View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Difusión de la Creación Intelectual

Recuperando prestaciones en clusters tras ocurrencia de fallos utilizando RADIC

Guna A. Santos, A. Duarte, D. Rexachs, E. Luque *
Departamento de Arquitectura de Computadores y Sistemas Operativos, Universidad Autónoma de Barcelona
Bellaterra, Barcelona 08193, España
{guna, angelo}@aomail.uab.es, {dolores.rexachs, emilio.luque}@uab.es

Abstract

After a fault recovering, the reduction of the planned nodes number and the existence of unplanned process node sharing, leads to application performance lost. This work presents a proposal to minimize the performance lost in rollback-recovery based fault tolerant parallel systems, after a fault occurrence, when the parallel machine reconfigure itself with one node less, affecting the application total execution time. In order to restore the performance, we propose a solution that extends the RADIC architecture: the possibility of, during the application execution, allow the faulty nodes replacement or to have process free spare nodes that may or not be started with the application, in order to under a node failure assumes the process that was in execution on the faulty node.

Keywords: Fault Tolerance, Cluster, Parallel Systems, Performance, MPI, RADIC.

Resumen

Tras la recuperación de un fallo, las aplicaciones pierden prestaciones debido, en gran parte, a que el número planificado de nodos ha disminuido y de la pérdida que provoca la existencia no planificada de procesos compartiendo el mismo nodo. Este trabajo presenta una propuesta para mitigar las pérdidas de prestaciones en sistemas paralelos con tolerancia a fallos basados en *rollback-recovery*, después de un fallo, donde la máquina paralela queda reconfigurada con un nodo menos, con la consiguiente repercusión en el tiempo de ejecución de la aplicación. Proponemos para recuperar las prestaciones, una solución que extiende la arquitectura RADIC: la posibilidad de permitir, durante la ejecución de la aplicación, el reemplazo de nodos que han fallado o disponer de nodos extras que pueden ser iniciados con la aplicación, pero sin procesos de la aplicación activos, de forma que cuando falle un nodo pase a ejecutar los procesos en dicho nodo

Palabras claves: Tolerancia a fallos, Cluster, Sistemas Paralelos, Prestaciones, MPI, RADIC.

1 INTRODUCCIÓN

En los últimos años, ha ido creciendo la utilización de los clusters de computadores para solucionar problemas que exigen cómputo con altas prestaciones, o para aplicaciones que requieren alta disponibilidad. Además, la utilización de aplicaciones paralelas se ha popularizado con el surgimiento de los clusters de *workstations* (COW). En este escenario, *Message Passing Interface*

* Este trabajo es soportado por el MEyC-España bajo contracto TIN 2004-03388

(MPI) se presenta hoy, como uno de los estándar *de facto,* para comunicaciones en aplicaciones paralelas, siendo utilizado con éxito en muchas áreas y teniendo muchas implementaciones como MPICH[8] y LAN/MPI[4].

Sin embargo, la especificación de MPI define un comportamiento basado en la semántica de *fail stop*, eso significa que en caso de la ocurrencia de fallo en algún nodo, toda la aplicación se parará. Así, el aspecto de garantizar la finalización correcta de una aplicación se convierte en un tema de gran importancia. Por eso, se ha dedicado mucho esfuerzo para dotar de niveles de tolerancia a fallos a sistemas paralelos basados en MPI, utilizando diferentes estrategias. Tenemos ejemplos como MPICH-V[3], Starfish[1], MPI/FT[2], Égida[10] entre otros.

Por otro lado, uno de los objetivos de los sistemas basados en *rollback-recovery* es garantizar la finalización correcta de la aplicación, pero analizando los resultados presentados vemos que existirán pérdidas en las prestaciones de una aplicación tras la ocurrencia de un fallo y su recuperación. Estas pérdidas son causadas por distintos factores: a) el *overhead* causado por la necesidad de introducir redundancia de la tolerancia a fallos (fases de prevención y detección) [11], b) el proceso de diagnóstico y de recuperación/re-configuración cuando existe un fallo, también tiene un coste de tiempo [11]; c) pero una cuestión que consideramos crucial es el hecho de que tras la recuperación, el cluster sigue ejecutando con un nodo menos y que muchos sistemas de tolerancia a fallos mantienen constante el número de procesos, por lo tanto, el proceso del nodo que ha fallado migra a un nodo ya utilizado y pasa a compartir su capacidad de procesamiento.

Redundant Array of Distributed Independent Checkpoints (RADIC)[5] es una arquitectura para tolerancia a fallos propuesta por nuestro grupo. RADIC propone una arquitectura basada en la técnica de *rollback-recovery* que provee a los clusters tolerancia a fallos transparente al usuario y al administrador, con escalabilidad, e independiente de elementos centrales. Manejando los propios recursos del cluster, RADIC crea una capa que aísla las aplicaciones de los fallos que pueden ocurrir, utilizando para eso dos componentes básicos llamados *Observadores* y *Protectores*.

En general, los sistemas paralelos estan diseñados con una planificación que logra satisfacer un umbral de prestaciones. En la mayoría de los sistemas basados en *rollback-recovery*, tras la recuperación de un fallo ocurre un cambio en la planificación original, y como consecuencia, las prestaciones planteadas inicialmente, quedan afectadas por este cambio. En sistemas que trabajan con restricciones temporales o especificaciones de tiempo, es tan importante finalizar la aplicación con resultados correctos como lo es concluirla antes del límite de tiempo esperado. En los sistemas que necesitan "alta disponibilidad" es importante tener dispositivos que permitan el reemplazo de nodos sin necesidad de parar las aplicaciones ya sea por mantenimiento o por prevención. En base a las necesidades que plantean muchas aplicaciones podemos considerar que las pérdidas de prestaciones en clusters tras la recuperación/re-configuración de fallos son un factor importante en el momento de buscar soluciones tolerantes a fallos y es conveniente disponer además de mecanismos que garanticen el mantenimiento de dichas prestaciones.

En este artículo, nosotros proponemos una solución que extiende la arquitectura RADIC, dotándola de elementos que permiten el restablecimiento de la planificación planteada por la aplicación. Es decir, permite volver a un sistema con las prestaciones existentes antes de la ocurrencia del fallo, en el que sólo se ha introducido el *overhead* para el diagnóstico del fallo, la recuperación de dicho

fallo y la re-configuración del sistema. Esta solución presenta dos formas para lograr la restauración del número de nodos inicial, permitir re-incorporar un nodo para suplir al nodo que ha fallado de forma que el cluster vuelva a disponer del mismo número de nodos, una vez que el nodo que ha fallado haya sido reemplazado o reparado, o si su fallo ha sido una ocurrencia transitoria volver a recuperarlo, disponiendo después de un tiempo de un sistema con el mismo número de nodos con el que empezó la ejecución. Otra posibilidad es disponer de nodos extras que puedan ejecutar los procesos asignados a un nodo cuando dicho nodo falla.

A continuación, el artículo está organizado de la siguiente forma: En la sección 2, presentamos la arquitectura RADIC y describimos el prototipo implementado. La sección 3 aborda el problema de las pérdidas de prestaciones tras la recuperación de un fallo. La sección 4 presenta la solución propuesta que permite recuperar la planificación original de la aplicación y por tanto disponer de un sistema que puede dar las prestaciones iniciales. Finalmente, en la sección 5 presentamos nuestras conclusiones y trabajos futuros.

2 ARQUITECTURA RADIC

RADIC es una arquitectura que provee tolerancia a fallos a sistemas paralelos que utilizan paso de mensajes. RADIC, proporciona una arquitectura paralela con tolerancia a fallos basada en *rollback-recovery* y *log* de mensaje pesimista [6], siendo una arquitectura escalable y transparente. Dos grupos de procesos distribuidos trabajan en colaboración para desempeñar las tareas que garantizan la finalización correcta de una aplicación. Como podemos ver en la figura 1, estos procesos hacen que RADIC funcione como una capa que aísla una aplicación de los fallos en el cluster, volviéndola libre de fallos.

Figura 1: Capas de una configuración de cluster utilizando RADIC

Los dos grupos de procesos llamados *Observadores* y *Protectores* son ejecutados en los mismos nodos en los que la aplicación paralela se está ejecutando. El trabajo de los *Protectores* y los *Observadores* propician a la aplicación una visión de un cluster virtual libre de fallos. Son distribuidos por todo el cluster y la cantidad de *Observadores* y *Protectores* en ejecución dependen respectivamente del número de procesos de aplicación y del número de nodos del cluster. Cada *Observador* está relacionado con un *Protector* en otro nodo. Los *Protectores* pueden relacionarse con varios *Observadores*. En la figura 2 vemos un ejemplo de una planificación simple de RADIC con los procesos (0..3), los *Observadores* (O_0..O_3) y los *Protectores* (P_0..P_3). A continuación, explicamos el papel de cada uno de ellos.

2.1 *Observadores*

Los procesos *Observadores* están asociados a cada proceso de la aplicación existente y son responsables de: a) gestionar la comunicación de paso de mensaje entre los procesos, b) mantener el *log* de mensajes recibidos y c) regularmente hacer los *checkpoints* del proceso al cual está asociado y transmitirlos al protector.

Los *Observadores* se comunican entre ellos y con los *Protectores*. En cada evento no determinista, como por ejemplo, una transmisión de un mensaje, el *Observador* del receptor se encarga de enviar una copia de éste a su respectivo *Protector*. De esta misma forma, los *Observadores* a cada cierto tiempo salvan el estado del proceso de aplicación (*checkpoint*) y también lo envían a su respectivo *Protector*.

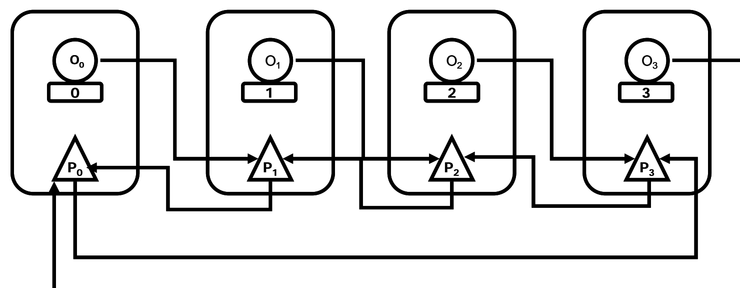

Figura 2: Planificación RADIC simple con 4 nodos y un proceso por nodo

2.2 *Protectores*

La detección, diagnóstico de fallos y la creación de un dispositivo de almacenamiento estable y totalmente distribuido son las actividades ejecutadas por los *Protectores*. En cada nodo del cluster es ejecutado un proceso *Protector*.

A través de un protocolo de *watchdog* y *heartbeat*, los *Protectores* se comunican entre si formando una cadena con el objetivo de detectar fallos en el cluster. En RADIC, se considera un fallo cuando se pierde la comunicación con un nodo. Cuando un fallo es detectado y diagnosticado, los *Protectores* inician el proceso de recuperación.

Tal y como hemos visto, otra actividad de los *Protectores* es recibir y almacenar los *checkpoints* y *log* de mensajes de los *Observadores* relacionados. Con eso, se crea un almacenamiento fiable y totalmente distribuido entre los nodos del cluster, que es la llave para la escalabilidad de la arquitectura.

2.3 RADICMPI

Con la intención de comprobar la funcionalidad y la eficacia de la arquitectura RADIC, se ha desarrollado un prototipo llamado RADICMPI. El prototipo implementa el estándar de paso de mensajes MPI utilizando los conceptos de RADIC ya explicados. Por el momento, RADICMPI incorpora un subconjunto de las funciones de MPI (MPI_Init, MPI_Finalize, MPI_Send, MPI_Recv, MPI_Sendrecv, MPI_Comm_Rank, MPI_Get_processor_name, MPI_Wtime y MPI_Type_size) y estamos desarrollando el subconjunto de las funciones no bloqueadoras de MPI con el objetivo de dejar el prototipo más completo en relación a la especificación MPI.

RADICMPI es una implementación orientada a objetos, *multithreaded* optimizada con la existencia de un pool de *threads* y utiliza la librería *Berkeley Labs Checkpoint/Restart* (BLCR v0.4.2) [7] para sacar los *checkpoints* de los procesos. Está implementada para sistemas operativos Linux *Kernel* 2.4.22 y el protocolo de red es el TCP/IP.

3 EFECTOS DE LOS FALLOS EN LAS PRESTACIONES

RADIC, garantiza, con un cierto coste, la finalización de la aplicación de forma correcta a pesar que ocurra el fallo y perdida de nodos. Basándose para eso, en aislar los nodos que fallan y acabar la ejecución con menos nodos, pero manteniendo la cantidad de procesos, lo que provoca, dependiendo del momento del fallo y del desbalanceo producido, gran repercusión en el tiempo de ejecución de la aplicación

En el proceso de recuperación de un fallo, las soluciones que utilizan la técnica de *rollback-recovery* se recuperan a partir del estado del proceso almacenado más reciente, o sea, su último *checkpoint* y el *log* de sus mensajes almacenados en un repositorio fiable. En seguida, utilizando el *checkpoint* y el *log*, re-creará el proceso en uno de los nodos existentes en el cluster para que este pueda proseguir desde un punto anterior al fallo. Durante las fases de recuperación y re-configuración, los procesos de la aplicación que intenten comunicarse con el proceso que está siendo recuperado, tendrán que esperar. Esto, por supuesto, ya es otro factor que se añade al coste de tiempo en la ejecución del programa.

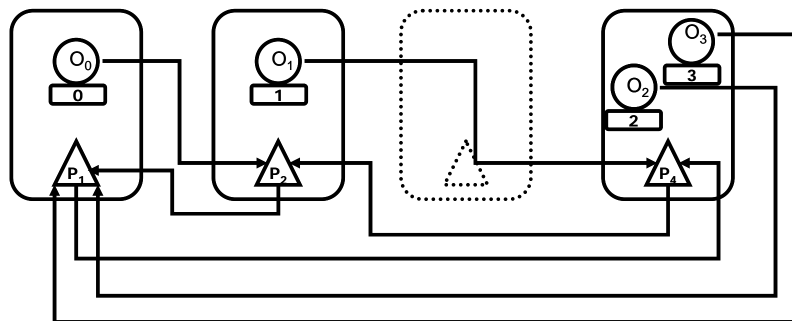

Figura 3: Planificación RADIC tras la recuperación de un fallo de un nodo

Como vimos, tras la recuperación de un fallo, se ha cambiado la planificación planteada originalmente del sistema, en la figura 3 podemos ver un ejemplo, utilizando RADIC, de como quedaría la planificación después del fallo de un nodo mostrado en la figura 2, una vez se ha realizado la recuperación. Como se puede observar, ahora tenemos los procesos 2 y 3, y sus respectivos *Observadores* compartiendo el mismo nodo. En esta situación, tenemos 2 factores que contribuyen fuertemente para que la aplicación siga el resto de su ejecución con unas pérdidas considerables de sus prestaciones: el cluster sigue su ejecución con un nodo menos y, de forma no planificada, hay más de 1 proceso compartiendo el mismo nodo.

Está claro que una aplicación que había sido planificada para una cierta distribución de procesos en los nodos de un cluster, no tendrá las mismas prestaciones ejecutándose con uno o más nodos de menos. Puede que en un cluster con millares de nodos esto no sea muy perceptible, pero en estos clusters, hay más probabilidad de que estos fallos ocurran con más frecuencia, y así las pérdidas

siguen sumándose a lo largo del tiempo hasta un punto crítico.

Por otro lado, un nodo del sistema pasará a tener más de un proceso en ejecución. Por tanto, pasan a compartir la capacidad de cómputo de este nodo, por tanto estos procesos seguramente tendrán una ejecución más lenta, tardando más en finalizar sus tareas. Además de este retraso, como estos pueden estar comunicándose con otros procesos en el cluster, la lentitud de su ejecución se reflejará en sus comunicaciones, propagando así sus retrasos a otros procesos en el cluster y estos por su turno pueden también provocar retrasos en otros procesos y así por delante.

En la tabla 1, mostramos los resultados de ejecución de un programa MPI para una multiplicación de matrices de tamaño 3000x3000 utilizando el paradigma *master-worker*. Hicimos experimentos con distintos contextos, variando entre 8 y 12 la cantidad de procesos. En el primero ejecutamos la aplicación sin tolerancia a fallos. Después con tolerancia haciendo *checkpoint* cada 120 s, pero sin inyectar fallos, para servir de base comparativa para los otros resultados. A continuación, inyectamos un fallo en un nodo a 50% de la ejecución total, teniendo así mitad de la ejecución con la cantidad total de nodos y mitad con 1 nodo compartido entre 2 *workers*. Por fin, para tenernos otra base comparativa, hicimos la ejecución completa con dos *workers* compartiendo 1 nodo el 100% del tiempo.

Tabla 1. Resultado de ejecución de programa de multiplicación de matrices en distintos contextos.

Procesos	Sin Tolerancia	Sin fallos Nodos=N	Con Fallo a 50%	Sin fallos Nodos=N-1	Overhead con fallos	Overhead con N-1
12	250 s	341 s	519 s	611 s	52,20%	79,20%
8	445 s	590 s	940 s	947 s	59,32%	60,50%

Como podemos observar, la ejecución tras la recuperación de un fallo está muy penalizada llegando a más de 50% del tiempo de ejecución, y de acuerdo con los resultados de la ejecución total con 1 nodo menos podemos notar que este hecho es el principal responsable de parte de las pérdidas cuando ocurre un fallo, pues tanto la carga normal de cómputo de los procesos está compartida, como las actividades de tolerancia a fallos, como por ejemplo el *Protector* de este nodo tiene que hacer dos *checkpoints* y de forma secuencial. Las pérdidas podrían ser aún mayores en el caso de que el fallo hubiese ocurrido cerca del inicio de la ejecución, el tiempo total de la ejecución estaría entonces más cerca de la ejecución con 2 *workers* compartiendo un nodo. El retraso de la aplicación después de la recuperación, es muy dependiente de factores como el patrón de comunicación, balanceo da carga, y memoria disponible.

El tiempo total de ejecución (T_e) de un sistema tras la recuperación del fallo puede ser representado con la ecuación siguiente, donde consideramos T_n^p el tiempo de ejecución con p proceso y n nodos, T_r el tiempo gastado en el proceso de recuperación del proceso, T_{n-1}^p el tiempo de ejecución del sistema consumido con un nodo menos y la misma cantidad de procesos, y siendo α la fracción ejecutada de la aplicación hasta el momento del fallo, variando de 0 a 1.

$$T_e = \alpha.T_n^p + T_r + (1-\alpha).T_{n-1}^p \text{ [Ecu. 1]}$$

Los datos de la tabla 1, muestran que una ejecución con p procesos y "*n-1*" nodos puede tener, dependiendo de la aplicación, un *overhead* de hasta el 79%. Así, vemos que dependiendo del momento del fallo (α) el aumento en el tiempo de ejecución podrá influenciar de forma diferente, influye más en el caso de que α represente un valor cerca del inicio (<0,5), provocando así impacto mayor en las pérdidas de prestaciones.

En ciertos tipos de aplicación estos retrasos pueden ser un factor que torne inviable su uso. Por otro lado, en el mundo de la programación paralela, las aplicaciones son planteadas, buscando una planificación que proporcione las mejores prestaciones.

Algunas aplicaciones con gran necesidad de cómputo, como por ejemplo las aplicaciones de búsqueda de imágenes, obtienen mejor precisión cuanto más grande son las prestaciones del sistema. En estas aplicaciones, el retraso puede generar datos que todavía no tengan la calidad esperada, por ejemplo, el análisis de la imagen puede ser incompleto, si es finalizado en el tiempo establecido como máximo y puede obtener una imagen que no represente el mejor resultado de la búsqueda, o el mismo resultado que si no se hubieran introducidos estos retrasos no planificados.

Otros tipos de aplicaciones que también pueden ser muy afectadas son las que llamamos en tiempo real, que necesitan producir un resultado en un determinado tiempo previamente definido como *deadline*. En estas aplicaciones, el retraso puede significar la inutilidad de la información generada, como por ejemplo en sistemas de predicción meteorológica, en los que si la predicción no puede ser hecha con la debida antelación, ya no tiene sentido.

También podemos citar las aplicaciones que necesitan alta disponibilidad, que buscan siempre dar resultados en un mínimo de tiempo. Tenemos como ejemplo, aplicaciones financieras (banco, crédito, etc.). Estas aplicaciones son planificadas para contestar peticiones en un determinado espacio de tiempo. Si ocurren retrasos, posiblemente ocasionará pérdidas financieras o de otro tipo.

4 RECUPERANDO EL NÚMERO DE NODOS

Como mostramos, el coste que pagamos por mantener la tolerancia a fallos en un sistema paralelo puede significar que algunas aplicaciones no produzcan los resultados de la forma esperada. Vimos también que gran parte de este coste, o sea, la pérdida de prestaciones, proviene del hecho que la distribución de procesos en los nodos del cluster se cambia en la re-configuración, al final del proceso de recuperación de un fallo, quedando con menos nodos y el mismo número de procesos. Además, también hay pérdidas inherentes al coste de la propia tolerancia a fallos, existen muchos trabajos de investigación centrados en minimizar este factor. Nosotros hemos focalizado nuestra investigación en la recuperación del número de nodos para mitigar el coste generado por el cambio de la distribución de procesos.

Proponemos una solución que restaura o mantiene la distribución de los procesos de la aplicación entre los nodos en la distribución existente antes del fallo, con el objetivo de recuperar las

prestaciones que el sistema poseía anteriormente. Nuestra propuesta tiene dos abordajes: posibilitar la adición de nodos extras o permitir el reemplazo de un nodo que haya fallado. Explicaremos cada uno de estos casos.

4.1 Añadiendo Nodos Extras (*Spare Nodes*)

Una de las formas más eficientes de mantener la planificación de una aplicación y sus prestaciones tras la recuperación de un fallo, es tener *spare nodes* para que, ocurra un fallo, el proceso del nodo que ha fallado se recupere en este nodo extra, manteniendo así la misma cantidad de nodos en ejecución y no teniendo ningún proceso compartiendo nodos..

En esta primera opción, la aplicación tiene en su mapa de nodos los cuales servirán como *spare nodes*, estos nodos solamente ejecutarán los procesos *Protectores*. Todos los *Protectores* tendrán información de la existencia de los *spare nodes*. En el momento del fallo, cuyo diagrama de flujo de ejecución podemos ver en la Figura 4a, el *Protector* del nodo que ha fallado buscará un *spare node* y en el caso de que éste no posea ya ningún proceso ejecutando en su nodo, le enviará el *checkpoint* y el *log* de mensajes que tiene guardados. El *Protector* que está en el *spare node* seleccionado, recibe esta información y activa el proceso de recuperación. Al finalizar, este *Protector* se unirá a la cadena de *Protectores* existente. Si no quedan mas *spare nodes* libres, el proceso se re-arrancará en el nodo del protector

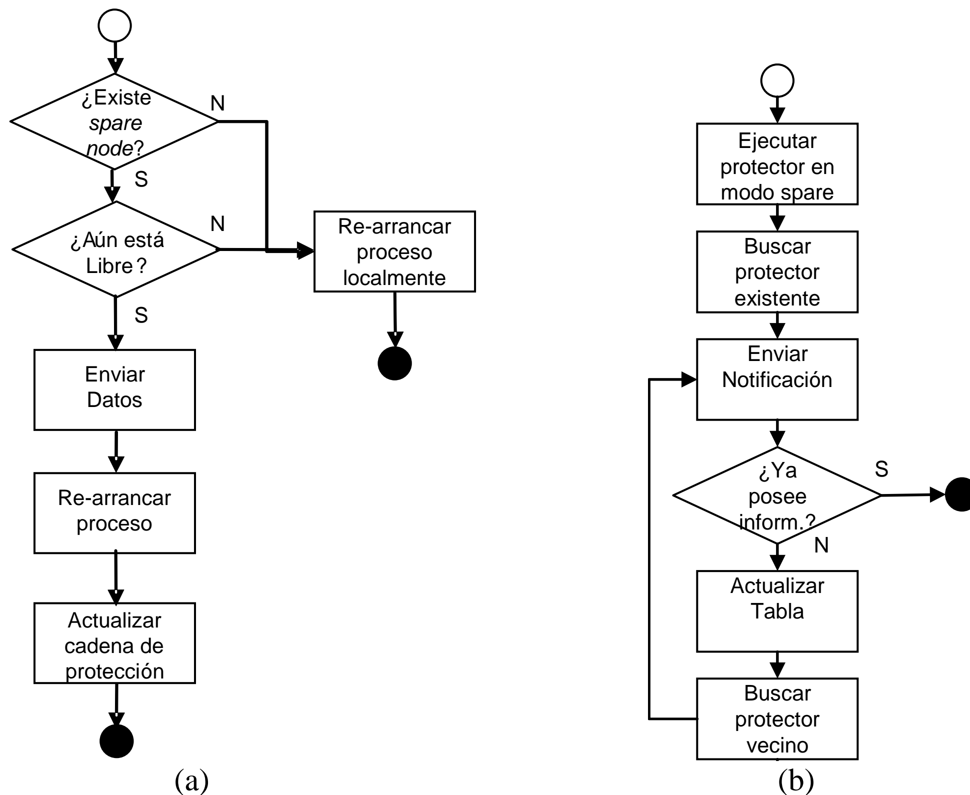

Figura 4: Flujo de ejecución de: (a) recuperación con presencia de *spare node* y (b) añadir *spare nodes* después de iniciada la aplicación.

En la otra opción, representada en el diagrama de flujo de la Figura 4b, tenemos ya una aplicación en ejecución en el cluster y queremos añadir nuevos nodos extras sin necesidad de parar o actualizar la aplicación. La propuesta es hacer que un nodo nuevo, ejecute un proceso *Protector*, que

llamaremos P_e en modo *spare node*. Así el *Protector* P_e irá a buscar un *Protector* vecino entre los nodos existentes de la aplicación (P_x) que informará de su existencia. El *Protector* P_x actualizará su tabla de nodos y *Protectores*, con la información del *spare node* y empezará un procedimiento de *broadcast*, utilizando la técnica de *message fowarding*[9], que notificará también a los otros nodos la existencia del *spare node*. La técnica de *message fowarding* consiste en que cada *Protector* en un nodo recibe la información y después de actualizar su tabla, la pasa adelante, de esta forma garantizamos la escalabilidad de RADIC, ya que no cargamos la red con un *broadcast* normal. Para realizar el *message fowarding*, aprovechamos la estructura ya existente de *watchdog/hearbeat*. El proceso de recuperación es el mismo de la opción anterior.

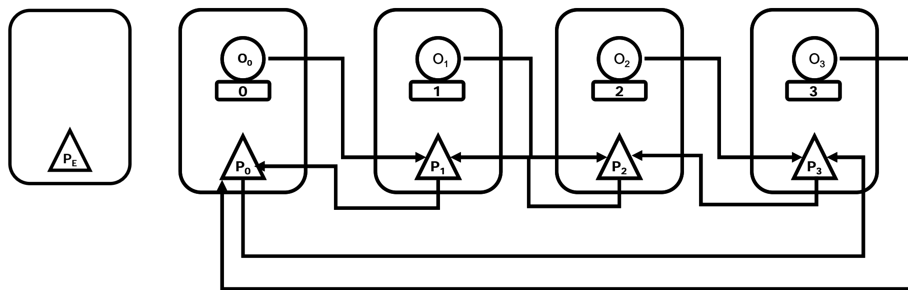

Figura 5: Distribución de nodos en RADIC con la existencia del *spare node.*

En ambos casos, la distribución de procesos en los nodos del cluster con el *spare node* es mostrada en la figura 5. En ella observamos que el *spare node* todavía no forma parte de la cadena de protección, para que así no carguemos el sistema en detectar un fallo de un nodo que no está haciendo cómputo. Como decimos, en el momento de la recuperación, el *Protector* del nodo que ha fallado, tiene la información que existe un *spare node* y la comprueba intentando comunicarse con él y descubriendo si éste sigue sin ejecutar procesos de la aplicación. En este momento el *spare node* pasa a formar parte de la cadena de protección.

En la ecuación siguiente, similar a la ecuación 1, pero teniendo en cuenta ahora el tiempo de ejecución total con fallos y disponiendo de *spare node*. La variable T_d representa el tiempo de enviar los datos necesarios para la recuperación (*checkpoint* y *log* de mensajes) y la conexión a la cadena de *Protectores*, las otras siguen teniendo mismo significado.

$$T_e = T_n^p + T_d + T_r \quad \text{[Ecu. 2]}$$

Se puede notar que esta ecuación no tiene una dependencia del momento donde ocurre el fallo manteniendo el mismo tiempo de ejecución anterior, pero añadiendo un cierto tiempo de retraso que es el envío de los datos hasta el *spare node.*

4.2 Reemplazando Nodos

En algunos casos, no es posible tener nodos extras, sino que es necesario usarlos todos en la planificación del trabajo del cluster, sea por causa de restricciones de coste, o por alguna política adoptada. En estos casos lo que proponemos es un mecanismo que permita que un nodo que haya fallado pueda ser reemplazado, o bien por otro nodo, o bien por el mismo, tras ser reparado. Este

mecanismo se adapta también a los fallos temporales o transitorios. Para la implementación, siguiendo la arquitectura RADIC, este mecanismo debe ser lo más transparente posible.

En este enfoque, el nodo que se va añadir tras un fallo, ejecuta un *Protector* en modo de adición, que es una nueva funcionalidad implementada. En seguida, este *Protector* se comunica con un *Protector* de algún nodo existente en el sistema solicitando información respecto de la cantidad de procesos en su nodo y cual es su *Protector* vecino. Sigue este procedimiento con los vecinos hasta que encuentre el proceso que se esté ejecutando en un nodo no planificado. Este método de elección hace que el usuario no necesite obligatoriamente definir el nodo que se va a reemplazar, manteniendo así el máximo de transparencia posible.

Tras identificar el proceso que debe asumir, el nuevo *Protector* envía una petición al *Protector* del nodo que está con el proceso para incorporarse a la cadena de protección. Tras el establecimiento de la nueva cadena de protección, el proceso seleccionado recibe un comando para que en el siguiente *checkpoint* realice la migración al nuevo nodo, evitando así que el proceso tenga que parar su ejecución inmediatamente. Finalmente el proceso migrado reasigna su nuevo *Protector*. Un diagrama de flujo de este procedimiento esta en la figura 6.

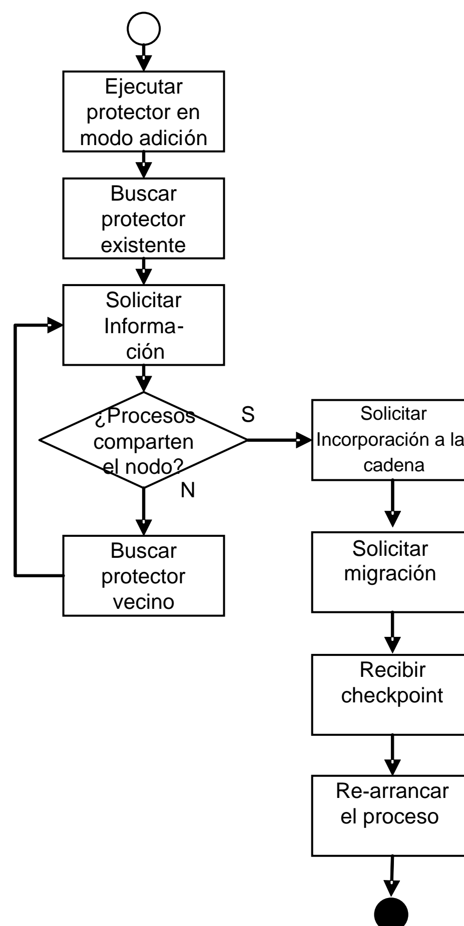

Figura 6: Flujo de ejecución del proceso de reemplazo de nodos que han fallado.

Es muy importante en todo este proceso, garantizar que la distribución final de los procesos, represente exactamente la que existía antes del fallo que generó este cambio. Es decir, que tenemos que migrar exactamente el proceso, o los procesos que antes se ejecutaban en el nodo reemplazado

con la finalidad de reestablecer la distribución que había sido planificada y que existía exactamente antes del fallo. Para eso, modificamos la estructura de RADIC creando el concepto de grupo de procesos, que es un conjunto indisociable de procesos que están obligados a estar siempre ejecutando en el mismo nodo.

5 CONCLUSIONES Y LÍNEAS ABIERTAS

En este artículo, hemos presentado un problema añadido a la tolerancia a fallos, de forma que a pesar de que la tolerancia a fallos sea algo importante y los sistemas que la implementan sean eficaces, las pérdidas de prestaciones que ocurren tras haber sido tolerado el fallo, son substanciales y pueden tornar inútil o impreciso los resultados de la ejecución de determinadas aplicaciones. De esta forma, creemos que la recuperación de las prestaciones es el siguiente paso a tener en cuenta después de garantizar la finalización correcta de las aplicaciones.

Hemos propuesto una solución basada en la arquitectura RADIC, que busca recuperar y reconfigurar el sistema para que se mantengan las prestaciones disponibles antes de la ocurrencia de un fallo. Esta solución está propuesta en dos situaciones diferentes, una con la creación de *spare nodes* y otra permitiendo el reemplazo de los nodos que hayan fallado. Esa solución sigue manteniendo las características de RADIC, de transparencia, escalabilidad y no existencia de elementos centrales.

Como trabajo futuro, se necesita ampliar los experimentos con la propuesta presentada, definir las posibles políticas de reemplazo a ser adoptadas y analizar el impacto de cada una de ellas en la aplicación. Realizaremos estudios sobre el impacto que la re-configuración causa en la eficiencia de la aplicación.

REFERENCIAS

[1] A.M. Agbaria and R. Friedman. *Starfish: fault tolerant dynamic MPI programs on clusters of workstations.* In Proceedings of 8th International Symposium on High Performance Distributed Computing, pages 167–176, August 1999.

[2] R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam,Y. Dandass and M. Apte, *“MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing”*, 1st International Symposium on Cluster Computing and the Grid, May, 2001.

[3] G. Bosilca, A. Bouteiller, F. Cappello, et al. *MPICHV: Toward a Scalable Fault Tolerant MPI for Volatile Nodes.* In Proceedings of SuperComputing 2002 (SC2002), November 2002.

[4] G. Burns, R. Daoud and J. Vaigl, *“LAM: An open cluster environment for MPI”*, Proceeding of Supercomputing Symposium, pp.379--386, 1994, Toronto, Canada.

[5] A. Duarte, D. Rexachs and E. Luque. *A distributed scheme for fault-tolerance in large Clusters of Workstations.* In Proceedings of Parallel Computer 2005 (Parco2005). September 13-16, 2005. Málaga. Spain. (in press)

[6] E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. *A survey of rollback-recovery protocols in message-passing systems.* ACM Computer Survey, 34(3):375–408, 2002.

[7] Future Technologies Group, *Berkeley Lab Checkpoint/Restart (BLCR),* http://ftg.lbl.gov/, HPCRD at Lawrence Berkeley National Laboratory, November 2005.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, *"A high performance, portable implementation of the MPI message passing interface standard"*, Parallel Computing, Volume 22(6), pp.789-828, 1996.

[9] Jalote P., *Fault tolerance in distributed systems*. Prentice Hall 1994

[10] S. Rao, L. Alvisi, and H. Vin. *Egida: An extensible toolkit for low-overhead fault-tolerance.* In Proceedings of IEEE Fault-Tolerant Computing Symposium (FTCS-29), Madison, WI, June 1999.

[11] S. Rao, L. Alvisi and H. Vin. T*he Cost of Recovery in Message Logging Protocols.* IEEE Transactions on Knowledge and Data Engineering, Vol. 12, No. 2, April 2000.