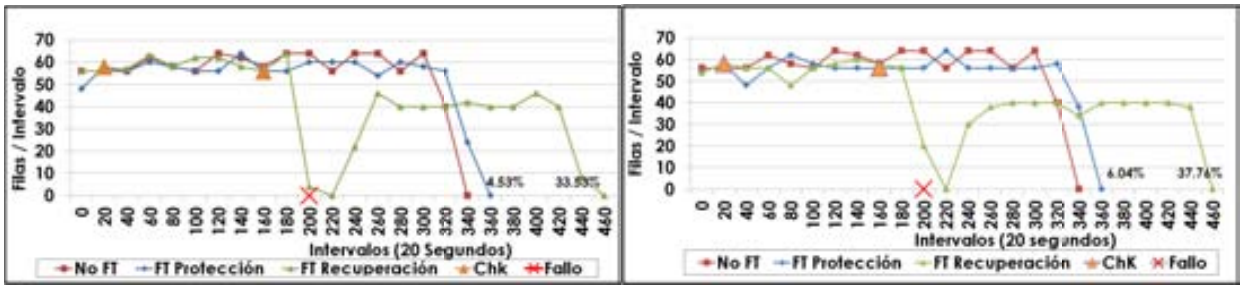
(a) Protocolo No Coordinado (b) Protocolo Semi-Coordinado

Figura 4.9: Ejecuciones del proceso P5 de la aplicación SPMD

Las distintas ejecuciones del worker ubicado en el N3 se grafica en la figura 4.10. A

diferencia de lo observado con la otra aplicación, aquí el uso del protocolo no coordinado es ligeramente mejor que el semi-coordinado. Las ejecuciones requieren 4.49 % y 33.23 % cuando no hay fallos y con recuperación respectivamente mientras que el semi-coordinado necesita un 5.99 % y un 37.43 % más de tiempo, todos comparados con la ejecución de referencia sin tolerancia No FT.

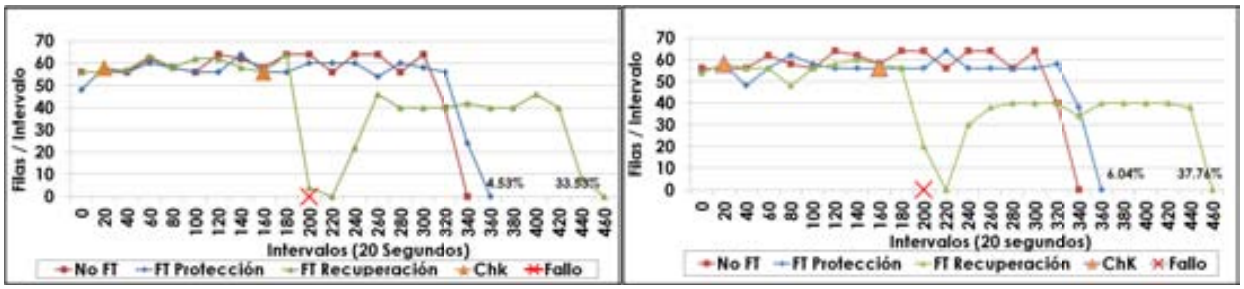
Se puede observar como el número de filas por segundo disminuyen en este último caso a causa del tiempo de coordinación añadido en cada checkpoint y, no hay beneficio porque los workers no se comunican entre sí.



(a) Protocolo No Coordinado (b) Protocolo Semi-Coordinado

Figura 4.10: M/W Comparación de Ejecuciones del worker W4

Las distintas ejecuciones del master ubicado en el N0 se grafica en la figura 4.11. La productividad de este proceso se recupera un poco antes en el caso del protocolo no coordinado. En ambos casos, se ve afectado por la recuperación de los workers en un mismo nodo.



(a) Protocolo No Coordinado (b) Protocolo Semi-Coordinado

Figura 4.11: M/W Comparación de Ejecuciones del Master

Para valorar si los resultados observados en las comparaciones realizadas se mantienen cuando se utiliza un mayor tamaño del paquete intercambiado por los procesos, se realiza una serie de experimentos con aplicaciones que requieren más comunicaciones que cómputo, ya que en principio, es en estos casos cuando el protocolo semi-coordinado es de mayor utilidad.

La tabla 4.12 muestra los tiempos de ejecución libre de fallos y con recuperación de la aplicación SPMD. Se observa que el protocolo no coordinado es mejor para paquetes pequeños. La comunicación en estos casos se solapa con el cómputo por lo que el log no incrementa el tiempo de ejecucion. Sin embargo, el tiempo de coordinación para los procesos se añade directamente en el camino crítico e impacta directamente.

SPMD Workload	PAQ.	NOFT	Protection			
			Uncoord.		Semi-Coordinated	
			Sec.	Ov.	Sec.	Ov.
1000x600	2K	168	186	10,7%	199	18,5%
1000x1000	4K	263	304	15,6%	295	12,2%
1000x1500	6K	370	435	17,6%	394	6,5%

(a) Protocolo No Coordinado

SPMD Workload	PAQ.	NOFT	Recovery			
			Uncoord.		Semi-Coordinated	
			Sec.	Ov.	Sec.	Ov.
1000x600	2K	168	217	29,2%	208	23,8%
1000x1000	4K	263	324	23,2%	319	21,3%
1000x1500	6K	370	473	27,8%	483	30,5%

(b) Protocolo Semi-Coordinado

Figura 4.12: Ejecuciones SPMD con diferente tamaño de paquete

Del mismo modo, en las ejecuciones de la aplicación M/W se observa un comportamiento similar, ya que el protocolo no coordinado también es la opción preferente para tener un mejor tiempo de ejecución cuando el tamaño del paquete es pequeño. Los resultados se muestran en la tabla 4.13. Cuando el tamaño de paquete aumenta, el protocolo semi-coordinado es una mejor opción para los dos casos de aplicaciones probadas.

MW WORKLOAD	PAQ.	NOFT	Protection			
			Uncoord.		Semi-Coordinated	
			Sec.	Ov.	Sec.	Ov.
10000x5000	55K	241	265	10,0%	269	11,6%
10000x6000	66K	283	325	14,8%	316	11,7%
10000x7000	77K	322	355	10,2%	354	9,9%

(a) Protocolo No Coordinado

MW WORKLOAD	PAQ.	NOFT	Recovery			
			Uncoord.		Semi-Coordinated	
			Sec.	Ov.	Sec.	Ov.
10000x5000	55K	241	301	24,9%	306	27,0%
10000x6000	66K	283	363	28,3%	359	26,9%
10000x7000	77K	322	401	24,5%	398	23,6%

(b) Protocolo Semi-Coordinado

Figura 4.13: Ejecuciones M/W con diferente tamaño de paquete

4.4. Conclusiones

En este capítulo se ha detallado la propuesta de incorporar un protocolo de checkpoint semi-coordinado que permite mejorar el rendimiento de la tolerancia a fallos en sistemas multicore.

Los componentes diseñados de la arquitectura RADIC disponen de toda la información que permiten llevar a cabo la coordinación a nivel de nodo y de hacer grupos de coordinación en un mismo nodo.

El protocolo de coordinación propuesto es flexible y permite configurarlo para adaptarse a distintos sistemas y comportamientos de aplicaciones.

Capítulo 5

Computador Paralelo libre de fallos para aplicaciones MPI

5.1. Introducción

En este capítulo se presentan los cambios en el modelo de RADIC en capa de socket para que se toleren fallos durante la ejecución de aplicaciones que utilicen *Message-Passing Interface (MPI)* [2].

Message-Passing Interface (MPI) es el estándar de programación más utilizado por las aplicaciones científicas. Sin embargo, este no especifica mecanismos de tolerancia a fallos y tiene un comportamiento *fail-stop*, explicado por Schlichting [3], realizando una parada segura de todos los procesos en caso de detectar un fallo en cualquiera de los nodos del clúster. Como consecuencia, se pierde la ejecución que se hubiera hecho hasta el momento, se ha de realizar un tratamiento manual para solucionar la causa del problema y finalmente, reejecutar la aplicación.

En el modelo que proponemos, los fallos no llegan a manifestarse en la librería MPI, debido a que estos se resuelven en la capa de socket. La figura 5.1 muestra un diagrama de capas de una aplicación MPI que se ejecuta en un computador paralelo libre de fallos. A su vez, dicha capa se sustenta en un sistema de tolerancia a fallos basado en RADIC.

Una de las librerías que implementan MPI más utilizadas es Open-MPI [7]. Para el desarrollo de esta sección hemos estudiado el uso de la API de socket que hacen las aplicaciones de paso de mensaje compilada y ejecutada con dicha librería, en particular con la versión 1.6.1.

Para realizar una gestión de comunicaciones entre procesos más eficiente, la librería analizada utiliza un conjunto más amplio de operaciones de socket que las que hemos mencionado hasta el momento.

Figura 5.1: Computador paralelo libre de fallos para aplicaciones MPI

El observador requiere extenderse para interponer las llamadas a este nuevo conjunto más amplio de operaciones de sockets, debido a que cada una de estas funciones puede tener un impacto en el sistema de tolerancia a fallos en varios aspectos. Primero, se tienen que enmascarar el uso de identificadores de sockets que puedan haber cambiado como resultado de un checkpoint o un fallo. En segundo lugar, la nueva operación puede corresponder a un evento no determinista cuyo determinante necesita ser grabado siguiendo el protocolo de log de mensaje [46] [47]. Por último, el resultado de las operaciones de sockets necesitan ser interceptados por el procedimiento de *socket seguro* para diagnosticar fallos de comunicaciones de los procesos remotos y recuperarlos en caso de ser necesario.

La gestión de arranque, ejecución, y finalización de una aplicación paralela MPI es controlada por programas o utilitarios provistos por la implementación de la librería MPI. En cada uno de los nodos multicore, el grupo de procesos es ejecutado bajo una organización de jerarquía de árbol, cuya raíz es el utilitario y los hijos los procesos paralelos que se ejecutan en dicho nodo. El sistema operativo POSIX reconoce la relación de estos procesos mediante una identificación de grupo *Group Identification GID*, y con una identificación del padre *Parent Process Identification PPID*.

La librería BLCR que utilizamos para realizar checkpoint / restart [44] de los procesos paralelos provee una funcionalidad para guardar un único estado de ejecución del árbol de procesos, pero requiere de una instrumentación diferente a la utilizada hasta el momento para asegurar que se guarda un estado consistente que sea recuperable.

Los modelos de protección y recuperación de RADIC en la capa de sistemas requieren incorporar la funcionalidad de reconocer estas estructuras jerárquicas de procesos paralelos y salvar el estado siguiendo las recomendaciones de la librería BLCR.

En la primera sección se explican cómo se mantiene el modelo de *socket seguro* cuando se utiliza una librería MPI, controlando la ejecución de las nuevas operaciones para garantizar que la librería utiliza un mecanismo de transporte libre de errores causados por la caída

de nodos. En la siguiente sección se abordan los cambios que se realizan en los modelos de protección y recuperación de RADIC en capa de socket para considerar a las aplicaciones paralelas que utilizan las librerías MPI.

5.2. Cambios en el modelo de *socket seguro*

El modelo de *socket seguro* permite que el nivel de aplicación utilice un socket de comunicaciones libre de errores, aunque en realidad estos pueden presentarse tanto en el nodo local o como remoto.

Para asegurar este funcionamiento, se requiere que todas las operaciones que hagan referencias al socket virtual sean interceptados por el observador y la operación se efectúe utilizando el socket real, que es el identificador de la comunicación que el observador mantiene conectado con el proceso remoto a pesar de las incidencias que pudieran ocurrir durante la ejecución. Además, el *socket seguro* requiere detectar los fallos del nodo remoto para evitar que los errores se propaguen a la capa de la aplicación, haciendo que esta pare su ejecución.

La figura 5.2 representa el flujo de acciones que sigue el socket seguro para todas las operaciones interceptadas.

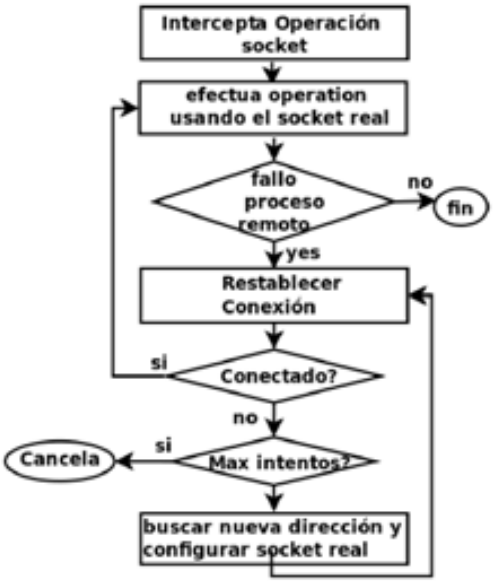


Figura 5.2: Diagrama de flujo de socket seguro

Para el mantenimiento del *socket seguro*, el observador aplica el flujo de acciones a las operaciones indicadas en la siguiente tabla 5.1 durante la ejecución de los procesos paralelos MPI.

Cuadro 5.1: Operaciones interceptadas según el tipo de función

Tipo de función	Operación	Socketable	Log de Eventos
Socket Servidor	socket-listen-bind	si	no
Aceptar Conexiones	accept	si	no
Conexiones clientes	socket-connect	si	no
Obtener config.socket	getsockopt	no	no
Recibir Datos	recv-readv	no	si
Enviar Datos	send-writev	no	si
Consulta Estado Comm.	poll-select	no	si

Se distinguen los siguientes grupos de operaciones:

Operaciones de conexión: Las operaciones relacionadas a la creación de una conexión, sea esta del tipo servidor o cliente, son las que tienen que registrarse en socketable para poder ser reejecutadas en caso de fallos durante el restablecimiento de la conexión.

Transporte de datos: Para optimizar el envío y recepción de mensajes las librerías utilizan las funciones *writev* y *readv* que, a diferencia de las básicas *send* y *recv* mencionadas hasta el momento, permiten transportar, en una única solicitud, varios segmentos de datos a la vez, en lugar de uno solo. Estas operaciones son incorporadas al modelo de socket seguro, y, además son tratadas por el protocolo de log de eventos, el cual se explica en la sección siguiente.

Consulta Estado Comunicaciones: Las librerías utilizan operaciones que permiten consultar al sistema operativo, el estado de las comunicaciones para saber si ha llegado alguna información que le corresponde procesar. Ejemplos de estas funciones son *poll* o *select*. Se indica el conjunto de identificadores de dispositivos de los cuales se consulta. El observador tiene que cambiar convenientemente los identificadores de dispositivos virtuales por los reales antes de efectuar la operación. Esta operación afecta también a los modelos de protección y recuperación, explicado en la sección siguiente.

Otras: Las operaciones de socket que se utilizan para configurar parámetros de funcionamiento, consultar dichas opciones o bien, obtener información sobre el protocolo utilizado, la dirección ip, puerto, etc. Por ejemplo, *getsockopt, setsockopt*, entre otras.

5.3. Cambios en los modelos de protección y recuperación

En esta sección se analiza y se detalla como cambian los modelos de protección y recuperación explicados en los capítulos anteriores para poder tolerar fallos durante la ejecución de aplicaciones paralelas de paso de mensaje basadas en MPI.

Se inicia describiendo la estructura de procesos paralelos que las librerías de paso de mensaje despliegan para su ejecución. Dicha estructura es importante para el uso de checkpoint/restart y para el log de mensaje. A continuación, se explican los cambios en los modelos de protección y recuperación, considerando la estructura de los procesos y las nuevas operaciones integradas por el *socket seguro*.

5.3.1. Estructura de procesos paralelos de una aplicación MPI

La gestión de arranque, ejecución, y finalización de una aplicación paralela MPI es controlada por utilitarios provistos por la implementación de la librería MPI. En cada uno de los nodos multicore, el grupo de procesos es ejecutado bajo una jerarquía de árbol, cuya raíz es el utilitario y los hijos los procesos paralelos que se ejecutan en dicho nodo. El sistema operativo POSIX reconoce la relación de estos procesos con una identificación de grupo *Group Identification GID*, y con una identificación del padre *Parent Process Identification PPID*.

El utilitario que invoca a los procesos paralelos de la aplicación toma el rol de padre, es la raíz del árbol. Los procesos paralelos son las hojas del árbol. Cada uno de estos procesos, incluyendo también al padre, es interceptado por un observador de la librería que garantiza el uso de socket seguros.

En cada uno de los nodos del clúster utilizados por la aplicación paralela, se forma una estructura de **árbol** de ejecución formado por un padre y sus hijos.

El utilitario *mpirun* es el encargado de lanzar la ejecución de los procesos paralelos en el clúster de nodos asignados para dicha ejecución, siguiendo una distribución de procesos en los nodos en base a la configuración del usuario o a un comportamiento por defecto.

La figura 5.3 muestra un ejemplo de despliegue de una aplicación paralela de paso de mensaje en ejecución con una librería MPI.

Los procesos del primer nodo con lanzados directamente por el programa *Mpirun*, mientras que en el resto de los nodos, el utilitario *orted* es el encargado de lanzar los procesos en cada uno de los nodos.

Los procesos paralelos de la aplicación tienen un padre que puede ser *mpirun* si están

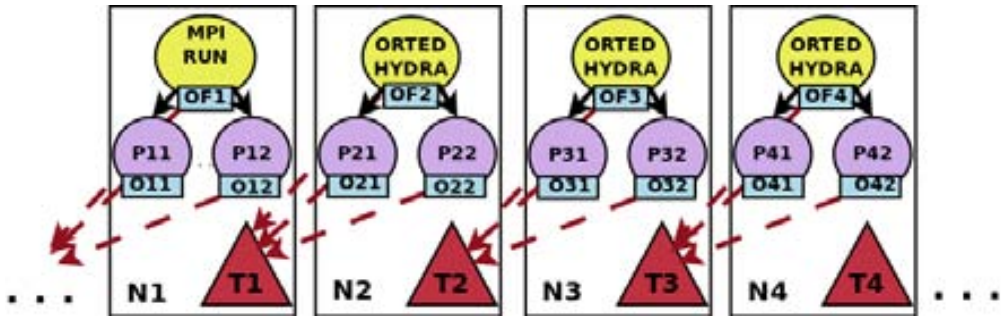


Figura 5.3: RADIC en Aplicaciones MPI

localizados en el primer nodo de lanzamiento o bien *orted*.

Esta estructura de árbol de procesos ha de ser conocida por los observadores y los protectores porque el modo de realizar checkpoint y restart cambia para este tipo de procesos.

El modelo utilizado por RADIC se basa en que los observadores, en el momento de inicializarse, durante las primeras operaciones que interceptan, reconocen si el proceso que están observando es un proceso *padre*. lo cual se determina porque es un utilitario de la librería o bien porque utiliza operaciones del tipo *clone* para la creación de los procesos hijos. Si el programa en ejecución es uno de los utilitarios de las librerías, o si se han lanzado procesos, el proceso es el padre del árbol. El atributo de si es o no padre, que denominamos *jerarquía del proceso*, se informa al protector local durante la primera conexión. El protector local, reconoce de esta manera, la jerarquía de todos los procesos que se ejecutan en su nodo.

5.3.2. Modelo de Protección

El modelo de protección se ocupa de salvar el estado de ejecución mediante checkpoint y log de mensajes.

El checkpoint de un árbol de procesos es posible realizarlo utilizando la librería BLCR. Sin embargo, hay que tener en cuenta que dicha librería guarda el estado a partir del proceso padre y que sus hijos, los procesos paralelos de la aplicación que se ejecutan en un nodo, no pueden tener mensajes en tránsito en el momento de estar salvando el estado.

En el capítulo anterior, en la propuesta del protocolo semi-coordinado, se presentó un protocolo de coordinación para utilizar con los miembros de un grupo 4.2.3.

La estructura de árbol que se crea en cada nodo del clúster que ejecuta una aplicación MPI, muestra que existe una vinculación entre los procesos de cada nodo similar a la del grupo.

Cuando se utiliza una librería MPI, se sigue el modelo de protección del protocolo semi-coordinado adaptado el caso particular de un grupo tipo árbol. Consideramos que el árbol es un grupo que tiene un único padre y uno o más hijos.

El log de los mensajes intercambiados entre miembros de un árbol, no se registran en el protector, debido a que se sigue un modelo coordinado entre ellos, aunque si se registran los que provienen desde procesos en diferentes árboles. Además, se registran los determinantes de los eventos que se generan a partir de las consultas del estado de las comunicaciones, para ser reproducidos en caso de fallo durante la reejecución. Estos eventos no corresponden a los mensajes en sí, sino al momento de la detección por parte de la aplicación.

A continuación, se explica en detalle como se llevan a cabo las tareas de checkpoint y de log de eventos cuando se protege una aplicación MPI.

Checkpoint

El checkpoint se realiza siguiendo el modelo semi-coordinado aplicado a un grupo del tipo árbol. En este modelo, el protector local, cuando se cumple el intervalo de checkpoint, coordina a los miembros del árbol para que suspendan sus comunicaciones para evitar mensajes en tránsito mientras se salva el estado.

La librería de BLCR contempla un tipo de checkpoint y restart que guarda y recupera el estado de todos los miembros del árbol a la vez. Esto se realiza invocando a la función correspondiente indicando el identificador de proceso padre del árbol. El estado de todos los miembros es salvado en un sólo fichero.

El protector local conoce la jerarquía de los procesos que se ejecutan en su nodo, que ha sido informada por cada observador durante la primera conexión.

Cuando se cumple el intervalo de checkpoint, el protector local envía un mensaje a cada uno de los observadores de acuerdo a lo indicado en el procedimiento de coordinación 4.2.3 y una vez todos respondan, se realiza el checkpoint del árbol de procesos en una sola operación y se envía al protector del nodo anterior un solo fichero correspondiente al conjunto de procesos en ejecución en el nodo.

Para garantizar que el proceso no es interrumpido para hacer un checkpoint durante una operación atómica o en una operación asincrónica, BLCR provee un API para marcar las secciones críticas de ejecución en las cuales no pueden ser interrumpidos los procesos.

Cada observador entra en zona crítica durante las tareas de log de eventos, que han sido explicadas en 3.3.2, tanto se corresponde a un *sender* o un *receiver*. Esto garantiza que al reiniciar una ejecución a partir de un checkpoint, se reejecute al inicio o al finalizar una operación de comunicaciones.

Log de eventos no deterministas

Las librerías utilizan otras operaciones de socket para el envío y recepción de mensajes además de las básicas de *send* y *recv* estudiadas hasta el momento para optimizar el transporte de datos.

Las operaciones de *writev* y *readv* permiten enviar y recibir varios segmentos de datos en una misma invocación al sistema de transporte. En cambio, con las operaciones básicas, sólo se transporta un solo segmento contiguo de datos. Para enviar y recibir varios segmentos se utiliza una estructura de vector. Para adaptar el log de mensajes explicado en los capítulos previos 3.3.2, se generaliza el funcionamiento para que en lugar de considerar un sólo segmento de datos determinado por una longitud, gestione un conjunto variable de segmentos, donde cada uno tiene una longitud diferente.

Se enumeran a continuación las adaptaciones realizados en el procedimiento de log, en referencia a cada uno de los pasos del procedimiento dibujado en la siguiente figura 5.4:

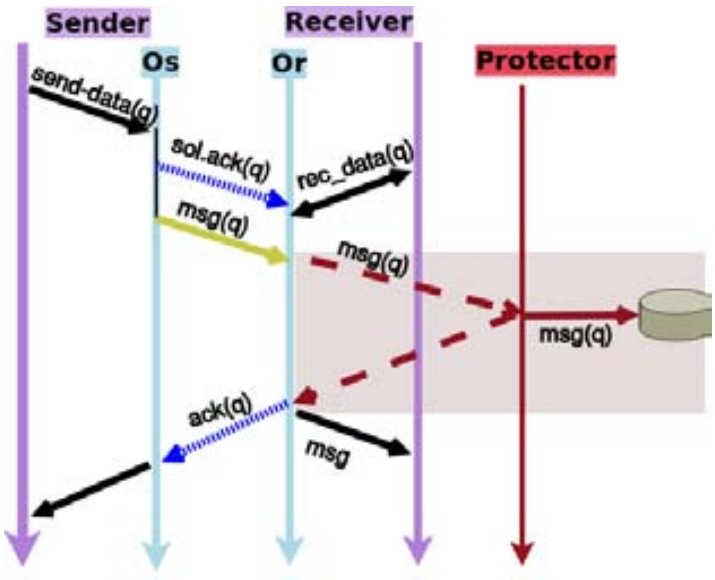


Figura 5.4: Log de Eventos para aplicaciones MPI - Socket Virtual: Línea continua negra - Socket real: Línea continua amarilla - Socket control: Línea punteada azul - Socket protector: Línea discontinua roja

1. El observador del proceso *sender* O_s interpone funciones de envío de datos, ya sean *send* o *writev* y envía por el socket de control una solicitud de confirmación con la nueva estructura que incluye la cantidad variable q de elementos de diferente longitud. Reproduce la operación interpuesta utilizando el socket real.

2. El observador del proceso *receiver* O_r interpone operaciones de recepción de datos, ya sean *recv* o *readv*, y recibe la solicitud de confirmación sol-ack enviada por

el sender a través del canal de control, con la nueva estructura, y reproduce la operación interpuesta utilizando el socket real.

3. Si el proceso remoto O_s no pertenece al mismo árbol de procesos, O_r envía al protector los q elementos, indicando la estructura de vector y se espera la confirmación de que este es guardado. Esta parte está sombreada porque no se aplica cuando O_s y O_r pertenecen al mismo árbol.

4. Una vez recibidos y guardados cuando corresponda, la totalidad de segmentos enviados, O_r envía la confirmación a O_s a través del socket de control y finaliza la operación de *recv* o *readv*.

5. O_s recibe la confirmación y finaliza la interposición.

Las operaciones que permiten consultar si hay datos disponibles en el sistema de comunicación como las funciones *poll()* o *select()*, que se han mencionado en el párrafo anterior de cambios en el *socket seguro*, son incorporadas el sistema de log. Cuando se reciben datos provenientes de un proceso remoto, la respuesta de estas operaciones provocan que la librería emita una operación de recepción de datos por ese socket. Por lo tanto, consideramos que la llegada de datos y la respuesta a la consulta es un evento no determinista que es grabado en el log de mensajes para ser reproducido de igual modo durante la reejecución del proceso.

5.3.3. Modelo de recuperación

En este apartado se explican los cambios que se realizan en el modelo de recuperación de RADIC para poder restablecer al árbol de procesos MPI cuando uno de los nodos falla.

En primer lugar, se detalla el procedimiento que se lleva a cabo para evitar problemas en la comunicación cuando no hay nodos de repuesto y se recupera un árbol en un nodo que ya tiene otro árbol. Se tienen que evitar utilizar los mismos puertos de socket servidores en una misma dirección IP, y los clientes tienen que poder ubicar a los procesos correctamente.

En segundo lugar, se exponen los cambios que se realizan en el protocolo semi-coordinado para adaptarse al caso de árboles de procesos.

La explicación se realiza asumiendo que el nodo N_{i+1} falla y que el protector T_i reinicia al árbol de procesos cuyo padre es $OF(i+1)$ en el nodo N_i y que el observador de cada hijo $M_{i+1,j}$ es $O_{(i+1),j}$.

Modelo de reconexión

Cuando uno de los nodos falla, el árbol de procesos tiene que reiniciarse en uno nuevo. Si no hay disponible uno de repuesto, se recupera en el nodo del protector.

En estos casos, al menos dos árboles comparten el mismo nodo. Esto puede traer un conflicto en las comunicaciones, ya que un puerto puede utilizarse una vez en cada dirección IP. Pueden haber puertos ya utilizados al recuperar un proceso.

El protector T_i reinicia los procesos que se ejecutaban en el nodo N_{i+1} y aumenta en uno a la variable interna que indica el número de grupos que gestiona en dicho nodo. Normalmente, si es la primera recuperación que se realiza en dicho nodo i, el valor del *grupo* es 1.

Cuando los observadores, incluidos los que se reinician, preguntan al protector T_i por el estado de alguno de los procesos que se estaban ejecutando en N_{i+1}, se le contesta la nueva IP address y el número asignado de grupo.

Este valor de grupo se usa para calcular los nuevos puertos de los socket servidores de los observadores que se están reiniciando y para los observadores clientes que requieren conectarse con un proceso reiniciado.

$$NuevoPuertoServidor = PuertoAnterior + Grupo * Desplazamiento\_Puertos \quad (5.1)$$

Desplazamiento_Puertos es un valor constante que tiene que ser mayor al número de puertos del tipo listener utilizados por un mismo procesos. Usamos un valor de 100.

Los observadores reiniciados OF_{i-1} y $O_{i-1,j}$ utilizan esta fórmula antes de efectuar la operación de *bind* en el nuevo nodo. De este modo se evitan las colisiones con puertos existentes.

Los observadores clientes, luego de recibir la dirección IP y el grupo del proceso reiniciado, crean una nueva conexión usando la nueva dirección y puerto.

Protocolo de checkpoint semi-coordinado en recuperación

Una vez reiniciado el árbol de procesos, se sigue un protocolo semi-coordinado durante la reejecución de las operaciones hasta llegar al punto de fallo.

El procedimiento es el de la figura 5.5. Las entidades involucradas son el protector local T_i y los observadores del nodo N_{i+1}, el padre del árbol $OF_{(i+1)}$ y los hijos $O_{(i+1),j}$.

- El protector T_i detecta el fallo del nodo N_{i+1}, aumenta en 1 el número de grupo y reinicia el árbol de procesos.

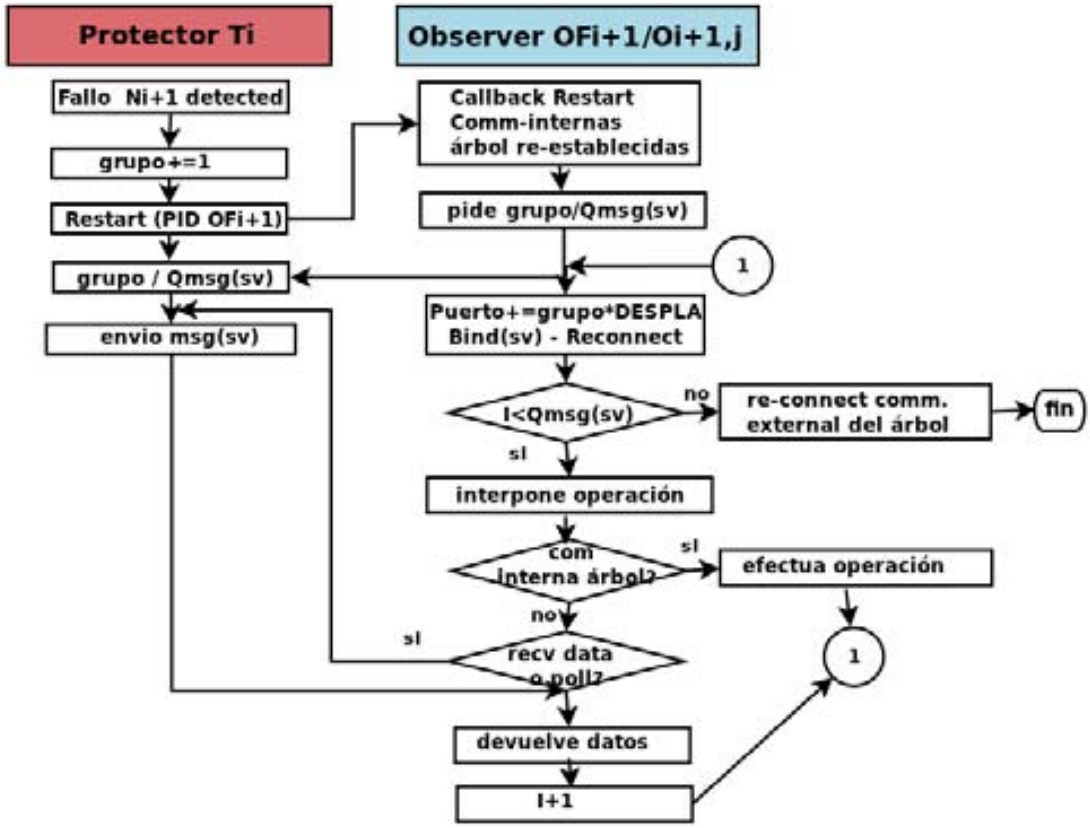


Figura 5.5: Procedimiento de Recuperación para Procesos MPI

- Cada observador ejecuta la función de *callback* en modo de recuperación. Se reconectan los padres e hijos.
- La primera operación realizada por los observadores en la reejecución es preguntar por el número del grupo y la cantidad de eventos salvados para cada uno *sv* de los socket virtuales registrado en la socketable.
- Se recalcula el valor de los nuevos puertos para cada uno de los sockets tipo servidor y se efectúan las operaciones de *bind* y *listen* de cada uno de ellos.
- La reejecución se realiza interponiendo cada uno de las funciones de socket hasta completar la cantidad Qmsg(sv) de cada socket virtual. Se representan las acciones del procedimiento relacionadas con un *sv*. Se repiten dichas acciones para cada uno de los sockets virtuales.
- Si el *sv* conecta procesos del mismo árbol, la operación se vuelve a realizar utilizando el socket real, debido a que dicha operación no fue grabada en el log

de eventos durante la primer ejecución.

- Cuando el *sv* conecta a procesos de árboles diferentes, las operaciones de envío de datos son evitadas para no repetir envío de mensajes.
- El log grabado previamente se utiliza para reejecutar operaciones de recepción de datos como *recv* o *readv* y para consulta de estado de comunicaciones como *poll*.

5.4. Validación experimental

El observador y el protector han sido modificados para que se consideren los aspectos mencionados previamente acerca de la estructura de procesos, las nuevas operaciones de socket para proteger y recuperar y los cambios en los modelos de funcionamiento.

Realizamos una validación experimental con el objetivo de probar el funcionamiento de RADIC en capa de socket durante las fases de protección del estado de ejecución mediante checkpoint y log, de detección de fallos de comunicaciones, recuperación del estado de ejecución y reconfiguración del sistema cuando se ejecuta una aplicación paralela de paso de mensaje basada en MPI.

Los experimentos fueron realizados en un clúster de 4 nodos con procesador Intel® Core™ i5-650 con 6GB RAM, y red Network Gigabit Ethernet. Cada nodo cuenta con 2 cores. El sistema operativo es Ubuntu 10.04 Kernel 2.6.32-43-server. La librería de checkpoint/restart utilizada es la BLCR 0.8.4.

Se ha utilizado la versión de Open-MPI 1.6.1 en su configuración por defecto. Si bien dicha implementación ofrece funciones de tolerancia a fallos, no fueron habilitadas así como tampoco configuradas.

Cabe aclarar que se ha analizado MPICH 1.4.1, también muy utilizada para el desarrollo de aplicaciones paralelas, y se ha observado que utiliza también una estructura jerárquica de procesos durante el despliegue y ejecución de procesos paralelos, así como un uso equivalente al estudiado en el caso de MPI. Sin embargo, la validación experimental no se ha podido realizar con MPICH debido a que durante la intercepción de funciones de socket se ha detectado una diferencia en la protección de acceso a los datos correspondientes a los sockets abiertos. Se puede resolver dicho problema mediante llamadas adicionales al sistema operativo. No se ha llegado a tiempo a realizar las modificaciones de la librería del observador para poder solventar este inconveniente.

5.4.1. Aplicaciones

Se ha ejecutado una aplicación de multiplicación dinámica de matrices del tipo Master/Worker y una aplicación de transferencia de calor del tipo *Single Program Multiple Data (SPMD)* utilizando Open-MPI versión 1.6.1.

La multiplicación de matrices utilizó ocho procesos, dos por nodo, con una matriz de cuadrada de 350 elementos dobles, con bloques de 35 elementos y realizando 1.000 iteraciones.

La aplicación de transferencia de calor, se ejecutó con ocho procesos, dos por nodo con una matriz de 250 filas por 250 columnas, con 150.000 iteraciones.

Los procesos son denominados P0 a P7, siendo P0 el distribuidor de tareas o el master en las respectivas aplicaciones SPMD y M/W. Se ejecutan P0 y P1 en el nodo N1, P2 y P3 en el nodo N2 y así sucesivamente.

Se utilizan dos procesos por nodo por estar utilizando una plataforma dual-core.

5.4.2. Metodología

Se han realizado tres tipos de ejecuciones.

No FT: No se utiliza tolerancia a fallos, y se realiza para tener una referencia de la productividad de cada proceso y del tiempo de ejecución.

FT Protección: Se utiliza la tolerancia a fallos para proteger dicha ejecución mediante checkpoint y log de mensaje. Permite comprobar el funcionamiento del checkpoint del árbol de procesos y el log de eventos no deterministas, teniendo en cuenta que no se guarden las comunicaciones internas dentro de cada nodo, y, en cambio, guardar los mensajes que provienen de otros nodos. También se registran los eventos de consultas de comunicaciones cuando estas son positivas. Utilizamos este tipo de ejecución para evaluar el impacto de la sobrecarga anadida por el log y del checkpoint en la productividad y en tiempo de ejecución. Se realiza un checkpoint de los procesos que se ejecutan en el nodo N3 a los 24 segundos.

FT Recuperación: Se utiliza la tolerancia a fallos para proteger la ejecución mediante checkpoint y log de mensaje y se inyecta un fallo en el nodo N3 en el segundo 34, que provoca una caída del árbol de procesos y su reejecución. Este tipo de ejecución con fallos nos permite comprobar el funcionamiento del mecanismo de recuperación del árbol de los procesos, del protector, la reejecución por parte del observador utilizando el log

de mensajes sólo para comunicaciones provenientes de otros nodos, así como utilizar las comunicaciones para las comunicaciones internas. También se comprueba la reproducción de los eventos no deterministas correspondientes a las llegadas de datos en comunicaciones. Por último, se valida la reconfiguración de los protectores y observadores aislando el nodo que ha fallado y enmascarando las comunicaciones de la librería.

Se utiliza un diagrama de productividad que mide el trabajo por unidad de tiempo que realiza cada proceso. Esta métrica permite observar como impactan las sobrecargas en cada proceso. También se mide y se compara el tiempo de ejecución.

Para el gráfico de la productividad se ha agrupado el trabajo realizado por cada proceso durante un intervalo de 5 segundos. El eje X muestra los distintos intervalos y el eje Y, el trabajo realizado durante dicho intervalo. El trabajo de los procesos de mide en ambas aplicaciones por iteraciones. Cabe aclarar que los procesos SPMD realizan un intercambio de los bordes cada 100 iteraciones.

Se realizaron al menos 10 ejecuciones de cada uno de los casos y se comprobó el funcionamiento esperado de protección, detección y recuperación. Se selecciona un caso para cada una de las aplicaciones con un valor medio de tiempo de ejecución y se muestra y explican los resultados obtenidos en el siguiente apartado.

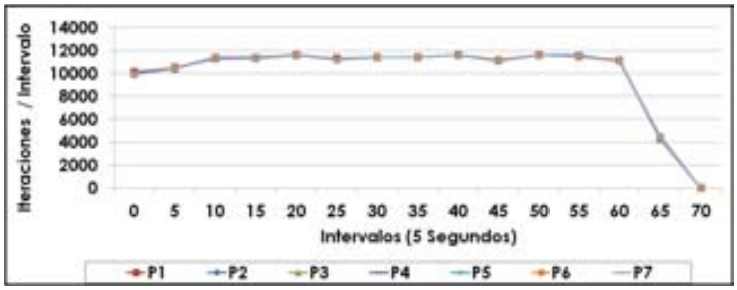
5.4.3. Experimentos

Aplicación SPMD de transferencia de calor

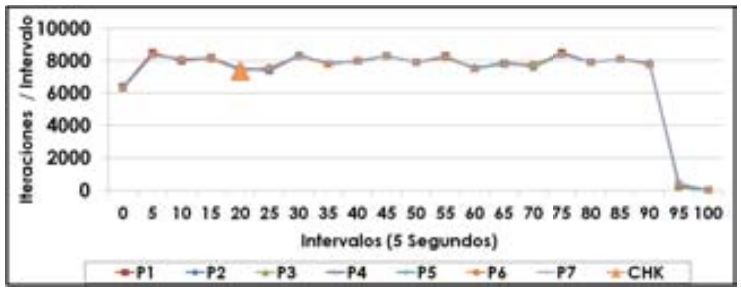
La figura 5.6 muestra el comportamiento de todos los procesos de la aplicación durante las tres ejecuciones realizadas. Comparando la ejecución sin tolerancia a fallos representada en 5.6(a) con la figura 5.6(b) que muestra la ejecución *FT Protección*, vemos como cae la productividad por el efecto de la protección del log que en promedio ronda en las 11.000 iteraciones por cada 5 segundos, cayendo a 8.000. La sobrecarga del protocolo de coordinación y el checkpoint se ve durante el intervalo de 20-25 segundos.

La ejecución con recuperación 5.6(c) muestra como en el momento del fallo de nodo, la productividad de los procesos cae porque se dejan de recibir mensajes de los procesos contiguos. Una vez detectado el fallo del nodo N3 y reiniciado el árbol de procesos en el nodo N2 a partir del checkpoint, se inicia la reejecución de los procesos desde el checkpoint hasta el momento del fallo. En ese momento se alcanza la mayor productividad dado que los mensajes se recuperan del log previamente grabado. Hay una caída posterior causada por las tareas de reconexión.

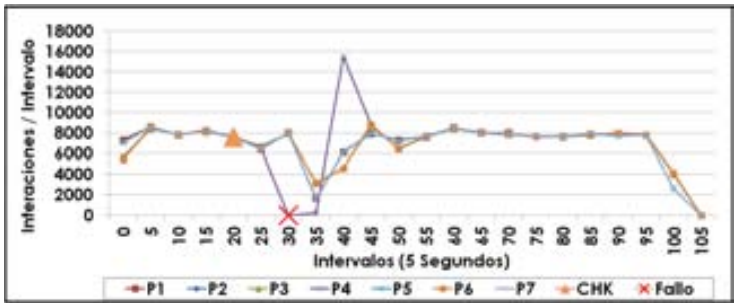
Para analizar en mejor detalle lo que ocurre durante la protección y la recuperación, la figura 5.7 muestra el comportamiento del proceso P4 durante los tres tipos de ejecuciones.



(a) Ejecucion No FT



(b) Ejecucion FT-Proteccion



(c) Ejecucion FT-Recuperacion

Figura 5.6: Productividad de aplicación SPMD en cada ejecución

El aumento en tiempo de ejecución fue de un 43.94 % durante la protección y de 54.55 % en presencia de un fallo.

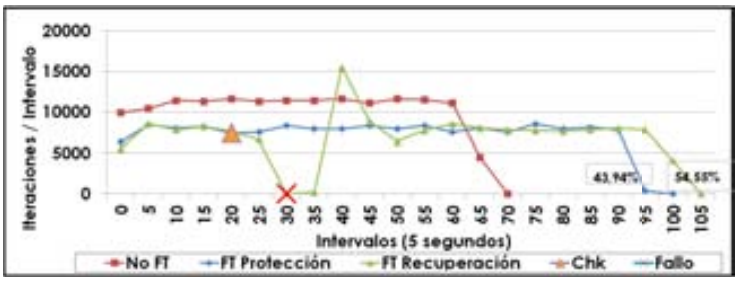


Figura 5.7: Comparación de ejecuciones del proceso P4

Aplicacion Master / Worker

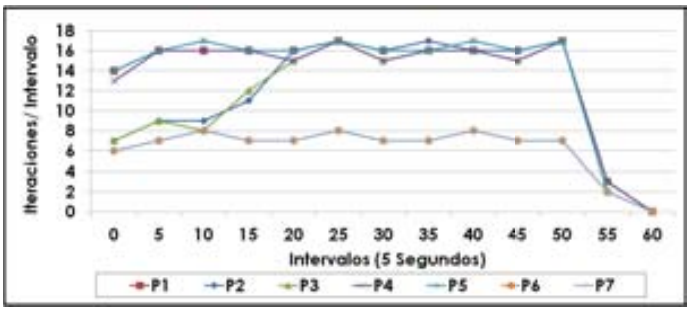
La figura 5.8 muestra el comportamiento de los siete workers de la multiplicación de matrices durante los experimentos. Comparando la ejecución sin tolerancia a fallos representada en 5.8(a) con la figura 5.8(b) que muestra la ejecución *FT Proteccion*, se observa la caída de las iteraciones por intervalo que procesan los workers por el efecto de la protección del log. Las caídas son más abruptas en protección debido a las latencias que sufren los workers añadidas por el log, la coordinación y el checkpoint. La sobrecarga de la protección hace que los procesos necesiten más tiempo para finalizar la ejecución.

La ejecución con recuperación 5.8(c) muestra como en el momento del fallo de nodo, la productividad de todos los workers cae, pero unos segundos después, cuando el master deja de enviarles trabajo a procesar porque está esperando la recuperación de los worker del nodo N3.

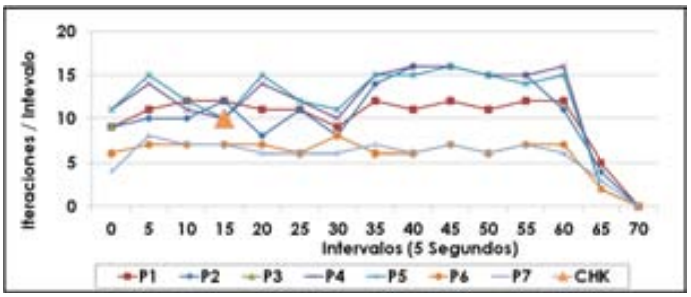
Una vez detectado el fallo del nodo N3 y reiniciado el árbol de procesos en el nodo N2 a partir del checkpoint, se inicia la reejecución de los procesos desde el checkpoint hasta el momento del fallo. En ese momento se alcanza la mayor productividad dado que los mensajes se recuperan del log previamente grabado.

Hay una caída abrupta posterior provocada por las tareas de reconexión. Durante el resto de la ejecución, la productividad cae aún más debido a que el nodo N2 queda saturado con la ejecución de los dos árboles de 4 procesos.

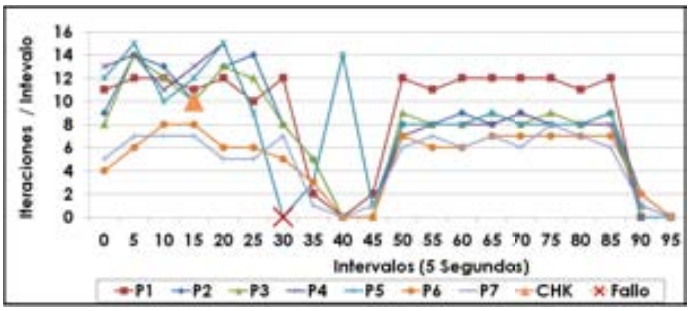
Para analizar en mejor detalle lo que ocurre durante la protección y la recuperación, la figura 5.9 muestra el comportamiento de los proceso P4 y P5 durante los tres tipos de ejecuciones. Dichos procesos son los que se ejecutan en el nodo N3. Se observa la caída de productividad por el log y checkpoint en la diferencia de las iteraciones por intervalo de las líneas rojas y azul. La línea verde, correspondiente a la ejecución con recuperación, cae considerablemente por estar compartiendo nodo con los procesos P2 y P3 del nodo N2.



(a) Ejecucion No FT



(b) Ejecucion FT-Proteccion



(c) Ejecucion FT-Recuperacion

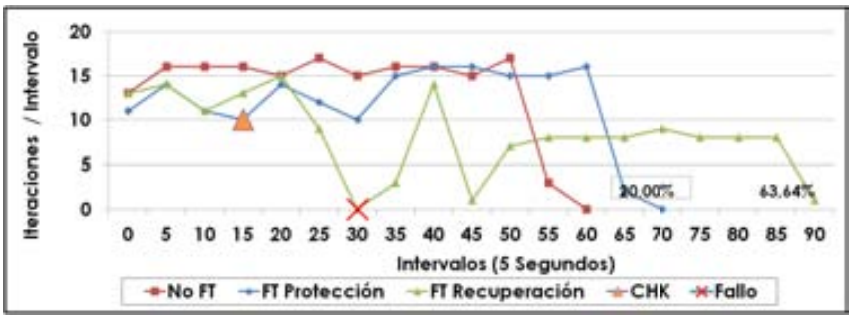
Figura 5.8: Productividad de la aplicación Master/Worker sin tolerancia a fallos, en protección y recuperación

El aumento en tiempo de ejecución fue de un 20 % por el efecto de la protección y de 63.64 % en presencia de un fallo para P4 y de 61.82 para P5.

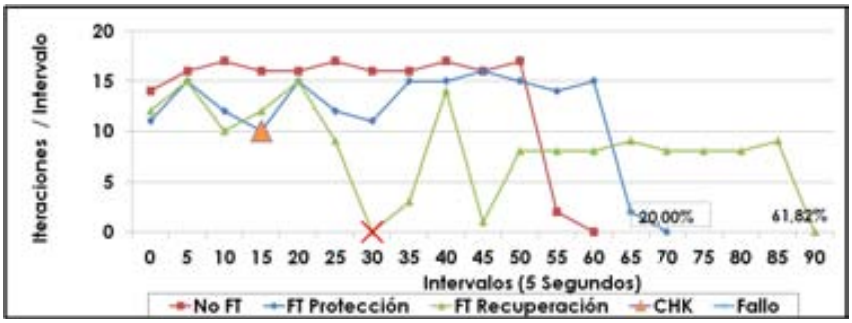
5.5. Conclusiones

Se ha propuesto un modelo de tolerancia a fallos de aplicaciones paralelas de paso de mensaje que permite proteger de fallos de nodos a aplicaciones que utilicen Message Passing Interface MPI.

La arquitectura RADIC gestiona por medio de los modelos de protección de estado de ejecución, detección de fallos, recuperación del estado de ejecución, reconfiguración y



(a) Proceso P4



(b) Proceso P5

Figura 5.9: Comparación de las distintas ejecuciones de los procesos en N3

enmascaramiento de los fallos, la información necesaria en esta capa de socket para llevar a cabo exitosamente las funciones de tolerancia a fallos de nodos en forma transparente y automática.

La validación experimental muestra que se soportan los paradigmas de aplicaciones más utilizados en el ámbito científico, como son Master/Worker y SPMD, y el modelo es de utilidad en la estructura utilizada por una de las librería MPI más importantes como Open-MPI y que se adapta a las arquitecturas de clústeres multicores al utilizar un protocolo semi-coordinado que tiene en cuenta la dependencia de los procesos que se ejecutan en un nodo para no añadir sobrecarga adicional.

Capítulo 6

Conclusiones

Se ha propuesto un modelo de tolerancia a fallos que permite que una aplicación de paso de mensaje continúe y finalice correctamente su ejecución cuando se presenten fallos en los nodos de procesamiento, sin realizar cambios en la aplicación así como tampoco en la librería de paso de mensaje que se está utilizando.

Se ha propuesto un middleware que incorpora todos los mecanismos de tolerancia a fallos: protección, detección, recuperación, reconfiguración y enmascaramiento del fallo, disenado para aplicaciones paralelas de paso de mensaje.

Para ubicar el middleware que realiza la tolerancia a fallos se han analizado las capas de software de los sistemas paralelos, y para garantizar la transparencia se ha elegido integrarlo en el API a nivel de sockets, una capa de sistema a nivel de usuario.

Se ha disenado para ser completamente transparente a las aplicaciones y a las librerías de paso de mensaje, por lo que el sistema puede utilizarse sin necesidad de modificar ni compilar las aplicaciones. Se ha considerado la importancia del funcionamiento distribuido y descentralizado que, entendemos, es una condición deseable en la actualidad, debido a que los computadores tienden a crecer en número de procesadores y las aplicaciones en número de procesos.

Una característica a destacar que diferencia esta investigación con otros sistemas de tolerancia a fallos para aplicaciones paralelas de paso de mensaje, es la capacidad de detección y recuperación automática. Estas propiedades son las que permiten a un centro de procesamiento ejecutar aplicaciones con garantía de finalización a un coste muy bajo, debido a que no se necesita intervenciones manuales por parte de operadores o administradores. Como además es transparente, permite dotar de tolerancia a fallos a aplicaciones cuyo coste de adaptación pueda ser muy caro o incluso imposible cuando no se dispone de los programas fuentes.

El sistema propone un protocolo semi-coordinado para disminuir la sobrecarga de la

tolerancia durante la ejecución libre de fallos cuando se utilizan sistemas multicores. Los componentes de la arquitectura RADIC disenados disponen de toda la información que permiten hacer la coordinación a nivel de nodo, o hacer grupos de coordinación en un mismo nodo. El protocolo de coordinación propuesto es flexible y permite configurarlo adaptándose a distintos sistemas y comportamientos de aplicaciones. La valoración experimental nos permite concluir que constituye una mejor alternativa al protocolo no coordinado para aplicaciones limitadas por comunicaciones, es decir, cuando los tiempos de espera por comunicaciones están en el camino crítico de la ejecución.

Para aplicaciones de paso de mensaje MPI ejecutadas en clústeres multicore se han analizado dos de las librerías más relevantes del mercado y se ha disenado un módulo extendido de intercepción de funciones de socket.

El diseno extendido y la evaluación experimental con aplicaciones MPI permiten concluir que el método es válido para utilizarse en la práctica, con las aplicaciones y librerias actuales. Sin embargo, la capa de socket, es un sitio de alto impacto y sensibilidad que requiere un desarrollo complejo y una gran número de pruebas para garantizar la robustez y fiabilidad de un sistema de este tipo.

La siguiente sección realiza un resumen de las partes de esta tesis y luego se continúa con las propuestas del trabajo futuro de las líneas abiertas de esta tesis.

6.1. Conclusiones finales

Las contribuciones se han presentado y desarrollado a lo largo de esta tesis. La primera de ellas corresponde a la propuesta del *socket seguro* que establece un transporte libre de fallos entre dos procesos. El mecanismo de tolerancia a fallos de comunicaciones a nivel de *socket seguro* permite detectar errores de comunicaciones y localizar una nueva ubicación del proceso remoto. Esta función es soportada por la segunda aportación de esta tesis, que es el diseno de un sistema de tolerancia a fallos transparente y distribuido, que, apoyado en los *sockets seguros*, y en los modelos de la arquitectura RADIC, provee mecanismos en la capa de socket para efectuar las funciones de protección del estado de los procesos, detección de fallos y recuperación de los procesos hasta el momento del fallo.

La tercera contribución es la incorporación de un protocolo semi-coordinado que permite mejorar el rendimiento de la tolerancia a fallos en sistemas multicore. Por último, la última contribución consiste en ampliar las funcionalidades del sistema de tolerancia para que pueda utilizarse con aplicaciones paralelas de paso de mensajes MPI, que es el estándar más utilizado en estos casos, y permite concluir que la estrategia de tolerancia a fallos en capa de socket es factible para las aplicaciones que se utilizan actualmente.

6.2. Trabajos futuros

Los líneas futuras que se abren a partir de esta investigación son:

- Continuar con el desarrollo del prototipo del presente trabajo de modo que constituya una herramienta robusta y fiable, capaz de tolerar fallos de nodos durante la ejecución de aplicaciones paralelas de paso de mensaje de uso actual, en clústeres multicores, y utilizando tanto las librerías Open-MPI como MPICH. Esto permitirá investigar el impacto de la tolerancia en el comportamiento de diferentes tipos de aplicaciones y distinguir si el efecto es causado por la aplicación o por la librería de paso de mensaje que se estuviera utilizando, así como proveer a la comunidad de usuarios de aplicaciones de paso de mensaje, una herramienta robusta para la tolerancia a fallos en capa de sistema.

- Permitir un protección selectiva con respecto al momento en que se inicia y se finaliza dicha protección y con respecto a activarse para determinados componentes. De este modo, los usuarios o administradores puedan determinar qué y cuando iniciar la tolerancia a fallos y así ajustar los costes de la protección a las necesidades.

- Estudiar los cambios que requieren los modelos de tolerancia propuestos en esta tesis para que puedan utilizarse en un entorno cloud, privado o público, que ejecute aplicaciones paralelas de paso de mensaje.

6.3. Lista de Publicaciones

El trabajo presentado en esta tesis ha sido publicado en

1. **H.Meyer, M. Castro, D. Rexachs, and E. Luque. *Propuestas para integrar la arquitectura RADIC de forma transparente*, CACIC 2011 XVII Congreso Argentino de Ciencias de la Computación, pp. 347-356, La Plata, Argentina, Julio 2011.** [64]
   Esta artículo presenta dos propuestas para integrar la arquitectura RADIC en aplicaciones de paso de mensaje. La primera es a nivel de librería de comunicaciones de paso de mensaje y la segunda a nivel de primitiva de comunicaciones de socket.

2. **M. Castro, D. Rexachs, and E. Luque. *Transparent Fault Tolerance Solution at Socket Level Based on RADIC*, In ISPA 2012 - The 10th**

IEEE International Symposium on Parallel and Distributed Processing and Applications, pp. 831-832, Leganés, Madrid, España, Julio 2012. [65]

Es un trabajo tipo *poster* que presenta el trabajo de investigación de integración de RADIC en la capa de socket de comunicaciones. La integración se realiza en forma de middleware que en forma automática y en capa de usuario es capaz de enmascarar fallos de nodos durante la ejecución de una aplicación paralela de paso de mensajes en forma transparente.

3. **M. Castro, D. Rexachs, and E. Luque. *Transparent Fault Tolerance Middleware at User Level*, HPCS 2012 The 2012 International Conference on High Performance Computing & Simulation, pp. 566-572, Madrid, España, Julio 2012.** [56]

   En este trabajo se presenta el diseno de un middleware de tolerancia a fallos en base a los modelos RADIC de protección, recuperación y detección de fallos para conseguir un funcionamiento descentralizado. Se presentan experimentos con dos aplicaciones paralelas de paso de mensaje que siguen diferentes patrones de comunicaciones.

4. **M. Castro, D. Rexachs, and E. Luque. *RADIC-based Message Passing Fault Tolerance System*, ADVCOMP 2012 - The Sixth International Conference on Advanced Engineering Computing and Applications in Sciences, pp.59-64, Barcelona, España, Septiembre 2012** [55]

   En este trabajo se realiza un análisis del diseno de RADIC en la capa de sockets, describiendo las alternativas y las decisiones tomadas para conseguir los objetivos de independencia de librería manteniendo las propiedades de transparencia, distribución y descentralización. Se presenta una evaluación experimental de las sobrecargas del sistema en tiempo y ancho de banda utilizando aplicaciones M/W y SPMD.

5. **M. Castro, D. Rexachs, and E. Luque. *Message Passing Fault Tolerance Design at Socket Level*, XXIII Jornadas de Paralelismo, pp. 276-280, Elche (Alicante), España, Septiembre 2012** [66]

   En este trabajo se presenta el diseno de un middleware de tolerancia a fallos en capa de socket. Se explican los requerimientos, los objetivos de diseno y se detallan los modelos de protección, detección y recuperación. Se presentan experimentos con dos aplicaciones paralelas de paso de mensaje en capa de socket que siguen diferentes patrones de comunicaciones (M/W y SPMD).

6. **M. Castro, D. Rexachs, and E. Luque. *Adding Semi-coordinated Checkpoint to RADIC in Multicore Clusters*, The 19th International Conference for Parallel and Distributed Processing Techniques and Applications PDPTA 2013, to be appeared, Las Vegas, USA, Julio 2013** [67]
   En este trabajo se presenta la propuesta de protocolo semi-coordinado para ser utilizado dentro del middleware de tolerancia a fallos en capa de socket. Se explican los cambios realizados en los modelos de protección y recuperación, y se estudian las sobrecargas de cada protocolo. Se compara el funcionamiento de los protocolos no-coordinado pesimista basado en receptor con el de checkpoint semi-coordinado en dos aplicaciones paralelas de paso de mensaje en capa de socket que siguen diferentes patrones de comunicaciones (M/W y SPMD).

Bibliografía

[1] D. A. Reed, C. da Lu, and C. L. Mendes, Reliability challenges in large systems, *Future Generation Computer Systems*, vol. 22, no. 3, pp. 293 302, 2 2006.

[2] R. Hempel, The mpi standard for message passing, vol. 797, pp. 247 252, 1994. [Online]. Available: http://dx.doi.org/10.1007/3-540-57981-8_126

[3] R. D. Schlichting and F. B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, *ACM Trans.Comput.Syst.*, vol. 1, no. 3, pp. 222 238, aug 1983. [Online]. Available: http://doi.acm.org/10.1145/357369.357371

[4] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, Mpich-v project: A multiprotocol automatic fault-tolerant mpi, *International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 319 333, FAL 2006.

[5] G. Fagg and J. Dongarra, Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world, pp. 346 353, 2000.

[6] A. Duarte, D. Rexachs, and E. Luque, Increasing the cluster availability using radic, in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2006.

[7] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, The design and implementation of checkpoint/restart process fault tolerance for open mpi, in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1 8.

[8] S. Louca, N. Neophytou, A. Lachanas, and P. Evrripidou, Mpi-ft: Portable fault tolerance scheme for mpi, *Parallel Processing Letters*, vol. 10, no. 04, pp. 371 382, 12/01; 2013/04 2000. [Online]. Available: http://dx.doi.org/10.1142/S0129626400000342

[9] R. Batchu, J. P. Neelamegam, Z. Cui, M. Beddhu, A. Skjellum, Y. Dandass, and M. Apte, Mpi/fttm: architecture and taxonomies for fault-tolerant, message-passing

middleware for performance-portable parallel computing, in *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, 2001, pp. 26 33.

[10] R. Batchu, Y. Dandass, A. Skjellum, and M. Beddhu, Mpi/ft: A model-based approach to low-overhead fault tolerant message-passing middleware, *Cluster Computing*, vol. 7, no. 4, pp. 303 315, 10/01 2004. [Online]. Available: http://dx.doi.org/10.1023/B%3ACLUS.0000039491.64560.8a

[11] S. Rao, L. Alvisi, and H. M. Vin, Egida: An extensible toolkit for low-overhead fault-tolerance, *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, Digest of Papers*, pp. 48 55, 1999.

[12] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall, and M. W. Sukalski, Architecture of la-mpi, a network-fault-tolerant mpi, in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, p. 15.

[13] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, The lam/mpi checkpoint/restart framework: System-initiated checkpointing, *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479 493, November 01 2005.

[14] W. Gropp and E. Lusk, Fault tolerance in message passing interface programs, *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363 372, FAL 2004.

[15] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen, Design and implementation of mpich2 over infiniband with rdma support, p. 16, 2004.

[16] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.4BSD operating system.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc, 1996.

[17] R. Gordon, The tcp/ip guide: A comprehensive, illustrated internet protocols reference. *Library Journal*, vol. 131, no. 1, pp. 146 146, JAN 2006.

[18] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, *ACM Comput.Surv.*,

vol. 34, no. 3, pp. 375 408, September 2002. [Online]. Available: http://doi.acm.org/10.1145/568522.568525

[19] L. Fialho, G. Santos, A. Duarte, D. Rexachs, and E. Luque, Challenges and issues of the integration of radic into open mpi, in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 73 83. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_14

[20] G. Santos, A. Duarte, D. Rexachs, and E. Luque, *Providing non-stop service for message-passing based parallel applications with RADIC*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2008, vol. 5168 LNCS.

[21] , *Increasing the performability of computer clusters using RADIC II*. LOS ALAMITOS; 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1264 USA: IEEE COMPUTER SOC, 2008.

[22] A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra, Correlated set coordination in fault tolerant message logging protocols, in *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, ser. Euro-Par 11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 51 64. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033408.2033415

[23] Y. Luo and D. Manivannan, Hope: A hybrid optimistic checkpointing and selective pessimistic message logging protocol for large scale distributed systems, *Future Generation Computer Systems*, vol. 28, no. 8, pp. 1217 1235, 10 2012.

[24] J.-M. Yang, K. F. Li, W.-W. Li, and D.-F. Zhang, Trading off logging overhead and coordinating overhead to achieve efficient rollback recovery, *Concurrency and Computation: Practice and Experience*, vol. 21, no. 6, pp. 819 853, 2009.

[25] Q. Gao, W. Huang, M. J. Koop, and D. K. Panda, Group-based coordinated checkpointing for mpi: A case study on infiniband, in *Parallel Processing, 2007. ICPP 2007. International Conference on*, 2007, pp. 47 47.

[26] C. D. V. Rao and M. M. Naidu, A new, efficient coordinated checkpointing protocol combined with selective sender-based message logging, in *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, 2008, pp. 444 447.

[27] J. C. Y. Ho, C.-L. Wang, and F. C. M. Lau, Scalable group-based checkpoint/restart for large-scale message-passing systems, in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1 12.

[28] B. Schroeder and G. A. Gibson, A large-scale study of failures in high-performance computing systems, *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337 350, OCT-DEC 2010.

[29] K. M. Chandy and C. V. Ramamoorthy, Rollback and recovery strategies for computer programs, *Computers, IEEE Transactions on*, vol. C-21, no. 6, pp. 546 556, 1972.

[30] E. N. Elnozahy and J. S. Plank, Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery, *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 97 108, 2004.

[31] F. Cappello, Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities, *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212 226, August 01 2009.

[32] S. Chakravorty and L. V. Kale, A fault tolerant protocol for massively parallel systems, in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, p. 212.

[33] L. V. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy, The charm parallel programming language and system: Part i description of language features, *Parallel Programming Laboratory Technical Report #95-02*, 1994.

[34] C. Huang, O. Lawlor, and L. V. KalA©, Adaptive mpi, vol. 2958, pp. 306 322, 2004. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24644-2_20

[35] K.-C. Li, C.-H. Hsu, L. T. Yang, J. Dongarra, and H. Zima, *Handbook of Research on Scalable Computing Technologies*. IGI Global, 2010. [Online]. Available: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-60566-661-7

[36] V. Cristea, C. Dobre, C. Stratan, F. Pop, and A. Costan, *Large-Scale Distributed Computing and Applications: Models and Trends*. IGI Global, 2010. [Online]. Available: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-61520-703-9

[37] K. Hwang, G. Fox, and J. J. Dongarra, *Distributed and Cloud Computing*. USA: Morgan Kaufmann Publishers Inc, 2012.

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

[39] K. S and Rajaraman, A survey of checkpointing algorithms for parallel and distributed computers, 2000.

[40] K. M. Chandy, Distributed snapshots: Determining global states of distributed systems, *ACM Transactions on Computer Systems*, vol. 3, pp. 63 75, 1985.

[41] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi, in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC 06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188587

[42] J. Ansel, K. Aryay, and G. Coopermany, Dmtcp: Transparent checkpointing for cluster computations and the desktop, in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1 12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1586640.1587579

[43] R. Koo and S. Toueg, Checkpointing and rollback-recovery for distributed systems, *Software Engineering, IEEE Transactions on*, vol. SE-13, no. 1, pp. 23 31, 1987.

[44] P. H. Hargrove and J. C. Duell, Berkeley lab checkpoint/restart (blcr) for linux clusters, *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006. [Online]. Available: http://stacks.iop.org/1742-6596/46/i=1/a=067

[45] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, Current practice and a direction forward in checkpoint/restart implementations for fault tolerance, in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005, p. 8 pp.

[46] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Commun.ACM*, vol. 21, no. 7, pp. 558 565, jul 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[47] A. Bouteiller, G. Bosilca, and J. Dongarra, *Retrospect: Deterministic replay of MPI applications for interactive distributed debugging*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2007, vol. 4757 LNCS.

[48] A. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello, Coordinated checkpoint versus message log for fault tolerant mpi, in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, 2003, pp. 242 250.

[49] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra, Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure, recovery, in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 2009, pp. 1 9.

[50] S. Rao, L. Alvisi, and H. M. Vin, The cost of recovery in message logging protocols, *IEEE Trans.on Knowl.and Data Eng.*, vol. 12, no. 2, pp. 160 173, mar 2000. [Online]. Available: http://dx.doi.org/10.1109/69.842260

[51] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, Improved message logging versus improved coordinated checkpointing for fault tolerant mpi, in *Cluster Computing, 2004 IEEE International Conference on*, 2004, pp. 115 124.

[52] A. Bouteiller, G. Bosilca, and J. Dongarra, Redesigning the message logging model for high performance, *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196 2211, 2010.

[53] H. S. Paul, A. Gupta, and A. Sharma, Finding a suitable checkpoint and recovery protocol for a distributed application, *Journal of Parallel and Distributed Computing*, vol. 66, no. 5, pp. 732 749, 5 2006.

[54] L. M. Silva and J. G. Silva, The performance of coordinated and independent checkpointing, *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pp. 280 284, 1999.

[55] M. Castro, D. Rexachs, and E. Luque, Radic-based message passing fault tolerance system, in *ADVCOMP 2012, The Sixth International Conference on Advanced Engineering Computing and Applications in Sciences*, 2012, pp. 59 64.

[56] , Transparent fault tolerance middleware at user level, in *HPCS'12*, 2012, pp. 566 572.

[57] G. R. M. Martin, P. Gonzalez, J. Tourino, and R. Doallo, Cppc: a compiler-assisted tool for portable checkpointing of message-passing applications, *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749 766, 2010.

[58] V. D. Florio, *Application-Layer Fault-Tolerance Protocols*. IGI Global, 2009. [Online]. Available: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-60566-182-7

[59] A. Agbaria and R. Friedman, Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations, pp. 227 236, 2003.

[60] MPI, MPI Forum. [Online]. Available: http://www.mpi-forum.org/

[61] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems, *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 119, 2007.

[62] M. Fertre and C. Morin, Extending a cluster ssi os for transparently checkpointing message-passing parallel applications, in *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks, I-SPAN*, vol. 2005, 2005, pp. 364 369.

[63] V. C. Zandy and B. P. Miller, Reliable network connections, in *Proceedings of the 8th annual international conference on Mobile computing and networking*, ser. MobiCom 02. New York, NY, USA: ACM, 2002, pp. 95 106. [Online]. Available: http://doi.acm.org/10.1145/570645.570657

[64] H. Meyer, M. Castro, D. Rexachs, and E. Luque, Propuestas para integrar la arquitectura radic de forma transparente, in *CACIC 2011 XVII Congreso Argentino de Ciencias de la Computación*, 2011, pp. 347 356.

[65] M. Castro, D. Rexachs, and E. Luque, Transparent fault tolerance solution at socket level based on radic, in *ISPA'12*, 2012, pp. 831 832.

[66] , Message passing fault tolerance design at socket level, in *XXIII Jornadas de Paralelismo*, 2012, pp. 555 560.

[67] , Adding semi-coordinated checkpoint to radic in multicore clusters, in *PDPTA 2013*, 2013.