Universitat Autònoma de Barcelona

Escola Tècnica Superior d'Enginyeria

Departament d'Arquitectura de Computadors i Sistemes Operatius

FTDR: Tolerancia a fallos, en *clusters* de computadores geográficamente distribuidos, basada en Replicación de Datos

Josemar Rodrigues de Souza

Bellaterra (Cerdanyola del Vallès), Barcelona, España

2006

Josemar Rodrigues de Souza

FTDR: Tolerancia a fallos, en clusters de computadores geográficamente distribuidos, basada en Replicación de Datos

Tesis presentada para optar al grado de Doctor en Informática por la Universidad Autónoma de Barcelona.

Director de Tesis: Profesora Dra. Dolores Isabel Rexachs del Rosario.

Bellaterra (Cerdanyola del Vallès), Barcelona, España

2006

FTDR: Tolerancia a fallos, en clusters de computadores geográficamente distribuidos, basada en Replicación de Datos

Memoria presentada por **Josemar Rodrigues de Souza** para optar al grado de Doctor en Informática por la Universidad Autónoma de Barcelona.

Trabajo realizado en el Departamento de Arquitectura de Computadores y Sistemas Operativos (DACSO), Escuela Técnica Superior de Ingeniería (ETSE), Universidad Autónoma de Barcelona (UAB), bajo la dirección de la **Dra. Dolores Isabel Rexachs del Rosario**.

Bellaterra (Cerdanyola del Vallès), Barcelona, España, Febrero 2006

Director de Tesis

Dra. Dolores Isabel Rexachs del Rosario.

A mi familia

Agradezco

Al Profesor Dr. Emilio Luque Fadón: por tu amistad y pelos sabios consejos. Tu es muy integro y coherente. Aprendí muchísimo contigo. Muchas gracias por todo y por las frases que yo leí en tu despacho: "Honestos pero no modestos", "Nunca hice lo que debía, siempre hice lo que creía", ellas fueran muy valiosas. Fuerte abrazo.

A Lola, Profesora Dra. Dolores Isabel Rexachs del Rosario, mi amiga: con tuya dirección, este trabajo se volvió, completo e perfecto. Besos. Fuerte abrazo en David.

A Marcelo Soares Souza: o especialista que acreditou e ajudou a provar o que os "especialistas" diziam ser impossível e por me fazer gostar e por me fazer ver o quanto é maravilhoso o Linux e o mundo do software livre.

A Marcela, minha mulher: pela paciência e dedicação! E que paciência! Beijão.

A Vini, Celo e Cacá: meus filhos, minha vida, meu mundo! Beijo do pai.

A Goja: mentor intelectual disso tudo.

A Leila: obrigado tiazona!

A Zú: mãe das mães, descanse em paz!

A meu pai: pela moral e integridade que sempre passou para todos nós, descanse em paz.

A minha mãe: pelo orgulho que sempre sente do filhão.

A meus irmãos: pelo apoio e por acreditarem no que faço.

A Leandro Coelho, meu novo filho, figura coerente e íntegra, pelo grande suporte moral nessa maravilhosa terra que se chama Espanha.

Ao Dr. Alfredo Matta: que me fez ver a academia com outros olhos.

Ao Dr. Marcelo Moret: que "abriu" meus olhos para os prazos.

Al Dr. Fernando Core: por la inolvidable ayuda en la Tesina.

Al Profesor Dr. Tomás Margalef: que me enseñó por la primera vez el PVM.

Al Profesor Dr. Porfidio Hernández, Profesora Dra. Ana Ripoll, Profesor Dr. Remo Suppi, Dr. Domingo Benitez, Gemma Roqué, Dr. Bahjat Qazzaz, Santiago Roata, Adriana Gaudiano, Eduardo Galobardes, Pilar Gómez, Dra. Anna Morajko, Juan Carlos, Dr. Xiao Yuan, Gustavo Martinez, Ijhab e demás personas del DACSO: coloco todos juntos aquí, mis amigos, para decir muchas gracias. Me perdonen los que no cité explícitamente, los amigos son muchos y el espacio para escribir es poco.

Mara, André Chagas, Likiso Hattori, Lorrene Vieira, Cláudio Amorim, Joseval Melo, Dr. Hernani Pereira, Mila, Rosali, Haydee Chavero, Durval, Paula Vanucci, Taize Araujo, Carlos Oliveira, Luiz Tadeu, Geraldo Brito, Chris, Manuela Rodriguez, Gustavo Arnold, Dr. Frede Carvalho, Dra. Lynn Alves, Dr. Fábio Ribeiro, Dr. Gilney Zebende, Rejane, Tânia Regina, Dr. Antonio Luiz, Dr. Carlos Alberto Caroso, Murilo Boratto, Thiago Tavares e demais pessoas do CEBACAD, do LPAD, do CEPPEV, da UNEB e da UCSAL: coloco todos juntos aqui, meus amigos, para dizer muito obrigado. Perdoem-me os que não citei explicitamente, os amigos são muitos e o espaço para escrever é pouco.

A aqueles que contribuíram muito; mesmo sem querer; muita luz.

A Dios y a San Jordi, mi Santo fuerte, por la protección recibida.

A Deus e a Oxossi, meu Santo forte, pela proteção recebida.

Resumen

El crecimiento de los *clusters* de computadores, y en concreto de sistemas *multicluster* incrementa los potenciales puntos de fallos, exigiendo la utilización de esquemas de tolerancia a fallos que proporcionen la capacidad de terminar el procesamiento.

El objetivo general planteado a sistemas de tolerancia a fallos es que el trabajo total se ejecute correctamente, aún cuando falle algún elemento del sistema, perdiendo el mínimo trabajo realizado posible, teniendo en cuenta que las prestaciones disminuyen debido al *overhead* necesario introducido para tolerar fallos y a la perdida de una parte del sistema.

Esta Tesis presenta un modelo de tolerancia a fallos en *clusters* de computadores geográficamente distribuidos, utilizando Replicación de Datos denominado FTDR (*Fault Tolerant Data Replication*). Está basado en la replicación inicial de los procesos y una replicación de datos dinámica durante la ejecución, con el objetivo de preservar los resultados críticos. Está orientado a aplicaciones con un modelo de ejecución *Master*/*Worker* y ejecutado de forma transparente al usuario. El sistema de tolerancia a fallos diseñado, es configurable y cumple el requisito de escalabilidad.

Se ha diseñado un modelo funcional, e implementado un *Middleware* . Se propone una metodología para incorporarlo en el diseño de aplicaciones paralelas. El modelo está basado en detectar fallos en cualquiera de los elementos funcionales del sistema (nodos de computo y redes de interconexión) y tolerar estos fallos a partir de la replicación de programas y datos realizada, garantizando la finalización del trabajo, y preservando la mayor parte del computo realizado antes del fallo, para ello es necesario, cuando se produce un fallo, recuperar la consistencia del sistema y reconfigurar el multicluster de una forma transparente al usuario.

El *Middleware* desarrollado para la incorporación de la tolerancia a fallos en el entorno multicluster consigue un sistema más fiable, sin incorporar recursos hardware extra, de forma que partiendo de los elementos no fiables del *cluster*, permite proteger el cómputo realizado por la aplicación frente a fallos, de tal manera que si un ordenador falla otro se encarga de terminar su trabajo y el computo ya realizado está protegido por la Replicación de Datos.

Este *Middleware* se puede configurar para soportar más de un fallo simultáneo, seleccionar un esquema centralizado o distribuido, también se pueden configurar parámetros relativos a aspectos que influyen en el *overhead* introducido, frente a la pérdida de más o menos computo realizado.

Para validar el sistema se ha diseñado un sistema de inyección de fallos. Aunque añadir la funcionalidad de tolerancia a fallos, implica una perdida de prestaciones, se ha comprobado experimentalmente, que utilizando este sistema, el *overhead* introducido sin fallos, es inferior al 3% y en caso de fallo, después de un tiempo de ejecución, es mejor el tiempo de ejecución (*runtime*) tolerando el fallo que relanzar la aplicación.

Índice

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

Este capitulo plantea el problema de manejar eficientemente un número elevado de computadores en un ambiente heterogéneo y la probabilidad de desconexión o fallo. Analizamos la necesidad de utilizar tolerancia a fallos en *clusters* en general y en particular se aplica a *clusters* de computadores geográficamente distribuidos.

El número de computadores que se interconectan en red en la actualidad está creciendo, por otro lado, el software que se desarrolla cada vez exige más y más recursos de los computadores, como memoria y espacio físico [14].

La llegada de la computación de altas prestaciones (h*igh performance computing*) a través de *clusters* de computadores basados en *Workstation* (WS) y redes convencionales (local o remota) o COTS (c*ommodity off the self*), ha posibilitado que muchas instituciones y organizaciones vean viable la construcción y el uso de computadores paralelos.

En las llamadas arquitecturas paralelas el objetivo principal es el aumento de la capacidad de procesamiento, utilizando el potencial ofrecido por un gran número de procesadores (alto rendimiento y alta disponibilidad).

Entre las plataformas de computación distribuida, los CoHNOW *(Collection of Heterogeneous Network of Workstation), basados en COTS,* son una solución eficiente y económica para la computación intensiva de datos. Los centros de investigación han aprovechado su red de área local (LAN) para construir *clusters* y se puede utilizar la red de comunicación extendida (WAN - red de comunicaciones que conecta computadores dispersos en una amplia área geográfica), para construir CoHNOW para procesamiento paralelo.

Al ser una solución independiente (no comercial/no propietaria), al construirse un COW es necesario para que el agrupamiento de las *workstations* interconectadas, se transformen en una Máquina Paralela Virtual (MPV), disponer de un software específico o librerías de comunicación, responsables de intercambiar mensajes (*message passing*) entre los nodos de la MPV. Algunas librerías de comunicación conocidas son el MPI (M*essage Passing Interface)* [50], PVM (*Parallel Virtual Machine*), [34], LAM/MPI [43].

Un CoHNOW requiere poner énfasis en el diseño de la arquitectura, para mejorar el tiempo de respuesta de la aplicación, en función de la funcionalidad o carga de trabajo. Con el aumento del número de computadores conectados, el aumento del ancho de banda y la garantía de funcionamiento de Internet, la posibilidad de interconectar *clusters* geográficamente distribuidos, se ha vuelto asequible técnicamente y económicamente. Así, un CoHNOW puede trabajar en la solución de grandes problemas computacionales, aunque su uso no sea trivial y requiera tener en cuenta aspectos claves como la arquitectura y la distribución de la carga de trabajo [11].

Los *clusters* tienen la característica de ser máquinas fácilmente escalables, o sea, la adición de un nodo en el caso de los *clusters*, representa normalmente una ganancia de

prestaciones en el sistema. La construcción de estos CoHNOW tiene tres retos: alto rendimiento (*High Performance:* HP) y alta disponibilidad (*High Availability:* HA) y alta productividad (*High Throughput:* HT) [14] [64]. A pesar de que, tal y como hemos comentado manejar eficientemente un numero elevado de computadores en un ambiente tan heterogéneo (heterogeneidad de nodos de computo, de redes, etc.) no es trivial, por ello requiere un cuidadoso diseño de la arquitectura y su funcionalidad. Respecto a la alta disponibilidad, cuando consideramos un elevado numero de nodos funcionando durante un tiempo elevado, fallos en los nodos o desconexión son los eventos más probables, reduciendo el MTBF (tiempo medio entre fallos - "*Mean Time Between Failures*") de la arquitectura paralela CoHNOW como un todo [13].

En un escenario de colaboración entre *clusters* geográficamente distribuidos, a través de una WAN pública, aumenta el número de computadores interconectados entre si y trabajando en la solución de un problema común, es decir, permite aumentar fácilmente el número de computadores de la máquina paralela, por lo tanto también aumenta la probabilidad de fallos transitorios, intermitentes, o permanentes, durante la ejecución de una aplicación, por ello también, es necesario buscar medios para proporcionar tolerancia a fallos [7] [8].

Es necesario definir una arquitectura que permite ejecutar aplicaciones paralelas que requieren alto rendimiento (HP) y alta disponibilidad (HA) y alta productividad (HT). Debe, garantizar que se realice la ejecución eficiente de una aplicación paralela cuando se utiliza un entorno con un elevado número de nodos, durante un largo período de tiempo. Por otro lado, es necesario considerar que la probabilidad de fallo aumenta y puede llegar a ocurrir que el fallo ocasione la perdida total del trabajo realizado. Por lo tanto para proveer HA es necesario utilizar técnicas de tolerancia a fallos [7].

En caso de aparición de fallo en algún componente, la tolerancia a fallos busca que el sistema siga trabajando, aunque esté funcionando en modo degradado, hasta que acabe la ejecución, lo que podrá incidir en mayor o menor medida en sus prestaciones. El número de fallos que pueden estar presentes en un momento dado dependerá del número de nodos del sistema, del tiempo medio entre fallos (MTBF) y del tiempo de ejecución de la aplicación. La probabilidad de que dos o más fallos ocurran simultáneamente decrece.

La tolerancia a fallos en un sistema se logra mediante la inclusión de técnicas de redundancia [8]. Puede haber redundancia en cualquier nivel: utilización de componentes *hardware* extra (redundancia en el *hardware*), repetición de las operaciones y comparación

de los resultados (redundancia temporal), codificación de los datos (redundancia en la información) e incluso la realización de varias versiones de un mismo programa y del uso de técnicas de Replicación de Datos (redundancia de datos) [69] o de *checkpoint* (redundancia de estados) [16].

Los mecanismos para tolerancia a fallos pueden tener un soporte *hardware* o *software*, o bien una combinación de ambos. En sistemas en que la incidencia de fallos sea escasa puede ser recomendable emplear mecanismos de bajo coste con soporte *software*, siempre que el algoritmo pueda proporcionar la garantía de que acabe la ejecución correctamente aunque se degraden sus prestaciones.

Un modelo de programación muy extendido en cluster de computadores es el *Master/Worker* (MW), tiene unas características intrínsecas que permite abordar soluciones de tolerancia a fallos sin que sea obligatorio considerar la utilización de nodos de cómputo extra (redundancia *hardware*). En una arquitectura *Master/Worker* si todos los *Workers* realizan el mismo cómputo, existe una redundancia intrínseca, o dicho de otro modo, existe una replicación de procesos si se hace un único programa con el código del Master y del *Worker* (SPMD). Por otro lado, usualmente no hay comunicación entre los *Workers*, esta restricción de las comunicaciones en *Master/Worker* simplifica el problema de los mensajes: cada *Worker* sólo se comunica con el *Master* [14]. Para una arquitectura basada en el modelo de ejecución *Master/Worker*, donde todos los nodos ejecutan el mismo programa (replicación de procesos), se puede considerar que no es necesario realizar *checkpoint*, siendo adecuado utilizar técnicas de Replicación de Datos [69] que poseen un menor coste de cómputo y comunicación.

De acuerdo con Venugopal, Buyya y Ramamohanarao en [67], “Replicación de Datos es la manera de categorizar cómo las copias de datos son creadas y mantenidas en la red. Replicación de datos tiene doble objetivo: uno, es mejorar las prestaciones a través de la reducción de la latencia de red y el otro, proveer garantía de funcionamiento, creando copias auxiliares múltiples de datos.”

1.1. Cluster de computadores geográficamente distribuidos

Nuestro trabajo está centrado en diseñar un sistema que permita garantizar alta disponibilidad en *cluster* de computadores geográficamente distribuido.

Un CoHNOW debe estar eficientemente estructurado para la ejecución de aplicaciones paralelas en entorno *Master/Worker*. Estos *clusters* pueden estar organizados de una forma

jerárquica (Figura 1-1) en forma de árbol. Utilizamos una arquitectura donde cada *cluster* es un *Master/Worker* en sí mismo [3] [32] [60], existe un *cluster* principal (MC), donde está el *Master* principal (MMT), encargado de comenzar y acabar la aplicación. Cada *cluster* del multicluster forma un subcluster con su propia estructura *Master/Worker*, de forma que los subcluster son considerados como *Worker* del *Master* principal. Para la comunicación entre *cluster* utilizamos unos gestores de comunicación diseñado para mejorar el rendimiento (lograr las máximas prestaciones en las comunicaciones a través de Internet) y gestionar la disponibilidad de la interconexión entre los *cluster*, de forma que se encarga de gestionar los fallos intermitentes que se pueden producir en Internet.

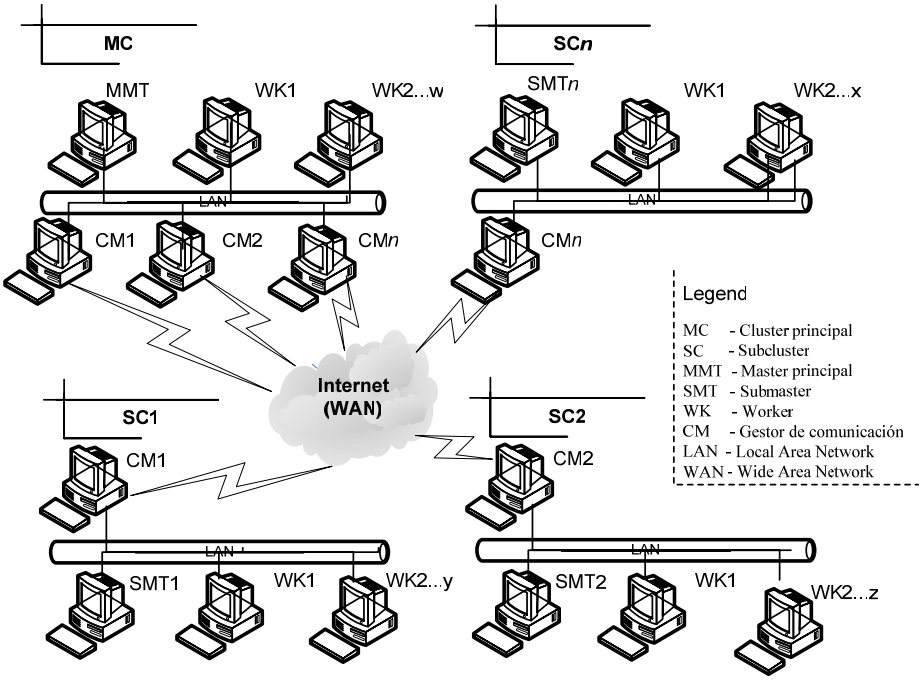

Figura 1-1: CoHNOW *Master/Worker*

Para que la utilización del CoHNOW geográficamente distribuido tenga la característica de alta disponibilidad, además de soportar los fallos intermitentes de la red, es necesario incorporar mecanismos de tolerancia a fallos con la finalidad de que aún en presencia de fallos en los nodos, el trabajo total se realice correctamente hasta el final, sin intervención del usuario. Nosotros hemos propuesto FTDR (*Fault Tolerant Data Replication*) con el objetivo de tolerar fallos en los nodos y como hemos visto, sobre una arquitectura que utiliza gestores de comunicación que se encargan de prevenir los fallos transitorios de la WAN.

Puesto que la arquitectura del sistema es una arquitectura *Master/Worker* jerárquica, cuando disponemos de varios Subclusters *Master/Worker*, el mecanismo de tolerancia a fallos está implementado en cada uno de los *cluster*, evitando (minimizando) en la medida de lo posible la Replicación de Datos a través de Internet, o sea utilizamos básicamente el mismo esquema en todos los subclusters.

1.2. Objetivos

El objetivo es ejecutar en un multicluster el trabajo total correctamente, aún cuando falle algún elemento del sistema, perdiendo el mínimo trabajo realizado posible, teniendo en cuenta que las prestaciones disminuyen debido a la *overhead* introducido para tolerar fallos y a la perdida por fallos de nodos del sistema. El sistema propuesto busca perder el mínimo trabajo posible realizado por los elementos que queden desconectados por fallo, en otras palabras, recuperar la mayor parte del trabajo que fue ejecutado por un nodo que falle.

La estrategia de tolerancia a fallos adoptada debe tener en cuenta que las prestaciones del *cluster* se degraden el mínimo posible, tanto en ausencia como en presencia de fallos (*overhead*), es decir, uno de los objetivos de FTDR es tener en cuenta la eficacia, o sea, controlar y predecir cuanto *overhead* se va a introducir en el sistema. La fuente de *overhead* en el modelo propuesto viene generado básicamente por los mensajes extras generados para la Replicación de Datos y la detección de fallos, introducidos en el sistema durante la ejecución de los algoritmos, ya que es necesario que la Replicación de Datos se realice en otro nodo del sistema.

Considerando que utilizamos *clusters* de bajo coste económico, FTDR está basado en redundancia de información (*software)*, no incluyendo redundancia física (*hardware)* especializado en la tolerancia a fallos. Teniendo en cuenta que en un *cluster*, la redundancia física de nodos de cómputo es intrínseca, estamos interesados en aprovechar dicha redundancia física intrínseca del *cluster*, para, de forma transparente al usuario crear una redundancia funcional basada en la Replicación de Datos, o sea, otras máquinas asumen funciones (programas y datos) de los nodos que fallen, no permitiendo que el sistema sufra una avería como un todo. Para esto, es necesario gestionar un pool de recursos de cómputo y comunicación.

El objetivo del modelo es asegurar que existe la redundancia funcional necesaria para que el trabajo se pueda terminar en caso de fallo, detectar y diagnosticar fallos en cualquiera de los elementos funcionales del sistema y tolerar este fallo reconfigurando el sistema y

recuperando la consistencia de forma que se garantice que el trabajo termina correctamente. El modelo corresponde a un *Middleware* transparente al usuario, implementado como una capa entre la aplicación y la librería de paso de mensajes.

Para la tolerancia a fallos se debe realizar un *Middleware* encargado de la protección de cómputo, utilizando el mismo esquema en cada uno de los *clusters*. Además esta tolerancia a fallos se realiza de forma transparente al usuario. Se basa en replicación de procesos inicialmente en todos los nodos, se configura el multicluster y cuando comienza la ejecución se replican los datos iniciales y a medida que avanza el cómputo se van replicando los resultados que computan los *Workers,* evitando el *checkpoint*. En cada uno de los Subclusters se realiza la Replicación local de Datos, además del envío de resultados al *cluster* principal, detección y diagnóstico del fallo, la recuperación del trabajo realizado y la re-configuración del *cluster*, con el re-direccionamiento de la ejecución, o sea, cuando un nodo falla, el sistema debe reconfigurarse, aislando el nodo que ha fallado y ejecutando el resto del trabajo entre los nodos activos.

El modelo permite configurar varios parámetros, como el número de fallos simultáneos en cada uno de los cluster, especificar si se desea trabajar con una Replicación de Datos centralizada, replicando todos los datos del *Master* en otro nodo del sistema que asumirá la tarea de *Master* en caso de fallo o utilizando una Replicación de Datos distribuida, replicando los datos entre los *workers* y los *subclusters* de lo CoHNOW. Estas opciones están soportadas por un *Middleware Master/Worker*.

1.3. FTDR

En resumen, nosotros proponemos un modelo funcional de tolerancia a fallos en un entorno multicluster geográficamente distribuido que usa un modelo de ejecución Master/Worker, basado en Replicación de Datos (FTDR) y hemos desarrollado el soporte necesario.

FTDR está orientado a un sistema CoHNOW con los *cluster* geográficamente distribuidos, enlazado mediante gestores de comunicación que toleran los fallos intermitentes en la WAN, donde se ejecutan aplicaciones paralelas de alto rendimiento, bajo el paradigma *Master/Worker*. También se propone una metodología al usuario que le permita proteger su aplicación frente a fallos, cuando está ejecutando sobre la arquitectura propuesta.

La arquitectura multicluster geográficamente distribuida con tolerancia a fallos, requiere:

- Los subclusters *Master/Worker* con sus gestores de comunicación (CM), que disponen del *middleware* necesario para lograr una comunicación eficiente entre *clusters* y tolerar los fallos intermitentes en la WAN.
- La tolerancia a fallos de los nodos, requiere la definición de un modelo funcional y la construcción de un *Middleware* específico para cada subcluster a nivel de sistema.
- El diseño de una metodología para que se incorpore a nivel de aplicación para explotar el *Middleware* que soporta la tolerancia a fallos.

Proponemos y analizamos dos alternativas de Replicación de Datos, que presentan diferentes ventajas e inconveniente:

- Replicación centralizada de datos, que presenta ventajas porque permite una rápida recuperación/reconfiguración del cluster en caso de fallo, pero tiene el inconveniente de que el costo de mantener el protector del *Master* dormido, con copia de todos los datos es elevado. Este coste es debido al *overhead* que se introduce en las comunicaciones (Figura 1-2) al tener que enviar los resultados a otro nodo para la Replicación de Datos, este *overhead* depende de la relación computo comunicación de la aplicación y del volumen de los resultados de computo que devuelve el *Worker,* pudiendo llegar a ser, como vemos en la gráfica en la ejecución de la Multiplicación de Matrices, casi 10 veces superior a

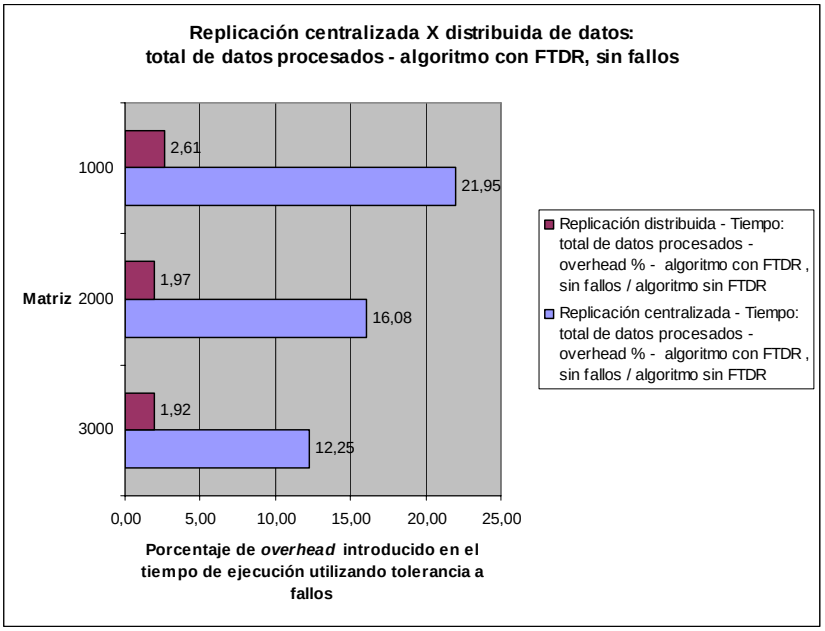

Figura 1-2: Replicación centralizada X distribuida de datos - sin fallo

la replicación distribuida. Este modelo presenta ventajas en el momento de fallos, pero empeora la solución sin fallos.

- Replicación distribuida de datos: existe una replicación distribuida de los resultados enviados al *Master* entre los *Workers*. Mejora el *overhead* sin fallos (Figura 1-2). La información que se mueve a través de la red para protección causa un *overhead* mínimo, ya que se envía una replicación de la información crítica del *Master*. En caso de fallo hay que reconstruir el *Master*, la re-configuración es centralizada, y costosa requiriendo para la ejecución en todos los nodos, pero presenta la ventaja de que el *overhead* introducido sin fallos es inferior al 3%.

1.4. Organización de la memoria

La memoria está organizada de acuerdo con los siguientes capítulos:

El Capitulo 2, muestra algunos conceptos de arquitectura de computadores paralelos, tolerancia a fallos, Replicación de Datos, tolerancia a fallos en *clusters* de computadores, *Message Passing Interface* (MPI) y validación de modelos. Muestra algunas definiciones básicas y explica parte de la terminología (muchas veces confusa) utilizada.

El Capitulo 2 , también, describe los trabajos relacionados más interesantes considerando la alternativa de la Replicación de Datos, dentro de cada *cluster*, debido a las características operativas y prestacionales, a nivel de1 *cluster* y con varios *clusters*: fallos en cada uno de los *cluster* y como interfieren en el global.

El Capitulo 3, describe la arquitectura multicluster y el sistema de tolerancia a fallo en la red. Para lo fallos de red, utilizamos el gestor de comunicación, que es un *middleware* básico de la arquitectura multicluster usada, necesario para configurar el sistema multicluster. Es un *software* mínimo que forman un multicluster, que permite la unión operativa de los *cluster* (análogo al *software* mínimo que requiere Grid): sobrevive a las caídas de la red. Para fallos en esta máquina consideramos que estaría duplicado.

El Capitulo 4 explica el modelo. La transparencia en el modelo está basada en una metodología para un *Master/Worker*, donde se muestra como se adapta a la aplicación. La metodología se puede generalizar a cualquier aplicación *Master/Worker*. Para escribir cualquier aplicación *Master/Worker* se tiene que escribir siguiendo estos pasos.

El Capitulo 5 describe la validación experimental del modelo de tolerancia a fallos propuesto. El trabajo experimental realizado es un ejemplo de aplicación de la metodología

del modelo.

A continuación, de los resultados experimentales hay un capítulo dedicado a las conclusiones y líneas abiertas.

Capitulo 2

Conceptos y trabajos relacionados

Este capítulo muestra algunos conceptos básicos de arquitectura de computadores paralelos, tolerancia a fallos, Replicación de Datos, tolerancia a fallos en *clusters* de computadores, *Message Passing Interface* y validación de modelos de tolerancia a fallos. Muestra algunas definiciones básicas y presenta la terminología utilizada.

Se describen algunos trabajos relacionados, realizados por grupos de investigación que en la actualidad trabajan en el problema de tolerancia a fallos en aplicaciones ejecutadas en *clusters* de computadores, que utilizan el paradigma *Master/Worker*, y MPI para la comunicación entre los nodos del *cluster*.

2.1. Introducción

Este capítulo presenta un resumen de conceptos relacionados con las diferentes áreas utilizadas de Arquitectura de Computadores que tienen influencia en este trabajo.

Existen diferentes tipos Computadores Paralelos propuestos y construidos a lo largo de los años. Amdahl [2], Batchu et al [9], Beaumont et al [11], Buyya, [14], Culler et al [23], Foster [27], Geist et al [33], Hennessy et al [38], Hwang [40], Pacheco [51], Patterson et al [52], Venugopal et al[67], entre otros investigadores, presentan distintas propuestas. Nosotros presentamos una forma de categorizarlos que permite contextualizar nuestro trabajo.

En la actualidad, grupos de investigación [9] [16] [25] [33] [69] están trabajando en el problema de tolerancia a fallos en aplicaciones paralelas con memoria distribuida.

2.2. Arquitectura de computadores

Una de las taxonomías más conocida y utilizada para computadores es la taxonomía de Flynn [57]. Aunque esta taxonomía a pesar de que no presenta mucho detalle, es la base para otras más detalladas. Nos centraremos en los computadores MIMD (*Multiple Instruction Multiple Data*), podemos considerarlos divididos en dos categorías básicas: multiprocesadores (arquitecturas paralelas con memoria compartida) y multicomputadores (arquitecturas paralelas con memoria distribuida o arquitecturas paralelas de paso de mensajes) [38] [52] (Figura 2-1).

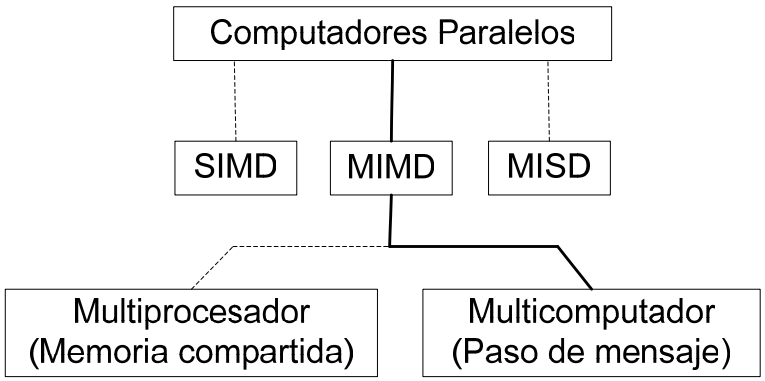

Figura 2-1: Clasificación de los computadores paralelos

Los multicomputadores no comparten memoria común. Cada nodo de procesamiento tiene su propia CPU/CPUs y memoria, también pueden disponer de discos u otros dispositivos de Entrada/Salida (E/S). Los nodos de procesamiento cambian datos entre ellos

usando mensajes (primitivas de comunicación *Send* y *Receive*) que transitan en las redes de interconexión [14] [38] [52].

La categoría multicomputadores (*multi-computers*), está dividida en: MPP (*Massively Parallel Processors*) y COW (*Cluster of Workstations*) donde se consideran los agrupamientos de estaciones de trabajo que son interconectadas a través de redes estándar (Ethernet, ATM, etc.).

Otro tipo de Computadores Paralelos que presenta un aspecto diferente desde el punto de vista de la arquitectura son los llamados Grid o colección de recursos autónomos geográficamente distribuidos conectados a través de una red y que conforman un sistema de altas prestaciones virtual. Su configuración cambia con el tiempo dinámicamente dependiendo de la disponibilidad de los recursos, capacidad, prestaciones y necesidades de calidad de servicio de los usuarios. En un *cluster* la asignación de recursos del sistema se realiza por un gestor de recursos y todos los nodos trabajan como un único sistema de cómputo, cosa que no ocurre en Grid [67].

2.2.1. *Cluster* of *Workstation*

La motivación inicial para los *Cluster of Workstation* (COW) fue construir computadores paralelos de bajo coste; existe una gran diferencia de precios entre una *workstation* y un supercomputador convencional. No obstante, las *workstation* han ido aumentando significantemente su capacidad de procesamiento, memoria y E/S, por otro lado el ancho de banda de las redes (LAN/WAN) están aumentando y la latencia disminuyendo, los COW son fácilmente interconectadas mediante las redes de computadores existentes, los COW pueden crecer añadiendo otras *workstation*, o agregando otro procesador en una máquina [14].

Utilizar COW es uno de los mejores caminos para superar el intervalo que existe entre nuestras necesidades de alto rendimiento y los recursos disponibles. Los COW han resultado ser un modo práctico y económico de se construir máquinas paralelas.

Al ser una solución independiente, al construirse un COW es necesario softwares tal como librerías de comunicación, generalmente “free software” [29], para que el agrupamiento de las workstation interconectadas, se transformen en una Máquina Paralela Virtual (MPV). Estos softwares y librerías de comunicación, son responsables de intercambiar mensajes (message passing) entre los nodos de la MPV. Algunas librerías de comunicación muy utilizadas por la comunidad científica son el MPI (*Message Passing*

Interface) [50], PVM (*Parallel Virtual Machine*), [34], LAM/MPI [43].

En la literatura especializada podemos encontrar COW como sinónimo de NOW, nosotros utilizamos la terminología NOW (n*etwork of workstations*), HNOW (h*eterogeneous network of workstations*) y CoHNOW (c*ollection of HNOWS, o collection of clusters*) como una subdivisión de COW (Figura 2-2).

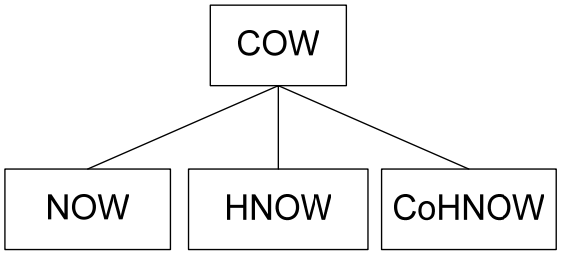

Figura 2-2: Subdivisión de COW

2.2.2. Clasificación de los *clusters*

Buyya [14 clasifica un *cluster* dependiendo de:

- el fin de su aplicación: alto rendimiento o alta disponibilidad (*high performance or high availability*),
- su propiedad: dedicado o no dedicado (*dedicated or non-dedicated*),
- su *hardware*: *cluster* de PC o *cluster* de *workstations*,
- el sistema operativo: Linux *clusters*, Solaris *clusters*, NT *clusters*, entre otros,
- los tipos de máquinas *clusters* homogéneos y *cluster* heterogéneos y
- el nivel de *clustering* (basada en la posición de los nodos).

Una NOW está compuesta por **máquinas homogéneas**, es decir, una NOW tiene como característica principal el hecho de que los computadores que componen la máquina paralela virtual son iguales a nivel de velocidad de procesador, memoria principal, capacidad de disco y sistema operativo iguales. En teoría, cuando estamos manejando esos tipos de máquinas, no tenemos que preocuparnos tanto sobre como dividir la carga de trabajo. Puesto que son homogéneos, la carga de trabajo puede ser dividida igualmente entre los nodos [14].

Un HNOW está compuesto por **máquinas heterogéneas**, cada una con sus características particulares: la velocidad del procesador, la capacidad de la memoria principal, el sistema operativo, la arquitectura de las máquinas, etc. [14].

Respecto a la posición de los nodos un CoHNOW es formada por una colección de

HNOWS, donde cada de ellas está separada geográficamente e interconectada por una WAN pública (Internet) o privada (dedicada). Ese tipo de máquinas tiene todas las particularidades de una simple HNOW, añadiendo, si utilizamos, la complejidad de todos los aspectos relacionados con el uso de una red de interconexión pública con latencia impredecible y ancho de banda variable. Por tanto, la comunicación entre las HNOWS, utilizando Internet, es más compleja que la comunicación dentro de la HNOW [14] [30] y la comunicación remota hecha mediante una red dedicada.

La arquitectura propuesta o usada en este trabajo, teniendo en cuenta esta clasificación *cluster* de Buyya, podríamos clasificarla, por el fin de su aplicación, como *high availability*, dedicado en un cluster de PC con el sistema operacional GNU/Linux, Kernel 2.4.20, con máquinas heterogéneas geográficamente distribuidas conectadas con dos tipos de redes: LAN y WAN.

2.2.3. Paradigma de programación

A fin de elegir un paradigma de programación paralela, el programador debe observar los recursos de computación paralela disponibles y también las características de la aplicación.

Muchos autores propusieron diferentes modos de clasificar los paradigmas de programación paralela. Buyya [15] después de analizar diferentes propuestas de clasificaciones, identificó: Flujo de datos o segmentada (*data pipelining)*; Divide y vencerás (*divide and conquer*); Paralelismo de datos (*single program multiple data* (SPMD)), Maestro-esclavo o granja de tareas (*Master/Worker o Task-Farming*) como los paradigmas de programación más usados.

Flujo de datos: *(Data pipelining)* está relacionada al mismo concepto de *pipeline* implementado en un procesador. La idea es atribuir para cada proceso una etapa de *pipeline* diferente, asociando diferentes fases con diferentes tareas.

En el paradigma **divide y vencerás** (*divide and conquer)*, el problema es dividido en pequeños subproblemas. La idea es crear tareas independientes y resolver en paralelo.

El paradigma SPMD (Figura 2-3) puede ser definido como un paradigma donde cada proceso (o nodo de procesamiento) ejecuta el mismo código sobre diferentes partes del dato.

La idea es replicar el programa en diferentes nodos de procesamiento y usarlo para ejecutar el mismo cálculo sobre diferente conjunto de datos. Obviamente, algunos nodos de procesamiento pueden ser responsables de distribuir los datos entre los otros nodos.

Normalmente este nodo de procesamiento también es responsable de juntar los datos al final del procesamiento y presentar los resultados.

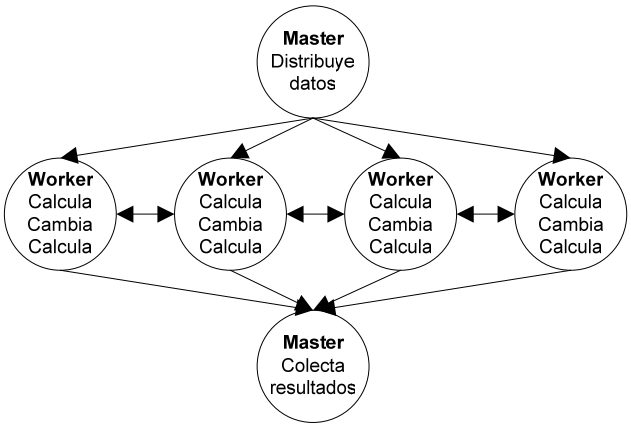

Figura 2-3: Paradigma SPMD

Generalmente, se pueden identificar tres pasos diferentes en este paradigma. Necesitamos de un paso de distribución de datos, un paso de cálculo y un paso de recepción de resultados. Otra característica del paradigma SPMD es que los nodos de procesamiento pueden comunicarse entre sí, para intercambiar datos [15].

En el paradigma *Master/Worker* (Figura 2-4), lo que utilizamos, es un proceso *Master* y múltiples *Workers*. El *Master* o controlador está relacionado a la idea de alguien controlando un contexto y asignando trabajos a los otros *Workers* o trabajadores.

La figura del controlador es llamada de *Master* y los otros componentes (los que reciben trabajo) son llamados *Workers*. La idea es tener un *Master* responsable de distribuir datos o

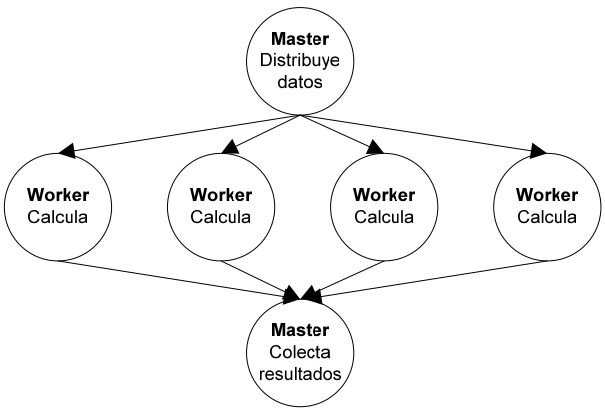

Figura 2-4: Paradigma *Master/Worker*

tareas a los *Workers*, y de ir recolectando los resultados parciales que van calculando los *Workers*, con lo que el *Master* obtiene el resultado final.

En el paradigma *Master/Worker,* solamente tenemos comunicación entre *Master* y

Workers; los *Workers* nunca se comunican directamente con otros *Workers*. La distribución de las tareas entre los *Workers* se puede realizar de forma dinámica (durante la ejecución) o de forma estática, cuando se conoce a priori que tarea va a ejecutar cada *Worker*.

En nuestro caso usamos el paradigma de programación *Master/Worker*. Desde el punto de vista del usuario básicamente utilizamos un sistema donde todos los *workers* ejecutan el mismo programa, además tenemos el programa del *Master*.

Junto con estos programas de la aplicación se ejecutan los procesos que corresponden al *Middleware* compuesto por programas de los procesos del Gestor de Comunicación (CM) y procesos de tolerancia a fallos que corresponden al *Middleware*. Nosotros utilizamos un único programa con el código del *Master*, el *Worker* y el *Middleware* que será replicado en todos los nodos. La distribución de tareas en cada uno de los *cluster* se hace de forma dinámica y entre *cluster* se realiza de forma estática.

2.2.4. Prestaciones en Computadores Paralelos

En Computadores Paralelos se utilizan usualmente como medidas de prestaciones el tiempo de ejecución y la productividad (*throughput*). Dependiendo de la utilización del sistema, se le concede más importancia a una medida que a otra.

Además se utilizan otras medidas de prestaciones adicionales como la alta disponibilidad (*high availability*) que está relacionada con la presencia de redundancia en el sistema (hardware y/o software) para reducir el tiempo de inactividad y la degradación de las prestaciones ante un fallo.

Relacionado con las medidas de prestaciones se suele hablar de eficiencia. La eficiencia evalúa en qué medida las prestaciones que ofrece un sistema para sus entradas se acercan a las prestaciones máximas que idealmente deberían ofrecer dado los recursos de que dispone.

También es importante el estudio de la escalabilidad de un sistema.

Otras propiedades importantes de un computador son la fiabilidad, la disponibilidad y servicialidad (RAS: *Reliability, Availability, Serviceability*).

La **fiabilidad** (*Reliability*) es la capacidad del sistema de producir consistentemente los mismos resultados y de acuerdo con sus especificaciones. La fiabilidad se puede expresar con un valor numérico referido a un periodo de tiempo, representando la probabilidad de que un sistema funcione conforme a sus especificaciones durante dicho periodo de tiempo. Pretende evaluar la frecuencia de fallos.

Para una tasa de fallos de λ averías/hora la media de tiempo entre averías es mostrado en la Figura 2-5.

$$MTTF = \frac{1}{\lambda}$$

Figura 2-5: Media de tiempo entre averías (MTTF)

La **disponibilidad** (*Availability*) es el grado en que un sistema sufre degradación de prestaciones o detiene su servicio por fallos de componentes, se puede incluir la penalización por prevención o mantenimiento. Está relacionada con la penalización que producen los fallos (Figura 2-6 y Figura 2-7).

$$Disponibilidad = \frac{MTBF}{MTBF + MTTR}$$

MTBF = *Mean TimeBetween Failure* (Tiempo Medio Entre Fallos).

MTTR = *Maximum Time to Repair* (Máximo Tiempo de Reparación).

Figura 2-6: Disponibilidad (*Availability*) - 1

En los sistemas con poca supervisión por estar situados en lugares remotos, requieren

$$Disponibilidad = \frac{MTTF}{MTTF + MTTR}$$

MTTF = *Mean Time to Failure* (Tiempo esperado hasta la ocurrencia de la avería).

Figura 2-7: Disponibilidad (*Availability*) - 2

una alta fiabilidad, pero pueden ser más tolerantes con la disponibilidad (Figura 2-8).

Las opciones para aumentar la disponibilidad son incrementar MTTF, es decir, incrementar la fiabilidad, lo cual es difícil o decrementar MTTR, reducir el MTTR es más habitual. Se puede conseguir mediante componentes *hardware* redundantes aislados o añadiendo sistemas de tolerancia a fallos (Tabla 2-1).

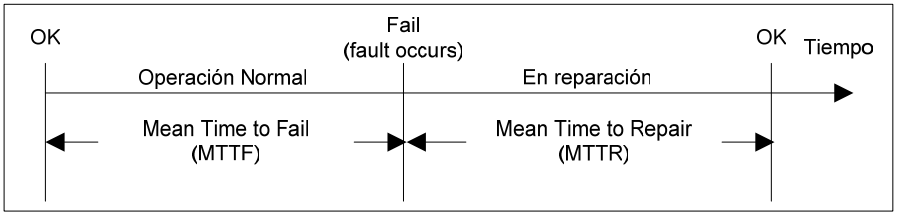

Figura 2-8: Alta fiabilidad X disponibilidad

La **servicialidad** (*Serviceability*) es la facilidad para que el sistema se mantenga funcionando. Se puede especificar indicando la probabilidad de completar un servicio en un periodo de tiempo dado.

Tabla 2-1: Opciones para aumentar la disponibilidad

Tipo de Sistema	Disponibilidad (%)	Tiempo de fallo por año
Estación de trabajo	99	3,6 días
Sistema de alta disponibilidad	99,9	8,5 horas
Sistema resistente a fallos	99,99	1 hora
Sistema tolerante a fallos	99,999	5 minutos

2.3. Tolerancia a fallos

Tolerancia a fallos es la capacidad de un sistema de mantenerse en funcionamiento ante un fallo. El término tolerancia a fallos fue presentado por Avizienis en 1967 [6], siendo ampliamente utilizado por la comunidad para designar toda el área de investigación ocupada con el comportamiento de sistemas sujetos a la ocurrencia de fallos. Muchas expresiones y conceptos aún no están consolidados y ni ampliamente aceptados. Diferentes grupos de investigadores utilizan las mismas expresiones de forma distinta, y expresiones diferentes para referirse a los mismos conceptos [22].

Diferentes autores se han ocupado de la nomenclatura y conceptos básicos del área de tolerancia a fallos. La formación del *IEEE-CS TC on Fault-Tolerant Computing* en 1970 y del IFIP WG 10.4 *Dependable Computing and Fault Tolerance* en 1980 aceleró el surgimiento de un conjunto consistente de conceptos y terminologías. Siete informes de intenciones fueron presentados en 1982 al FTCS-12 en una sesión especial sobre conceptos fundamentales de Tolerancia a Fallos, y Laprie preparó una síntesis en 1985 [45]. Otros trabajos de miembros de la IFIP WG 10.4, liderados por Laprie, se publicaron en 1991 en el libro *Dependability: Basic Concepts and Terminology* [44].

Dependiendo de la aplicación, la garantía de funcionamiento pondrá énfasis [45] en:

El sistema funciona sin interrupciones: fiabilidad (***reliability*)**.

El sistema no provoca averías catastróficas: seguridad (***safety***).

El sistema está disponible el máximo tiempo posible: disponibilidad (***availability***).

El sistema es fácilmente reparable: mantenimiento (***maintainability***).

El sistema impide el acceso no autorizado: confidencialidad (***confidentiality*)**

El sistema impide la alteración inadecuada de la información: integridad (***integrity***).

Una exposición sistemática de la terminología, presentada por Avizienis et al en [8], relacionadas con la garantía de funcionamiento de los sistemas informáticos son mostradas en la Figura 2-9, constando de tres partes: los problemas o **daños** para, las medidas o **atributos** de, y los **medios** por los cuales la garantía de funcionamiento es alcanzada.

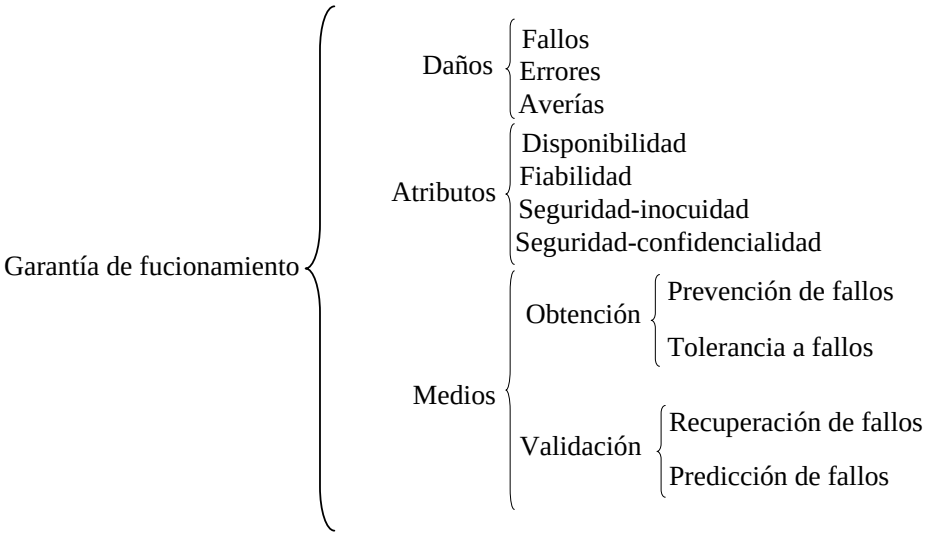

Figura 2-9: Terminología relacionada con la garantía de funcionamiento de los sistemas informáticos

Los problemas o **daños** son circunstancias no deseadas (aunque no inesperadas) que reducen la garantía de funcionamiento, es decir, que hacen que no se pueda confiar en el servicio suministrado por el sistema.

Una **avería** ocurre cuando el servicio entregado por el sistema no es el especificado y por tanto el usuario aprecia que el sistema no funciona bien. Un **error** es la manifestación de un fallo, un estado interno incorrecto del sistema. Un **fallo** es un defecto o imperfección física en el *hardware* o *software* del sistema. El fallo (*fault)* es la causa de un error (*error*) y éste, a su vez, es la causa de una avería (*failure*). Si un error ocurre en durante la ejecución y ocasiona una ejecución incorrecta de las funciones del sistema, se tiene una avería [54] [68].

La Figura 2-10 es una simplificación de la relación en el proceso de producción de los fallos, errores y averías, sugerida por Barry W. Jonson [42]. Estos pasos, no se producen de forma simultánea en el tiempo, sino que existe un tiempo de inactividad, llamado latencia del error desde el instante en que se produce el fallo hasta que se manifiesta el error. Durante este tiempo de latencia, se dice que el fallo no es efectivo y que el error está latente. De forma análoga se puede definir **la latencia de detección del error** y la latencia

de la producción de la avería.

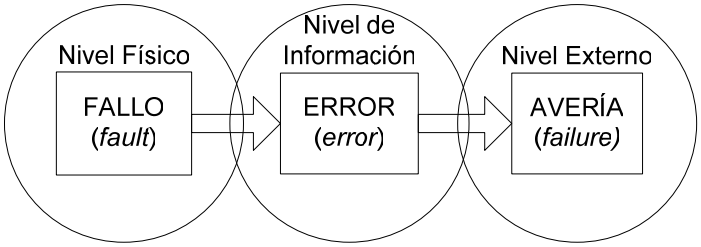

Figura 2-10: Simplificación de la relación entre fallos, errores y averías

Dependiendo de la duración, los fallos se clasifican en:

Permanentes: es un cambio irreversible en el componente.

Temporales: presentes durante un periodo corto de tiempo. Se dividen a su vez en:

- Transitorios: se deben a interferencias externas. La forma que aparecen y la duración son aleatorias.
- Intermitentes: aparece de forma transitoria pero es repetitivo, puesto que se debe a cierta combinación específica del sistema.

Los medios para obtener la garantía de funcionamiento son los métodos y técnicas. Se busca que no se produzca ninguna avería en el sistema. Para ello se aplican una serie de técnicas que conforman la tolerancia a fallos del sistema [8]:

Prevención de fallos (*Fault avoidance*): El objetivo es reducir la posibilidad de fallo del sistema, y para ello se eligen componentes de alta fiabilidad, se realiza un diseño e implementación extremadamente cuidadosa del sistema y se trata de proteger contra los agentes externos provocadores de fallos.

Enmascaramiento de fallos (*Fault masking*): Una vez superada la prevención de fallos, el objetivo siguiente es que el sistema siga funcionando a pesar de la existencia de fallos. Es decir, se producen fallos, pero estos no evolucionan hacia un error. Mediante técnicas de redundancia se suministra la información necesaria al sistema para evitar los efectos de los fallos.

Tratamiento del error: Se elimina el error antes de que produzca la avería mediante un **proceso de detección, diagnóstico, aislamiento, re-configuración y recuperación del mismo**. Se construyen sistemas con redundancia dinámica (funcional o física) donde, ante la detección del error y a través de la re-configuración, el sistema se degrada para seguir funcionando a coste de reducir su rendimiento.

Las técnicas de tolerancia a fallos están todas basadas en redundancia, exigiendo

componentes adicionales o **algoritmos especiales** [68]. Son implementadas generalmente por detección del error y subsiguiente recuperación del sistema [8]. Puede haber redundancia en cualquier nivel: utilización de componentes *hardware* extra (redundancia en el *hardware*), repetición de las operaciones y comparación de los resultados (redundancia temporal), codificación de los datos (redundancia en la información) e incluso la realización de varias versiones de un mismo programa y del uso de técnicas de consistencia para comprobar que el sistema funciona correctamente (redundancia en el *software*). La redundancia está tan íntimamente relacionada a la tolerancia a fallos que, en la industria, la terminología usada para designar un sistema tolerante a fallos es sistema redundante.

Cuando utilizamos **redundancia en el *software***, debemos considerar la prevención, la detección y la recuperación del sistema como un todo. La prevención consiste en generar la redundancia, la detección consiste en monitorizar y detectar el fallo y la recuperación consiste en retornar el sistema a condiciones operativas razonables, después de un fallo.

2.3.1. Evaluación del modelo de tolerancia a fallos

Una vez definido el sistema de tolerancia a fallos, tendremos que **evaluar** su comportamiento para comprobar que cumple con las expectativas de funcionamiento correcto. Debemos a priori, obtener la garantía de funcionamiento del sistema. Para ello se realiza una evaluación del comportamiento ante el fallo mediante la evaluación de modelos teóricos o la inyección de fallos.

Las medidas relacionadas con la garantía de funcionamiento, están determinadas estadísticamente observándose el comportamiento de los sistemas que se quiere medir. Las medidas para evaluación de tolerancia a fallos más utilizadas en la práctica son: tasa de defectos, MTTF, MTTR, MTBF. La Tabla 2-2 muestra una definición de esas medidas.

Tabla 2-2: Medidas de tolerancia a fallos

Medida	Significado
MTTF – mean time to failure	Tiempo esperado hasta la primera ocurrencia de la avería
MTTR – mean time to repair	Tiempo medio para reparo del sistema
MTBF – mean time between failure	Tiempo medio entre los defectos del sistema

La tasa de averías de un componente se expresa por averías por unidad de tiempo y varía con el tiempo de vida del componente.

Una representación usual para la tasa de averías de componentes de hardware es dada

por la curva de la bañera. En la Figura 2-11 pueden distinguirse tres fases:

- mortalidad infantil: componentes frágiles y mal fabricados
- vida útil: tasa de defectos constante
- envejecimiento: tasa de defectos creciente

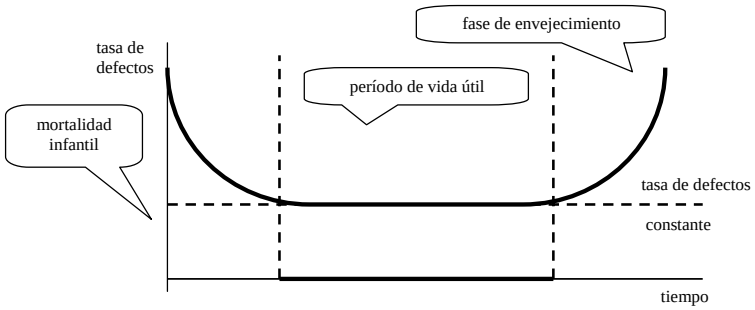

Figura 2-11: Curva de la bañera

Los componentes de hardware solo presentan tasa de defectos constante durante un período de tiempo llamado de vida útil, que sigue a una fase con tasa de defectos decreciente llamada de mortalidad infantil. Para acelerar la fase de mortalidad infantil, los fabricantes recurren a técnicas de *burn-in*, donde es efectuada la remoción de componentes frágiles por la colocación de los componentes en operación acelerada antes de colocarlos en el mercado o en el producto final.

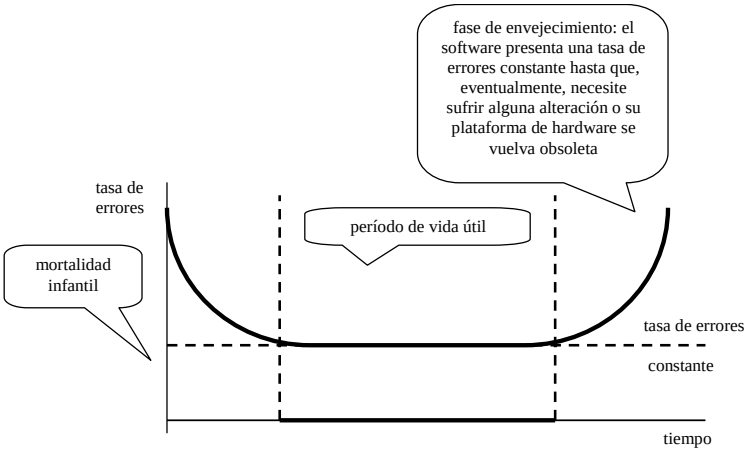

Figura 2-12: Curva de la bañera para *software*

Es cuestionable si la curva de la bañera puede ser aplicada también para componentes de *software*. Puede observase, sin embargo, que los componentes de software también presentan una fase de mortalidad infantil o tasa de errores alta al inicio de la fase de tests, que decrece rápidamente hasta la entrada en operación del *software*. Desde ese momento, el

software presenta una tasa de errores constante hasta que, eventualmente, necesite sufrir alguna alteración o su plataforma de *hardware* se vuelva obsoleta. En ese momento, la tasa de errores vuelve a crecer (Figura 2-12). Intencionalmente se menciona tasa de errores para *software* y no averías, porque error es el término usualmente empleado cuando se trata de programas incorrectos [68].

2.4. Tolerancia a fallos en *clusters* de computadores

Los *clusters* CoHNOW tal y como hemos visto, se construyen a partir de varios nodos procesadores independientes conectados por alguna tecnología de red. Esos sistemas se diferencian de las máquinas masivamente paralelas por el débil acoplamiento entre sus nodos, o sea, los elementos del *cluster* no tienen acceso una memoria común compartida. Otra característica que marca los *clusters* es inexistencia de un reloj común que pueda ser utilizado para ordenar los eventos. Toda la comunicación entre los nodos se da a través de paso de mensajes a través de canales de comunicación. Además, son generalmente construidos con elementos heterogéneos y asíncronos.

La tolerancia a fallos en CoHNOW, tiene el objetivo de proveer al *cluster* de una capacidad de operación continuada, intentando que sólo provoque una caída pequeña de prestaciones, tanto en la presencia de fallo como en la ausencia de fallos ocasionada por la prevención y la detección de fallos. A pesar de conocerse un buen conjunto de técnicas de tolerancia a fallos, su aplicación en los CoHNOW es muy compleja y sus resultados son muchas veces insatisfactorios debido al *overhead* introducido en el sistema [59].

Los CoHNOW presentan una redundancia física intrínseca debido a la existencia de muchos nodos de cómputo, que puede ser utilizada para el empleo de tolerancia a fallos. La ocurrencia de fallo en algún nodo procesador no tiene que significar necesariamente la interrupción del suministro del servicio. Una solución posible para esto, es que el sistema sea reconfigurado, utilizando solamente los nodos disponibles.

Con el objetivo de aumentar la garantía de funcionamiento de las CoHNOW, son esenciales, técnicas de prevención de fallos, detección de errores, diagnóstico, recuperación y re-configuración.

Para evitar la interrupción en el suministro del servicio, debido algún fallo en sus componentes, los fallos deben ser detectados lo más rápidamente posible: **latencia del fallo**. El nodo en que ha ocurrido el fallo debe ser identificado a través de diagnóstico apropiado y finalmente reparado o aislado a través de re-configuración del sistema. Esa re-configuración

se hace reasignando tareas y seleccionando caminos alternativos de comunicación entre los nodos.

Para detectar fallos y evitar que el *cluster* llegue al estado de **avería** (Figura 2-13), es necesario agregar algunos mensajes al esquema original. En la práctica, la identificación de las causas de fallos en *clusters* es muy complicada y fuertemente dependiente de la arquitectura. Los fallos en *clusters* pueden ser causados por problemas en el *hardware*,

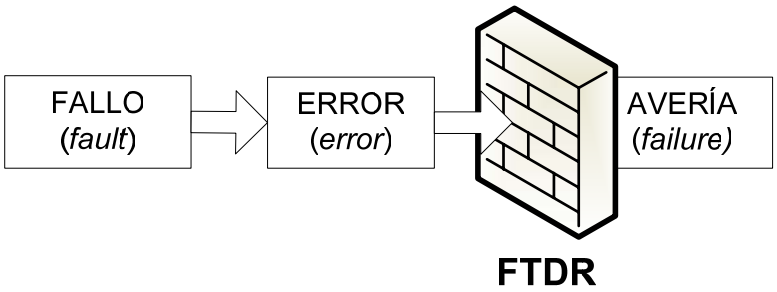

Figura 2-13: Fallo, error, FTDR y avería

sistema operativo (OS), aplicaciones, o por el propio *software* de tolerancia a fallos [66].

Como vemos es necesario utilizar alguna técnica o mecanismo que proteja al sistema, utilizamos **redundancia en el *software***, debemos considerar la prevención (generar redundancia), la detección (monitorizar), el diagnóstico (latencia del error) y la recuperación y reconfiguración del sistema (retornar el sistema a condiciones operativas razonables después un fallo) como un todo. Teniendo en cuenta, el equilibrio que se debe establecer entre la latencia y el *overhead* que introducen las técnicas de tolerancia a fallos.

2.5. Replicación de datos

Los recursos físicos utilizados en las CoHNOW son, por su propia naturaleza, susceptibles a fallos. Una amplia gama de fallos puede alcanzar estos recursos, pueden ser fallos en los computadores y/o en las redes. El uso de grandes redes, con decenas, cientos o miles de nodos, tiende a agravar este problema y las consecuencias de fallos pueden ser catastróficas.

Debido a necesidad de tolerar el fallo, diversas técnicas han sido desarrolladas para lograr que los CoHNOW se conviertan es sistemas con una alta disponibilidad, [69] sin que una solución haya sido consolidada como definitiva.

En los CoHNOW es posible la adopción de técnicas de distribución de réplicas de procesos y datos en diferentes nodos. El algoritmo de replicación es el centro de todo el sistema de Replicación de Datos y determina de forma decisiva las decisiones que deben ser

tomadas en el tratamiento de fallos y en la recuperación.

Un algoritmo de Replicación de Datos debe resolver principalmente el problema de mantenimiento de la **coherencia de las réplicas**, pues diversos procesos se ejecutan simultáneamente. La cuestión fundamental está en garantizar que todas las alteraciones generadas sean aplicadas a todas las réplicas. Los algoritmos deben también ser capaces de operar en el caso de que ocurran fallos, aún cuando el número de nodos se reduce.

La estrategia general a aplicar para conseguir tolerancia a fallos es aplicar redundancia. Nosotros utilizamos una redundancia temporal. Cuando una acción se realiza y algo falla, se ejecuta de nuevo, como transacciones atómicas. Consideramos una operación de replicación de datos como una transacción. Una transacción se puede definir como un conjunto de pasos que se deben ejecutar conjuntamente y siguiendo un orden, está basada en un conjunto de operaciones básicas atómicas (*read*, *write)* [12] [21] Una transacción es una agrupación de operaciones básicas que verifican un conjunto de propiedades:

- **A**tómica: o se realizan todas las operaciones de la transacción o no se realiza ninguna de ellas.
- **C**onsistencia: la transacción debe pasar al sistema de un estado consistente a otro.
- **I**ndependiente: las transacciones concurrentes no interfieren entre sí ni pueden "ver" estados intermedios de otras transacciones.
- **D**urable: una vez que la transacción se ha realizado sus resultados son permanentes. La durabilidad debe garantizar que una vez realizada la transacción sus efectos sean permanentes, incluso cuando existan fallos

Por eso se dice que las transacciones son **ACID**. Normalmente se tolera un modelo de fallos que incluye fallos de *crash* y de omisión en las comunicaciones.

En nuestro sistema algunos pasos los realizan distintos nodos, y debemos tener en cuenta que algún elementos pueden fallar. Los pasos se deben ejecutar conjuntamente, es decir la transacción no se puede dejar a medias, se debe abortar completamente, si algún elemento falla.

La Replicación de Datos, es una técnica importante para asegurar que el sistema esté disponible el máximo tiempo posible (*system availability*) [49] y se basa en que un conjunto de datos es copiado y asignado a más de un nodo.

La Replicación de Datos, así como todas las otras técnicas de tolerancia a fallos, añade *overhead*, o sea, consume recursos computacionales y de entrada y salida, reduciendo las

prestaciones del sistema como un todo [69].

2.6. Inyección de fallos

La Inyección de Fallos puede verse como un procedimiento para el testeo de la eficacia de técnicas de tolerancia a fallos. A través de la introducción controlada de fallos, el comportamiento de la técnica, bajo fallos, puede ser evaluado, o sea, puede ser determinado si la técnica permite al sistema, tolerar o no los fallos inyectados y cual es el *overhead* añadido por la cobertura de fallos diseñada (Figura 2-14).

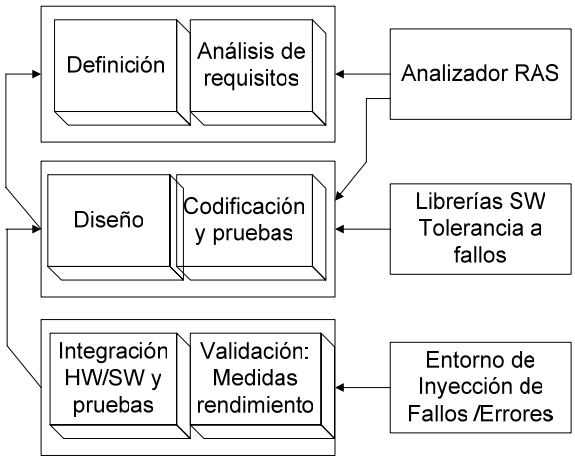

Figura 2-14: Inyección de Fallos - 1

En sistemas paralelos, específicamente en CoHNOW, la complejidad de los subsistemas de conexión y comunicación facilita la propagación de fallos y dificultan su detección. Sin control sobre el tipo y origen de los fallos se vuelve extremadamente difícil validar la implementación de tolerancia a fallos en dichos sistemas [4].

La Inyección de Fallos puede ser aplicada al sistema simulado o a sistemas físicos operando en su ambiente natural. Pueden ser *hardware* o *software*, las herramientas de Inyección de Fallos son específicas a un determinado sistema (Figura 2-15). Herramientas de Inyección de Fallos implementados por *software* no requieren *hardware* especial para ser aplicadas y poseen como ventajas: el bajo coste; la complejidad; y el esfuerzo de desarrollo que requiere. Son fácilmente adaptables a nuevas clases de fallos, y no presentan ningún problema con interferencias físicas. Sin embargo, la Inyección de Fallos implementado por *software* presenta algunos problemas:

- La capacidad para modelar ciertos tipos de fallos, todavía no ha sido totalmente desarrollada.

- La ejecución del software responsable de la Inyección de Fallos afecta las características de temporalización del sistema, lo que perjudica, por la intromisión o de forma que puede distorsionar la ejecución de funciones críticas en el tiempo.

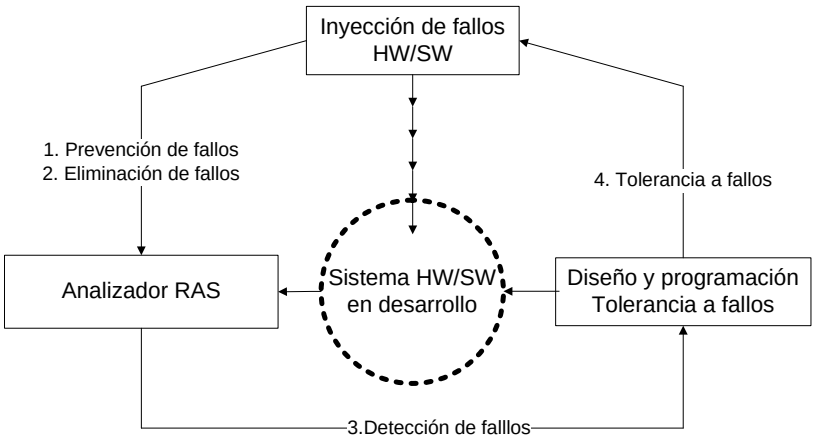

Figura 2-15: Inyección de fallos - 2

A pesar de estas desventajas, las inyección de fallos implementado por *software* ha suscitado el mayor interés de investigadores y desarrolladores, debido al control sobre los fallos y la ayuda que supone para realizar un análisis sistemático del sistema de tolerancia a fallos .

2.7. *Message Passing Interface* (MPI)

El MPI (*Message Passing Interface*) es usado en ciertas clases de máquinas paralelas, principalmente aquellas que cuentan con memoria distribuida. Existen muchas variaciones, pero, el concepto básico en el proceso de comunicación mediante paso de mensajes es bien entendido. Diferentes sistemas han demostrado que un sistema de paso de mensajes puede ser implementado eficientemente y con un alto grado de portabilidad.

La meta del MPI forum [50], es desarrollar un estándar para escribir programas que implementen el paso de mensajes. Por lo cual el interfaz intenta establecer para esto un estándar práctico, portable, eficiente y flexible.

En un entorno de comunicación con memoria distribuida en la cual las rutinas de paso de mensajes de nivel bajo, los beneficios de la estandarización son muy notorios. La principal ventaja al establecer un estándar para el paso de mensajes es la portabilidad y el ser relativamente fácil de utilizar.

De acuerdo con Chan [17] “La interface de Paso de Mensaje provee una base poderosa para construir programas paralelos.”

2.7.1. MPI y tolerancia a fallos

La correcta ejecución de una aplicación MPI está garantizada solo cuando todos los procesos constituyentes son finalizados con éxito. La "muerte" o el perdido de uno o más procesos, llevan a finalizar la aplicación, siendo éste el comportamiento estándar. Los usuarios necesitan reiniciar la aplicación manualmente. Esta visión simplista de finalización y reinicio no es capaz de proveer la tolerancia a fallos requerida por algunos ambientes puesto que las actuales implementaciones del MPI son inadecuadas en los siguientes aspectos:

- El estándar MPI y las implementaciones actuales del MPI asumen un modelo limitado de fallos.
- El MPI “asume” que la capa de comunicación es confiable y no provee mecanismos para manejar una capa no confiable.
- El estándar no provee métodos para manejar fallos en los nodos o pérdida de mensajes. Esto tipo de fallos hace que el MPI sea una alternativa insuficiente para sistemas construidos en ambientes inestables.
- En el MPI no hay una definición de fallos.
- El estándar MPI provee un ambiente con limitada notificación de fallos en forma de códigos de retorno de funciones. Fallos críticos tales como perdida de procesos, pueden tener preferencia sobre esos códigos de retorno, haciendo que el error aparezca y no presenta ningún código de retorno asociado al mismo.

Gropp [35], un de los creadores del MPI, afirma que el estándar presenta un número de indicaciones precisas, en la área de tolerancia a fallos, así como flexibilidad en la manipulación de fallos. Características como garantía de comunicación confiable, presencia de manejadores de fallos, y flexibilidad en la definición y extensión de un conjunto de errores son presentadas como factores que tornan el estándar MPI apropiado para la cuestión de la tolerancia a fallos.

El MPI Foro lanzó el estándar MPI-2 en 1998. MPI-2 consiste en extensiones en las áreas de creación y gestión de procesos, comunicaciones unilaterales, extensión de operaciones colectivas, y operaciones paralelas de Entrada/Salida. Una contribución

significativa del MPI-2 es el DPM (*Dynamic Process Management*), que permite a los usuarios crear y finalizar procesos adicionales en demanda. DPM puede ser usado para compensar la pérdida de un proceso, pero la falta de detección y recuperación compromete la garantía de funcionamiento.

Los estudios de mecanismos de tolerancia a fallos en sistemas MPI, de acuerdo con Bolsica en [13], se han concentrado en diferentes soluciones, pero pueden ser clasificadas de acuerdo con dos criterios:

El nivel del *software* donde el mecanismo de tolerancia a fallos se encuentra.

Las técnicas utilizadas para tolerar fallos.

Aún de acuerdo con Bolsica en [13], existen tres niveles principales en el contexto del MPI:

Alto nivel: donde el ambiente del usuario puede asegurar la tolerancia a fallo para una aplicación MPI a través de la re-inicialización de la aplicación desde un punto coherente anterior.

Bajo nivel: donde el MPI puede apurar mecanismos para informar a la aplicación que ocurrió un fallo y permitir que ésta lo trate.

Más bajo nivel: donde la capa de comunicación en la cual el MPI es construida se hace responsable por de la tolerancia y debe tratarla de forma transparente.

Dos técnicas principales están siendo utilizadas para permitir la transparencia del fallo:

Una solución está basada en Replicación de Datos, que consiste en guardar el histórico de los procesamientos y replicarlos, para a partir de allí poder ejecutar la aplicación nuevamente después de la ocurrencia del fallo.

La otra solución está basada en un conjunto de *checkpoints* global, que permite que la aplicación MPI sea reiniciada desde el último punto coherente.

Cuando el ambiente MPI es iniciado, todos los procesos involucrados pertenecen a un mismo *communicator* (MPI_COMM_WORLD).

Cada uno de esos procesos recibe un identificador único que los diferencian dentro del *communicator* (Figura 2-16). Solo procesos pertenecientes a un mismo *communicator* pueden intercambiar mensajes entre sí, creándose así un contexto de comunicación entre ellos.

El *communicator* es una estructura de datos definida en el estándar MPI: un proceso solo puede comunicarse directamente con otro, si ambos pertenecen a un mismo *communicator* [35] [50].

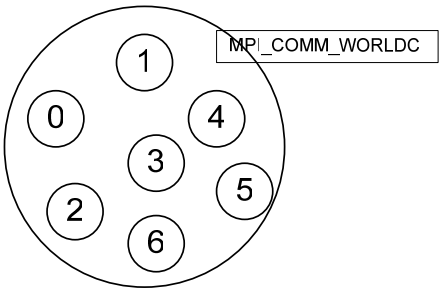

Figura 2-16: Identificador dentro del *communicator*

El comportamiento patrón de una aplicación MPI, cuando ocurre un error, es matar todos los procesos pertenecientes al mismo *communicator*. Este comportamiento base llevó a afirmaciones de que el estándar MPI no es tolerante a fallos [35].

Pero si tenemos en cuenta que existe una estructura en el MPI para tratamiento de errores (*error handler*) que está asociada a un *communicator*, así, cuando frente a la ocurrencia de un error, es llamado el manejador de errores. El manejador de errores estándar, asociado al MPI_COMM_WORLD, es llamado: MPI_ERRORS_ARE_FATAL y es el responsable de este comportamiento.

El estándar MPI especifica otro manejador de errores, el MPI_ERRORS_RETURN, que, puede ser anexado a cualquier *communicator*, incluso el MPI_COMM_WORLD, permite el retorno de un código de error, en el caso de la ocurrencia de éstos.

Los fallos que interesan para FTDR son los **fallos de *hardware*.** Los errores hasta aquí referidos, son errores de aplicación, o sea, errores “empotrados” por el programador, a ejemplo de *ranks* y *tags* inválidos, *communicators* inexistentes, etc. Esa clase de errores, suportada por el MPI, tal como explicitado por Batchu [9] constituye un modelo limitado: “El MPI posee un conjunto de fallos que engloba parámetros incorrectos en llamadas de funciones y errores de recurso. Este soporte de fallos está incompleto e insuficiente para ambientes inestables” [9]. Sin embargo, el estándar MPI ofrece funciones que permiten que los usuarios creen sus propios manejadores de errores, conforme sus necesidades [51]. La función que permite esto es llamada de MPI_Errhandler_create.

2.8. Trabajos relacionados

La principal preocupación de los programadores y usuarios que utilizan *clusters* de computadores para ejecutar aplicaciones largas (tiempo de ejecución largo), con mucho computo, estaba relacionada al principio con los aspectos de prestacionales. Sin embargo, con el transcurrir del tiempo, surgió la necesidad de buscar “algo más”, pues nada se

adelantaba con tener soluciones cada vez más rápida, que acababan perdiéndose por fallos.

La tolerancia a fallos es un aspecto crítico para aplicaciones a gran escala, ya que aquellas ejecuciones que pueden tardar del orden de varios días o semanas para ofrecer resultados deben tener la posibilidad de manejar cierto tipo de fallos del sistema o de alguna tarea de la aplicación.

Sin la capacidad de detectar fallos y recuperarse de estos, dichas ejecuciones pueden no llegar a completarse.

De cualquier forma, debería haber algún modo de detectar y responder automáticamente a ciertos fallos del sistema o al menos ofrecer cierta información al usuario en el caso de producirse un fallo.

Existen varios grupos investigando el problema de tolerancia a fallos de las aplicaciones ejecutadas en *clusters* de computadores, que se utilizan del paradigma *Master/Worker*, y que utilizan MPI como forma de comunicación, teniendo en cuenta que MPI sacrifica en cierto modo la tolerancia a fallos en favor de la eficiencia.

La comunidad científica trabaja en diferentes soluciones: **transparentes**, **semitransparentes** y no **transparentes** al usuario.

2.8.1. Algunas soluciones de tolerancia a fallos transparentes al usuario

Algunos investigadores critican el estándar MPI [9] [14] [35] [50] [51], afirmando que éste no posee características de tolerancia a fallos. Por esto muchos grupos de investigación se han centrado en proponer modificaciones. Otros investigadores para resolver el problema de la tolerancia a fallos utilizan otras técnicas para lograr que los sistemas paralelos sean tolerantes a fallos.

CoCheck [62] fue la primera tentativa para hacer aplicaciones MPI tolerante al fallo y fue con la utilización de *checkpointing* y *roll back*. De allí, surgieron otras tentativas (Starfish [1], MPI-FT [47], MPICH-V [13], Egida [55]), algunas de las cuales detallamos a continuación.

2.8.2. Librerías para tolerancia a fallos

Una forma muy extendida para proteger aplicaciones paralelas distribuidas es implementar alguna estrategia de *checkpoint* [16] [24] y *log* de mensajes. Con *checkpoint*, que replica el estado del programa, el principal problema de usar esta técnica de tolerancia a fallo es el coste (volumen de información, complejidad y overhead alto y poco predecible).

CLIP (*Checkpointing Libraries for the Intel Paragon*) [19] es una librería que permite la administración de los *checkpoints* en el nivel de usuario y puede ser utilizada en códigos MPI (*Message Passing Interface*) [50] para garantizar un sistema semitransparente de *checkpoint*. Funciona a través de la llamada a sus funciones desde el código de la aplicación y ésta no necesita preocuparse por la gestión de estados.

El CLIP puede hacer *checkpoints* en programas escritos tanto en NX (*Intel's NX message-passing protocol*) [53] como en MPI; ambos son plataformas *message passing* "de facto" para computación de alto rendimiento. Para usar el CLIP, el usuario debe conectar (*link*) la librería CLIP con su aplicación. Además, el usuario debe insertar una o más llamadas (*calls*) a subrutinas de en su código especificando cuando deben realizarse los *checkpoints*. Cuando se ejecuta el código, el CLIP ejecuta *checkpoints* periódicos en el programa y envía el estado (*state*) a disco. Si por alguna razón existe un fallo de *hardware* o *software* y la ejecución del programa termina prematuramente, puede ser reiniciado desde el archivo de *checkpoint* previamente guardado en disco [19].

El CLIP es considerado semitransparente porque el usuario debe hacer algunas modificaciones en su código para definir los lugares de *checkpoints*. Sin embargo, es más transparente que otras herramientas de *checkpointing*, que exigen que el usuario reconstruya el programa, estado de los mensajes y recuperación del sistema de archivo. Estas tareas son automáticamente manejadas por el CLIP [19].

2.8.3. Tolerancia a fallos responsabilidad del programador de aplicaciones

El FT-MPI [25] deja que la aplicación sea responsable del proceso de administración de la recuperación de los fallos [16]. Cuando un fallo es detectado, todos los procesos de comunicación son informados. Esa información es transmitida a la aplicación a través de un valor de retorno de una llamada MPI. La aplicación puede tomar decisiones y ejecutar una acción correctiva. La gran ventaja de ese proceso es la velocidad (ya que no existe *checkpoint*), sin embargo exige más del programador (programador experto) que deberá ser responsable por la transparencia al usuario del proceso. La propuesta del FT-MPI es construir una implementación MPI tolerante al fallo, ofreciendo al desarrollador de aplicaciones una serie de opciones de recuperación, que hacen con que las mismas retornen a un estado previamente *checkpointed* [26].

2.8.4. Comparación de los distintos modelos de tolerancia a fallos

El FT-MPI tiene menos *overhead* si lo comparamos con Starfish, MPI-FT y MPICH-V, y además mejora las prestaciones [26]. Por otro lado, estos beneficios traen algunas consecuencias, una aplicación que utiliza FT-MPI tiene que ser diseñada para que aproveche las características de tolerancia a fallo, aunque éste trabajo extra pueda ser trivial, es dependiente de la estructura de la aplicación, por lo tanto el programador tiene que ser un experto.

Las Tabla 2-3, Tabla 2-4, Tabla 2-5, Tabla 2-6, Tabla 2-7, Tabla 2-8 y Tabla 2-9, muestran algunas características de algunos modelos de tolerancia a fallos.

Tabla 2-3: Modelos de tolerancia a fallos -1

Solución	Objetivo	Ideas más importantes	Como ellos hacen
Cocheck (1996)	*Checkpointing* para aplicaciones paralelas	Implementado en el nivel de ejecución de la aplicación, funciona en el tope de la librería *messages passing.* Los procedimientos de *checkpoint* y *rollback* son dirigidos por un coordinador centralizado.	*Checkpoint Framework*

Tabla 2-4: Modelos de tolerancia a fallos -2

Solución	Objetivo	Ideas más importantes	Como ellos hacen
Clip (1997)	Es una biblioteca *user-level* que provee checkpointing semitransparente para programas paralelos que son ejecutados en de computadores múltiple (*multi-computers*).	*Checkpoint* semitransparente. El usuario agrega llamadas para *checkpoin*t en su código, pero, no necesita administrar el estado del programa.	*Checkpoint* API

Tabla 2-5: Modelos de tolerancia a fallos -3

Solución	Objetivo	Ideas más importantes	Como ellos hacen
Egida (1999)	Conjunto (kit) de herramienta orientada a objeto, proyectada para soportar *rollback recovery* transparente.	Implementa la detección de fallo y recuperación en una capa de bajo nivel (*low-level layer*). Restringido para el MPICH.	*Log* *Checkpoint* API

Tabla 2-6: Modelos de tolerancia a fallos -4

Solución	Objetivo	Ideas más importantes	Como ellos hacen
Starfish (1999)	Ambiente para ejecución dinámica (y estática) de programas MPI-2 en COW.	Posee una API que permite al usuario controlar el proceso de recuperación. Restringido para el MPI-2.	*Checkpoint* *Framework* API

Tabla 2-7: Modelos de tolerancia a fallos -5

Solución	Objetivo	Ideas más importantes	Como ellos hacen
FT-MPI (2000)	Completamente controlado por la aplicación.	La aplicación es responsable por la administración del proceso de recuperación de los fallos. Cuando un fallo es detectado, todos los procesos de un *Communicator* son informados sobre el fallo. Estas informaciones son transmitidas para que la aplicación retorne el valor del *MPI call*.	Trabaja con fallo del *MPI Communicator* y permite que la aplicación administre la recuperación. API

Tabla 2-8: Modelos de tolerancia a fallos -6

Solución	Objetivo	Ideas más importantes	Como ellos hacen
MPI-FT (2000)	Detección y recuperación de fallos (*failure recovery*).	Implementa la detección de fallos en el nivel del MPI y recupera en el nivel de ejecución (*runtime level*). Varios sensores son utilizados para detectar fallo de aplicación, sean ellas, del MPI, de la red y / o del sistema operativo.	*Pessimistic log Communication library*

Tabla 2-9: Modelos de tolerancia a fallos -7

Solución	Objetivo	Ideas más importantes	Como ellos hacen
MPICH-V (2002)	Ambiente MPI tolerante a fallo, soportado en *checkpoint / rollback* y *log* de mensaje distribuido.	La biblioteca implementa subrutinas de comunicación originada en el MPICH. Restringido para el MPICH.	Librería de comunicación basada no MPICH

2.8.5. Replicación de Datos X *checkpoint-recovery*

Weissman [69] [70], explora dos opciones para alcanzar tolerancia a fallo para una clase común de aplicaciones paralelos, SPMD. Compara cuantitativamente *checkpoint-recovery* y Replicación de Datos como un medio de alcanzar tolerancia a fallo. Los resultados experimentales obtenidos sugieren que *checkpoint-recovery* puede ser una opción preferible para problemas con poco computo o poco tiempo de ejecución y la replicación de datos y en especial *wide-area* es preferible para problemas con mucho computo o mucho tiempo de ejecución, precisamente este es el tipo de problemas que se utilizan de ambiente CoHNOW geográficamente distribuido. Los resultados también muestran que es posible predecir el

overhead en los dos métodos.

Según Saito, Y. y Shapiro [56], Replicación de Datos es una tecnología clave en sistemas distribuidos, pues mejora significativamente la disponibilidad (*availability*) y las prestaciones de los sistemas.

Replicación de Datos consiste en mantener copias múltiplas de datos, llamadas réplicas, en nodos separados. Es una tecnología importante para sistemas distribuidos, pues mejora la disponibilidad permitiendo acceso a los datos en caso de fallos en determinado nodo [56].

2.9. Conclusiones

Se ha presentado en el capitulo conceptos sobre arquitectura de computadores paralelos, tolerancia a fallos, Replicación de Datos, tolerancia a fallos en *clusters* de computadores, *Message Passing Interface* (MPI) y validación de modelos y también mostró algunas áreas de aplicación de computadores tolerantes a fallos.

Prácticamente todos los ejemplos citados toleran errores provocados por fallos de *hardware*. Es fácil de imaginar que con la utilización de componentes cada vez más fiables y *software* cada vez más complejo, cobren importancia errores predominantemente debidos a fallos de *software*. Mecanismos contra fallos múltiples poco probables o contra fallos de *software,* es raro que estén disponibles debido al elevado coste asociado.

La tolerancia a fallos dispone muchas técnicas que permiten aumentar la calidad de servicios en sistemas computacionales, no garantiza comportamiento correcto en la presencia de cualquier tipo de fallo. Las técnicas empleadas para garantizar el funcionamiento de la aplicación involucran algún grado de redundancia y pueden generar sistemas mayores y más caros.

El desarrollador debe saber elegir el modelo de tolerancia a fallo equilibrando coste, latencia, menor *overhead*, escalabilidad para suplir las exigencias de garantía de funcionamiento del sistema y saber desarrollar los mecanismos complementarios necesarios. Debe conocer las técnicas de tolerancia a fallos que pueden ser utilizadas.

Algunos investigadores critican el estándar MPI [9] [14] [35] [50] [51], afirmando que éste no posee características de tolerancia a fallos. Debido a que eso, ellos lo modificaron o se utilizaron de otras técnicas para tornar algunos sistemas paralelos tolerantes a fallos.

Batchu [9], uno de los creadores del MPI-FT, que es una técnica utilizada para proveer

tolerancia a fallos, en la distribución LAM-MPI enfatiza que el MPI es limitado en cuanto a tolerar fallo ya que las principales metas del MPI es proveer altas prestaciones y portabilidad. La incorporación de características de tolerancia a fallo implicaría un *overhead* adicional, lo que comprometería una de las metas del MPI.

La tolerancia a fallos plantea diferentes desafíos que no han sido solucionados, no es un área de investigación completamente dominada y a pesar de ser relativamente antigua, aún necesita mucho trabajo de investigación.

Capitulo 3

Arquitectura multicluster

Este capitulo presenta en detalle la arquitectura propuesta para poder trabajar con *clusters* geográficamente distribuidos, especificando como en dicha arquitectura se resuelve el problema de los fallos transitorios de comunicación.

Muestra los problemas relacionados con la heterogeneidad de los *clusters* y la asignación de tareas estática y granularidad diferentes a *clusters* distantes.

Plantea el problema de los fallos en nodos de cómputo y de la necesidad del sistema de tolerancia a fallos.

3.1. Introducción

Para poder trabajar con *clusters* geográficamente distribuidos es necesario resaltar diferentes aspectos de la arquitectura, como el tipo de nodos de cómputo a utilizar, sus componentes, prestaciones, disponibilidad, red de interconexión y el nivel de paralelismo utilizado. La arquitectura propuesta está pensada para trabajar con *clusters* geográficamente distribuidos, utiliza computadores tipo PC disponibles comercialmente, es un sistema heterogéneo, los computadores en cada ubicación forman un *cluster*, con computadores comunicados a través de una red local dedicada y disponen de un nodo dedicado a la comunicación con los otros cluster, para dicha comunicación utilizan Internet.

El paradigma de programación define el proyecto de la aplicación paralela. Usando un paradigma específico, determinará como debe ser implementada la aplicación.

Para elegir un paradigma de programación paralela, el programador debe observar los recursos de computacionales disponibles y también las características de la aplicación [14]. En nuestro caso se utiliza el paradigma o modelo de ejecución *Master/Worker* [14].

3.2. Arquitectura propuesta

Para desarrollar el modelo de tolerancia a fallos para CoHNOW geográficamente distribuidos se ha utilizado una arquitectura, *Master/Worker* jerárquica (Figura 3-1).

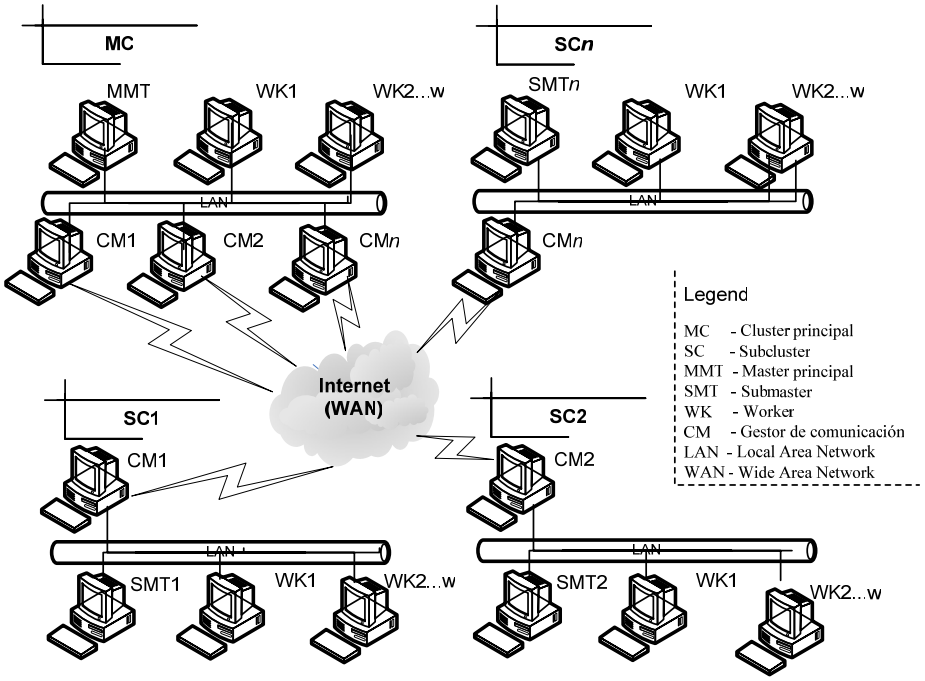

Figura 3-1: Arquitectura CoHNOW

Esta arquitectura además de la heterogeneidad de los nodos de computo presenta una fuerte heterogeneidad a nivel de comunicación, por un lado tenemos una red local entre *cluster* (LAN) que puede ser de 10/100 Mbits que permite comunicar localmente las máquinas de cada uno de los *cluster* y para la comunicación entre *cluster* se usa Internet (WAN).

Esta arquitectura, además de los nodos *Master* y *Worker,* interconectados a través de una red local dedicada, para conectar *cluster* geográficamente distribuidos utiliza un elemento adicional que gestiona la comunicación (CM) entre el *cluster* principal y los Subcluster en una organización *Master/Worker,* es decir, los Subclusters no se pueden comunicar entre ellos, sólo se pueden comunicar con el *cluster* principal. Esto permite considerar desde el *cluster* principal (*Master*) al resto de Subclusters como *Workers* con una potencia de cómputo elevada, pero que también añaden una latencia elevada, de forma que al *Master* se comunica con ellos a través del gestor de comunicaciones comos si fueran "*Workers* especiales".

En esta arquitectura tenemos que considerar:

- Un *cluster* principal (MC), también llamado *clusters* locales o HNOW locales, con las siguientes funcionalidades lógicas principales asignadas a sus nodos: *Master* Principal (MMT), *Workers* (WK) y Gestores de comunicación (CM) (tantos como subclusters).
- Los Subclusters que tendrán su propio Submaster (SMT), *Workers* (WK) y Gestores de comunicación (CM).

3.2.1. *Master* principal

El *Master* principal (MMT), es el *Master* (MT) del CoHNOW, desde donde a computación es arrancada (*started*) y donde inicialmente residen todos los datos a ser procesados. Distribuye la carga de trabajo, entre los Subclusters y sus propios *Workers* locales. Una vez terminado el cómputo es donde quedan los resultados. Vemos que es un elemento crítico del sistema que necesitará una protección específica.

3.2.2. Submaster

Los Submaster (SMT) representan los *Masters* (MT) de los Subcluster (SC), los SC también son llamados *clusters* remotos o HNOW remotas. La computación de los SMT es un subconjunto do problema del MMT, recibe los datos a través del Communication

Manager o Gestor de Comunicación que está conectado a la red de comunicación extendida (WAN). El Submaster, es el encargado de distribuir las tareas entre los *Workers* del Subcluster y recolectar los resultados para reenviarlos al *Master* principal. Lleva el control del trabajo realizado, y almacena los resultados obtenidos por los *Workers*. Puesto que hace de *Master* del Subcluster, es el único nodo con este papel y necesitará una protección específica similar a la de *Master* principal.

3.2.3. *Workers*

Los *Workers* (WK) en todos los *clusters* procesan los datos computacionales. Son máquinas heterogéneas, con distintas capacidad de cómputo y memoria. Sólo se comunica con el *Master*. El *Master* envía una tarea y recibe los resultados. Esta distribución de tareas puede ser estática o dinámica.

Para solapar cómputo con comunicación, cada *Worker* puede usar un doble *buffer*, mientras está computando los datos almacenados en un *buffer*, el *Master* está enviando los datos para el siguiente computo. La primera vez que reparte tareas el *Master* envía dos a cada uno de los *Workers* y luego dinámicamente, cada vez que recibe resultados vuelve a enviar una nueva tarea, mientras el *Worker* computa la que tiene en el otro *buffer*.

La razón entre cómputo y comunicación es conocida como granularidad. Es importante conocer esta relación para realizar una distribución de tareas adecuada.

Debido a la redundancia intrínseca de un sistema *Master/Worker*, en un *cluster* existen varios *Workers*, el trabajo de un *Worker* que falla puede ser fácilmente asumido por otro *Worker*

Para el *Master* principal un Subcluster es considerado como un *Worker* de altas prestaciones, es decir con una gran capacidad de cómputo (suma de la capacidad de cómputo de todos los Subworkers de dicho *cluster*) y una elevada latencia de comunicación, con tiempo de comunicación poco predecible, debido a que está conectado a través de la WAN.

Buscamos un paralelismo de grano grueso para mejorar la razón cómputo comunicación, pero teniendo en cuenta el balanceo de carga, por esto, establecemos un grano diferente para los *Workers* en el *cluster* local y para el Subcluster

Por ello se le asigna un trabajo o grupo de tareas de forma estática teniendo en cuenta la razón computo/comunicación.

3.2.4. Gestores de comunicación

Para separar la Red de Área Local (LAN) del tráfico de la WAN y para proveer comunicación transparente y confiable entre el *Master* Principal y los Submaster, la arquitectura incorpora unos Gestores de Comunicación (CM): en el *Cluster* Principal (MC), para cada Subcluster existe un CM (máquina física o lógica), En cada Subcluster existe un Gestor de Comunicación para la comunicación del Subcluster con el *cluster* principal.

La arquitectura con Gestores de Comunicación fue desarrollada con el objetivo de obtener una menor latencia de conexión entre *clusters* remotos comunicados a través de Internet y dejar que los nodos trabajadores utilicen el máximo de su potencia de computo en un CoHNOW [3] [30] [31] [60].

Estos Gestores de Comunicación son transparentes a la aplicación del usuario. El conocimiento del funcionamiento del sistema permite una mejor sintonización de la aplicación con la arquitectura propuesta.

En esa solución, los HNOWS no se comunican entre sí utilizando primitivas de comunicación MPI, sino que utilizan una conexión vía *socket* TCP. Para aumentar la velocidad de comunicación utiliza varios *threads* de comunicación entre ambos Gestores de Comunicación, esto requiere diseñar un control de mensajes enviados y recibidos.

Los Gestores de Comunicación fueran creados para proveer capacidad de *buffering* para el Submaster (para mejorar el flujo de datos entre los HNOWS), desacoplando la red local de la red lenta que comunica los *clusters*, agrupando datos (para mejorar la granularidad o relación computo/comunicación), controlando los bloques que fueron enviados/recibidos para/de los HNOW remotas, y haciendo la reasignación de datos remotos, cuando comprometan las prestaciones del sistema. También se encargan de soportar los fallos intermitentes de la red, llevando un control de los paquetes enviados y recibidos.

Dentro de cada *cluster* existe una aplicación paralela que usa primitivas de comunicación MPI. Los Gestores de Comunicación son responsables de mantener una conexión local MPI y una conexión remota a través de *sockets* TCP, actuando como un "portal" para todos los datos que entran y que salen (lógica y físicamente).

El *Master* Principal y Submaster solo se comunican con su Gestor de Comunicación, que tiene dos tarjetas de red, para aislar las comunicaciones con ambas redes, es el responsable de manejar el tráfico local y remoto.

Todos los nodos del *cluster* disponen de dos tarjetas de red, de forma que cualquier nodo del *cluster* pueda asumir cualesquier funcionalidad, sea *Master* Principal, Submaster,

Worker o Gestor de Comunicación.

Con los Gestores de Comunicación se ha logrado mejorar el rendimiento global del sistema. Permite gestionar mejor la mejor relación entre cómputo y comunicación, combinada. También pudimos notar la ventaja de desacoplar el control de la comunicación de los nodos *Master* Principal y Submaster, ya que sin los gestores de comunicación los *Workers* permanecían tiempo inactivos aguardando el *Master* Principal o Submaster para atender sus pedidos.

Los Gestores de Comunicación permitirán incrementar la carga de trabajo y mejor distribuirla para un CoHNOW.

3.3. Balanceo de carga

Los CoHNOW, están formados por nodos heterogéneos con diferentes tipos de configuraciones, como velocidad de procesamiento, cantidad de memoria y/o arquitectura. Como las máquinas son distintas, es necesaria una distribución proporcional del trabajo para todos los nodos integrantes del cluster.

Un CoHNOW puede ser considerado como un segundo nivel HNOW. Un CoHNOW es formada por una colección de HNOW, donde cada de ellos puede estar geográficamente separada e interconectada por una WAN pública, o sea, por Internet. Ese tipo de maquina tiene todas las particularidades de una simple HNOW, más los aspectos relacionados al uso de una interconexión de red con latencia inestable y anchura de banda (*bandwidth*) variable, tal cual a Internet [14] [60].

El balanceo de carga es una cuestión importante en los CoHNOW y buscamos la mejor manera de distribuir los procesos y los datos entre los nodos que componen la Máquina Paralela Virtual. Esta distribución de carga depende de diversos factores como la homogeneidad de los granos de paralelización, la velocidad y prestaciones de las máquinas.

Para distribuir o configurar los procesos, debemos tener en cuenta que los *Master* gestionan básicamente E/S, los *Workers* requieren la mejor potencia de cómputo y los Gestores de Comunicación necesitan memoria para gestionar adecuadamente la comunicación con los nodos remotos.

Un abordaje simple para el problema es la asignación estática de recursos. En este abordaje, el usuario elabora alguna heurística de asignación de tareas que busca

balancear la carga entre los nodos presentes. La gran ventaja de este método es su facilidad de proyecto e implementación. Puede ser la solución adoptada para problemas cuya solución no sea crítica al funcionamiento de un sistema y todo cuanto se quiere es obtener un *speedup* utilizando la capacidad ociosa de la red actual.

En el abordaje dinámico del balanceo, a pesar de la dificultad relativa para implementación, la independencia de la máquina paralela se vuelve evidente pues a cada problema será realizado el calibrado de las cargas posibilitando la retirada o entrada de máquinas en el sistema.

De forma sucinta, el balanceo de carga puede ser visto como el proceso de dividir, de forma equitativa, las tareas y datos en el sistema de computación paralela con el objetivo de mejorar el tiempo de ejecución.

La caracterización del *cluster* es un paso importante para obtener todas las informaciones necesarias. Es necesario hacer la caracterización de los nodos dentro de cada uno de los *clusters*.

Para caracterizar los nodos, se puede utilizar un algoritmo de multiplicación de matrices [58]. Se puede utilizar de distintos tamaños de matrices para medir el tiempo de ejecución en cada nodo del cluster. El acceso a la memoria caché crece a medida que se aumenta el tamaño de la matriz. Cuando la capacidad de memoria principal no es tenida en cuenta, pueden ocurrir muchos errores de páginas (*page faults*) y el disco es frecuentemente accedido (*swapping*) para intercambiar información, esto repercute negativamente en el tiempo de ejecución [31].

Estas información que proviene de la caracterización, es importante para determinar el rendimiento de cada Subcluster, para calcular los MFLOPS (*Millions of Floating Point Operations per Second*) y el grado de heterogeneidad (Figura 3-2) de cada de ellos, para construir una base para determinar las políticas de distribución de datos.

$$Gh = texecR / texecL$$

Figura 3-2: Grado de heterogeneidad

El grado de heterogeneidad (*Gh)*, es calculado entre el tiempo de ejecución obtenido por el *clusters* más rápido (*texecR*) y el tiempo de ejecución obtenido por el *cluster* más lento (*texecL*), cuando ambos ejecutan el mismo algoritmo.

El grado de heterogeneidad (*Gh*) es un índice que representa la diferencia de capacidades de procesamiento entre nodos o *clusters*. Esta diferencia la tendremos en cuenta para la asignación estática de carga de trabajo entre los diferentes *clusters*.

En la caracterización de la red WAN (Internet) se debe analizar las comunicaciones entre los diferentes Subclusters. Para ello, se puede enviar paquetes utilizando la primitiva de sistema "*ping pong*". Estas pruebas se realizan para colectar datos relativos al comportamiento de la red pública en las diversas horas del día, observar las posibles pérdidas de paquetes y conocer la relación pérdida y tamaño del paquete, teniendo en cuenta la hora del día.

Para adaptar la aplicación al entorno multicluster es conveniente considerar la nueva heterogeneidad que añade la organización jerárquica del *Master/Worker*, ya que los Subclusters se comportan como *Workers* con unas características prestacionales muy diferentes, una potencia de cómputo prácticamente igual a la suma de la potencia de todos sus *Worker* y una latencia en la comunicación, muy alta y además impredecible, dependiente del funcionamiento de la red WAN no dedicada.

El programador, para hacer una distribución de la carga de trabajo debe considerar dos granularidades, para que la relación computo/comunicación sea adecuada: por ello normalmente se especifica la carga de trabajo para una tarea de "*Worker*", y la carga de trabajo total que se tiene para distribuir para cada uno de los Subcluster, o sea, es conveniente que el programador, adapte la aplicación a la arquitectura *Master/Worker* jerárquico.

La distribución de las tareas a los *Workers* en cada uno de los *clusters* se realiza de modo dinámico, mientras que la distribución de tareas al Subcluster se realiza de una forma semiéstática, se le asigna y se le envía un trabajo inicialmente y se redistribuye al final de la aplicación si es necesario.

3.4. Entorno utilizado para la validación experimental

El entorno de pruebas elegido (*clusters*) es representado por dos laboratorios de investigación en computación paralela, geográficamente distribuidos uno de ellos localizado en el C*entro Baiano de Computação de Alto Desempenho* (CEBACAD), *Universidade Católica do Salvador* (UCSAL), Brasil y otro localizado en el *Departament d'Arquitectura de Computadors i Sistemes Operatius* (DACSO), *Escola Tècnica Superior d'Enginyeria* (ETSE), *Universitat Autònoma de Barcelona* (UAB),

España, constituyéndose en un CoHNOW.

Los *clusters* de los laboratorios de investigación del entorno están compuestos por los nodos presentados en la Tabla 3-1 y Tabla 3-2.

Tabla 3-1: *Cluster* local UAB

				Función en el cluster	
Tag	**Nombre del Modelo**	**CPU (Info)**	**Mem (Info)**	**Homogéneo**	**Heterogéneo**
aoquir1	Pentium III (Katmai)	502,500	126288 KB	WK/PMT	WK
aoquir2	Pentium III (Katmai)	501,200	126288 KB	MMT	CM
aoquir3	AMD Athlon(TM) XP 2600	1.905,510	255724 KB	-	WK/PMT
aoquir7	Pentium III (Katmai)	451,160	126340 KB	WK/PMT	-
aoquir8	Intel(R) Pentium(R) 4 CPU 2.60GHz	2.594,282	248196 KB	-	WK/PMT
aoquir10	Pentium III (Katmai)	451,070	126340 KB	WK	-
aoquir11	Pentium III (Katmai)	501,350	126288 KB	WK	-
aoquir12	AMD Athlon(TM) XP 2600+	1.905,510	256180 KB	CM	MMT

Tabla 3-2: *Cluster* remoto UCSAL

				Función en el cluster	
Tag	**Nombre del Modelo**	**CPU (Info)**	**Mem (Info)**	**Homogéneo**	**Heterogéneo**
infoquir1	Pentium 75 – 200	166,200	29340 KB	WK	WK
infoquir2	Pentium 75 – 200	166,200	29340 KB	CM	CM
infoquir3	Pentium III (Katmai)	499,160	61356 KB	WK/PMT	WK/PMT
infoquir5	Pentium 75 – 200	132,870	29340 KB	SMT	SMT
infoquir6	Pentium 75 – 200	132,870	29340 KB	WK/PMT	WK/PMT
infoquir7	Pentium 75 – 200	132,870	29340 KB	WK	WK
infoquir8	Pentium 75 - 200	132,870	21252 KB	-	-

Las características de las máquinas de este entorno se muestran en las Figura 3-3 y Figura 3-4. Para realizar las pruebas se han configurado *clusters* con más o menos heterogeneidad entre los nodos seleccionados.

Los programas utilizados en estos experimentos, fueron compilados (*gcc 3.2.3*) y ejecutados bajo el Sistema Operativo *GNU/Linux, Kernel 2.4.20*.

La LAN utilizada en cada uno de los *clusters* sigue el estándar Ethernet de 10 Mbps. Del lado de la UCSAL (peor condición), la conexión con el Internet (WAN) fue hecha a través de un link no dedicado de 512 Kbps.

Por cuestión de seguridad, el acceso al *cluster* es realizado a través de conexiones SSH. El proceso de distribución de trabajos en el *cluster* es realizado a través del SCP del SSH (*Secure Shell*) para cada estación trabajadora.

La comunicación a través de Internet en el entorno de desarrollo se realiza a través de

los Gestores de Comunicación, que utilizan *sockets.* Esta forma de comunicación, proporciona mayor garantía de funcionamiento para un CoHNOW [31].

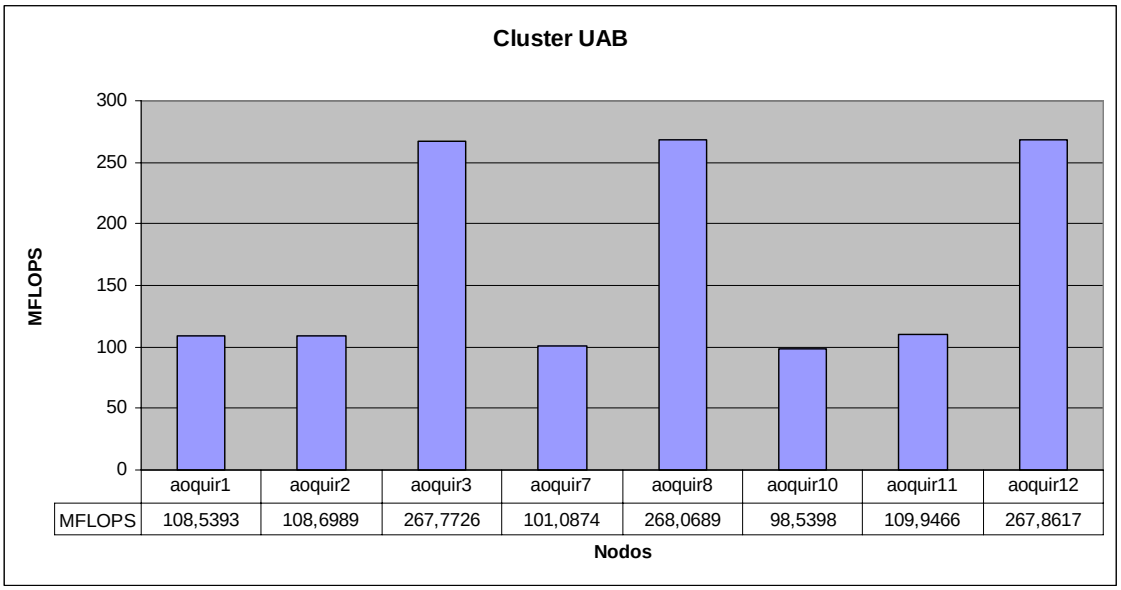

Figura 3-3: *Cluster* local UAB - Características

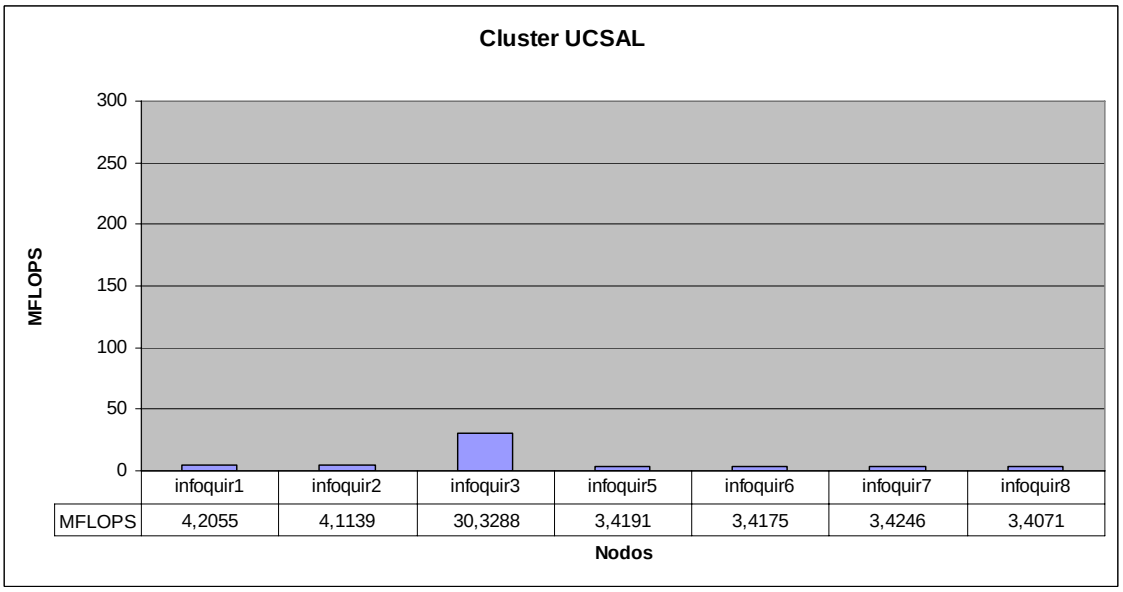

Figura 3-4: *Cluster* remoto UCSAL - Características

Los resultados obtenidos en las pruebas experimentales pueden tener ligera variación, pues debemos tener en mente que el *throughput* y a latencia de conexión es variable e influenciada por las condiciones de la red en la hora de la ejecución de los experimentos. Probamos esa condición, cuando intentamos predecir algo acerca de la

velocidad de tráfico de datos en Internet. Esto se dio a través del análisis de los datos obtenidos por medio de la ejecución, en diferentes horarios y días en diferente meses, de pings entre dos *clusters* (UAB y UCSAL) geográficamente distribuidos. Los datos recogidos (Figura 3-5), se mostraron muy dispersos, de modo que no fue posible determinar ningún valor medio o determinar alguno patrón de comportamiento [60].

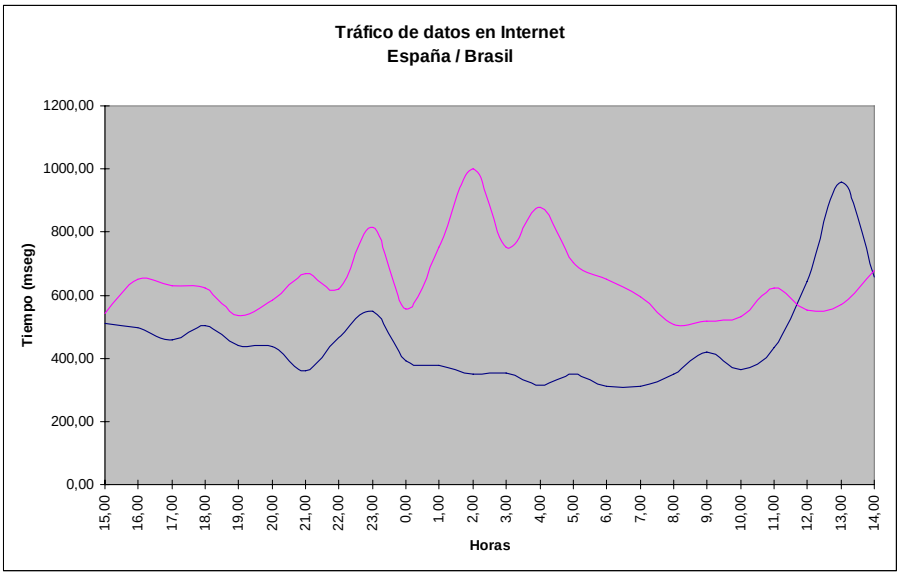

Figura 3-5: Tráfico de datos en Internet – España/Brasil

Para sacar los resultados, ejecutamos la aplicación tres veces, a diferentes horas y en diferentes días, siendo que utilizamos un valor "considerado óptimo"; que fue el que se puso entre el peor y el mejor tiempo de ejecución.

Tal y como hemos visto, la solución fue implementada bajo el paradigma de programación *Master/Worker* [14] [31]. La comunicación entre los nodos de cada uno de los *cluster* se realiza mediante paso de mensajes (*message passing*) utilizando la librería estándar MPI (*Message Passing Interface*) [50], específicamente la librería MPICH v. 1.2.6 [36] [37].

Como aplicación de pruebas utilizamos un programa de multiplicación de matrices.

3.5. Conclusiones

En este capitulo se ha presentado la arquitectura propuesta y el modelo de ejecución basado en *Master/Worker* jerárquico para poder utilizar en CoHNOW.

En esta arquitectura, cualquiera de los nodos de lo HNOW puede asumir las funciones de *Master* Principal, Submaster, Gestor de Comunicación o *Worker*; sin embargo, debemos

tener en mente la heterogeneidad del *cluster* a la hora de escoger que nodo puede asumir que funciones.

Ha sido analizada la complejidad de los problemas que se presentan y analizado el sistema propuesto en nuestro grupo de investigación para soportar los fallos intermitentes de la red para evitar que la ejecución se vea afectada.

En el Capitulo 4 vamos a ver el modelo propuesto para tolerar fallos en los nodos computacionales, basado en Replicación de Datos.

Para implementar la tolerancia a fallos se ha realizado un *Middleware* y se propone una metodología para diseñar aplicaciones que utilicen esta capa para la tolerancia a fallos.

Capítulo 4

Modelo de tolerancia a fallos usando Replicación de Datos para arquitecturas multicluster

Este capítulo presenta el modelo propuesto de tolerancia a fallos para trabajar con la arquitectura multicluster.

Se detalla como tolerar los fallos en los nodos de cómputo del *cluster*, teniendo en cuenta el papel de cada nodo en la arquitectura *Master/Worker* jerárquico.

4.1. Introducción

Basado en el marco teórico descrito en el Capitulo 2 (Marco teórico y trabajos relacionados), se ha diseñado un modelo de tolerancia a fallos para multiclusters de computadores, geográficamente distribuidos, basados en Replicación de Datos, denominado FTDR (*Fault Tolerant Data Replication*).

En el Capitulo 3 (Arquitectura multicluster) se ha presentado como se toleran los fallos transitorios de la red WAN, gestionado por el Gestor de Comunicación, responsable de la comunicación entre *clusters*, utilizando un *Middleware* específico. En este capítulo vamos a ver como tolerar los fallos en los nodos de cómputo del *cluster*.

El objetivo del modelo es prevenir fallos en cualquiera de los nodos de cómputo del sistema, generando para ello la redundancia necesaria, basada en la Replicación de Datos, detectar fallos en cualquiera de los nodos de cómputo del sistema, tolerar este fallo recuperando la consistencia del sistema y garantizar la finalización del trabajo, teniendo en cuenta que existe una degradación de las prestaciones, porque no se utilizan componentes adicionales.

Es necesario garantizar que se ejecuta correctamente el trabajo total, a pesar de que falle algún elemento del sistema, esto supone que introducir un *overhead* ocasionado por el sistema de tolerancia a fallos (protección y detección), por otro lado en caso de fallo, el objetivo es perder el mínimo trabajo posible de los nodos que queden desconectados por fallo, considerando que disminuyen las prestaciones del sistema, debido tanto a las consecuencias que tiene perder un nodo, como al *overhead* añadido para la recuperación y re-configuración del sistema.

FTDR utiliza una redundancia funcional, es decir, no aumenta el número de componentes físicos ya que utiliza la redundancia implícita de un multicluster en el que por definición existe más de un nodo de cómputo. Las ventajas de este modelo es que no añade un coste adicional, en ausencia de fallos todos los nodos están siendo usados para mejorar las prestaciones, y debemos tener en cuenta que cuanto menor es el número de componentes, menor es la probabilidad de fallo, lo que no ocurre en modelos que utilizan redundancia física, ya que tal y como dice Weber [68] “La redundancia física aumenta el número de componentes del sistema y cuanto mayor es el número de componentes, mayor la probabilidad de fallo”.

FTDR provee tolerancia funcional a fallo para todos los nodos del *cluster*. Protegemos la información del *cluster* por medio de Replicación de Datos [69].

FTDR Considera que un *Worker* es una máquina sin estado (*stateless*), cuando existe un fallo, no se recupera el trabajo o la tarea que está realizando en ese momento, dicha tarea se debe reejecutar entera, sin embargo, si que se conserva todo el trabajo finalizado por ese *Worker* a lo largo de la ejecución del programa paralelo. El proceso *Worker* no se relanza en otro nodo, sino que la tarea asignada se reasigna a otro *Worker*. Tal y como hemos visto, los Subcluster son considerados desde el *cluster* principal, como *Workers* con mayores prestaciones, por lo tanto, en el *cluster* principal se conserva el trabajo finalizado, pero no se conserva el estado de los otros *cluster*.

Para la protección del *Master*, FTDR mantiene replicada la información (datos y estado) en otro/s nodo/s, en caso de fallo se reconfigura el *cluster*, y otro nodo pasa a ejecutar las tareas del *Master*. Normalmente sólo se usa una copia de los datos, salvo si existe un fallo que se utiliza la copia. En caso de fallo en un *Worker* o Gestor de Comunicación (CM), se reasigna la tarea perdida a otro *Worker* u otro CM.

Una de las piezas claves en el criterio de decisión por parte de los usuarios, al seleccionar el modelo más apropiado de tolerancia a fallos para sus aplicaciones, son las **prestaciones**, por lo tanto, es una de las características que debemos tener en cuenta, en el mecanismo de tolerancia a fallos. Esto es particularmente necesario en aplicaciones que requieren altas prestaciones. Por ello, para validar el sistema, uno de los criterios a tener en cuenta será el *overhead* introducido por el sistema de tolerancia a fallos FTDR diseñado, tanto en ausencia de fallos como la degradación que sufre el sistema en presencia de fallos, o sea, se evalúa el tiempo total de ejecución con y sin fallo y se compara con el tiempo de ejecución sin el sistema de tolerancia a fallos.

Otra característica importante es la **escalabilidad**, en nuestro caso, estamos trabajando en un entorno multicluster y cada uno de los *cluster* utiliza recursos propios para la tolerancia a fallos, de forma que la tolerancia a fallos en un *cluster* se detecta, previene, recupera y reconfigura en el propio *cluster*, de esta forma los datos siempre se replican en el *cluster* local, sin tener que usar la red WAN para la Replicación de Datos, provocando el mínimo *overhead*, afectando sólo al tiempo de ejecución de la aplicación en el multicluster la perdida de prestaciones en cada uno de los *cluster*. El fallo o la pérdida de un *cluster* entero, es considerado por el multicluster como la pérdida de un *Worker* de altas prestaciones.

En el Capitulo 3 (Arquitectura multicluster) se ha visto como se toleraban los fallos transitorios de la red. En este capítulo vamos a ver como tolerar los fallos en los nodos

computacionales del *cluster*.

El modelo simplifica al programador la gestión de tolerancia a fallo, siendo un sistema semitransparente, que puede configurarse para admitir la posibilidad de tolerar un determinado número de fallos simultáneos.

El sistema lo podemos analizar como un modelo de capas (Figura 4-1). Para la tolerancia a fallos, se ha diseñado y desarrollado un *Middleware*, o capa del *software* que está entre el sistema operativo y la aplicación, para cada nodo del multicluster. Este *Middleware*, también sigue un modelo de ejecución *Master /Worker*, de forma que los componentes del *Middleware* en los *Workers* sólo se comunican con los componentes del *Middleware* en el *Master*.

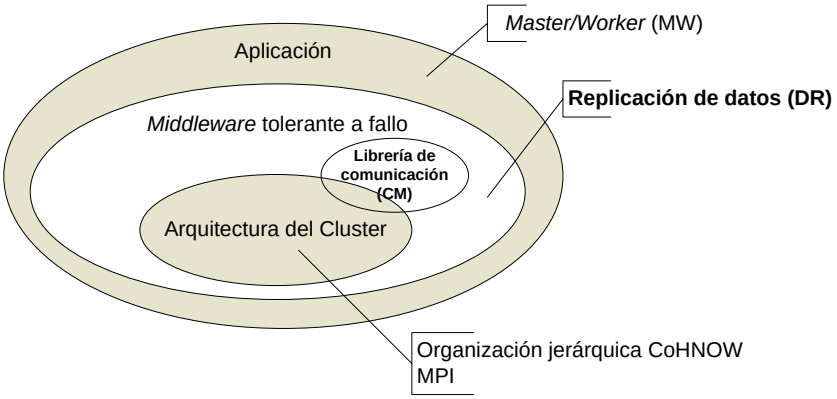

Figura 4-1: Modelo de software en capa

En la capa correspondiente al *Middleware* (Capa 2), están todos los componentes del *Middleware* para dar soporte a la gestión de la tolerancia a fallos (Figura 4-2), vemos que para cada uno de los Subclusters se instalan todos los componentes del *Middleware*, de forma que hay una protección distribuida en cada uno de los *clusters*. La capa 2 del modelo, corresponde a un *Middleware* diseñado para que el sistema sea tolerante a fallo, de un modo transparente al usuario**,** sin intervención del usuario durante la ejecución, a pesar de que exista un fallo, el usuario sólo debe configurar los parámetros iniciales para establecer la tolerancia a fallos, parámetros tales como el nivel de protección de fallos (número de fallos simultáneos en el mismo *cluster* que se quieren soportar) o el tiempo para los procesos de detección de fallos.

FTDR está localizado entre la capa 1 de arquitectura del *cluster* (organización jerárquica CoHNOW) [3] [32] [60] y librería de paso de mensaje MPI [50] y la capa 3 o capa de aplicación.

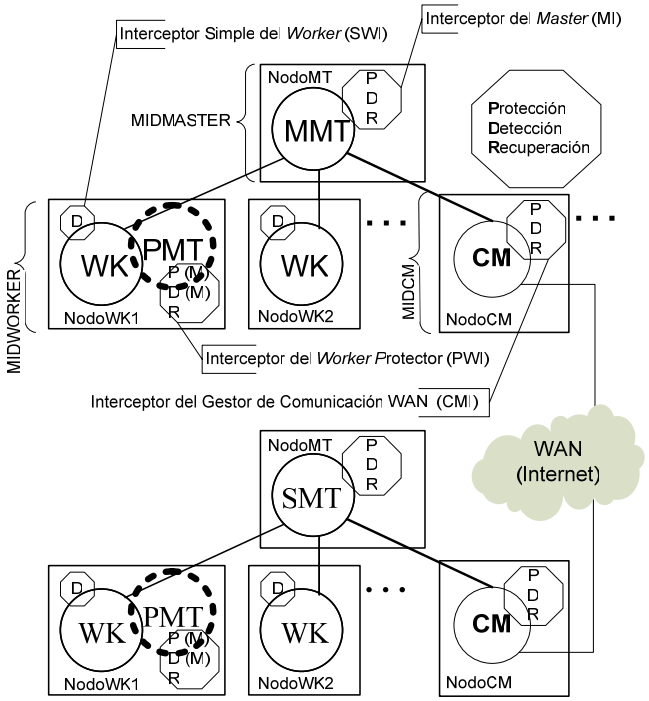

Figura 4-2: Capa 2 - *Middleware* tolerante a fallo

4.2. Tipos de fallos

Los Fallos en la red (LAN y/o WAN), tal y como vimos en el Capitulo 3, son considerados fallos transitorios (no existe un fallo definitivo, a pesar de que puede ser un transitorio muy largo), de forma que los fallos transitorios en la LAN requieren reintentos de envío / recepción de mensajes, en concreto son tratados por reintentos de TCP/IP y MPI.

Si falla el interfaz a la red de un nodo concreto, consideramos que el nodo ha fallado. Nos basamos en que los enlaces y conmutadores que forman la red son cada vez más robustos y, por tanto, menos propensos a fallos permanentes, por lo que dichos fallos los podemos considerar despreciables en este tipo de sistemas.

Si se quisiera prevenir los fallos en la red local (LAN) se deberían tener una red que permita establecer rutas alternativas, que permitan el reenvío de los mensajes afectados. Por ejemplo, se podría disponer de más de un canal físico uniendo los nodos, de modo que el fallo de uno de ellos no eliminase necesariamente la conexión, sino que permitiese el uso del otro canal. A pesar de su simplicidad, el principal inconveniente de este tipo de estrategias es el elevado número de recursos que precisa, lo que contribuye a incrementar el coste, tamaño, y consumo del sistema. No obstante, el empleo de componentes redundantes es la alternativa que podemos considerar para proporcionar tolerancia a fallos de los

interfaces de red.

Tal y como hemos visto, el mecanismo de tolerancia a fallos permanentes en la redes locales de interconexión (LAN), que se podría considerar es el empleo de componentes físicos redundantes, no considerados en nuestra propuesta. Puesto que en el multicluster se utiliza una red local tipo Ethernet, tal y como las que se usan en un sistema de bajo coste, no podemos considerar el empleo de algoritmos de encaminamiento tolerantes a fallos o el empleo de estrategias de re-configuración de la red.

Los fallos en la WAN también se consideran como desconexiones transitorias, no consideramos que Internet pueda fallar permanentemente. Lo fallos detectados de perdidas de mensajes son tratados por la función de tolerancia a fallo del Gestor de Comunicación, que hace control y retransmisión de mensajes entre los *clusters*, estas retransmisiones muchas veces utilizan rutas alternativas. En el caso de que ocurra un fallo en la WAN, los nodos continuarán trabajando hasta que se acabe el trabajo total o la red vuelva la normalidad: el trabajo total podrá concluirse en el *Cluster* Principal.

A diferencia de los fallos en la red, los fallos en nodos del CoHNOW, se consideran fallos permanentes, o sea, si un nodo falla, se aislará del *cluster* y su trabajo será asignado a otro nodo. Para ello es necesario realizar el diagnóstico (para no aislar nodos sanos por una falsa detección de fallo) y pasar a una fase de re-configuración y recuperación. Debemos tener en cuenta, en el momento de la recuperación, recuperar el trabajo realizado para que la aplicación termine con el mínimo *overhead*, pero también recuperar la redundancia necesaria para poder volver a tolerar un nuevo fallo.

Para poder tolerar los fallos y continuar la aplicación con las mínimas pérdidas posibles, al comenzar la aplicación se crean unas tablas de configuración/estado que almacenan el estado del sistema. La tabla de configuración/estado: es una tabla creada/mantenida dinámicamente por el *Middleware* del *Master*, está almacenada en los nodos *Master* (*Master* Principal y Submaster) y replicada en los Protectores del *Master* (PMT). Esta tabla sirve para reconfigurar el sistema en caso de fallo y se actualiza, cada vez que se asigna una tarea, cuando se reciben los resultados y cuando se diagnóstica un fallo.

4.3. Modelo de programación

FTDR utiliza como modelo de programación el *Master/Worker* (Figura 4-3), donde todos los nodos *Workers* ejecutan el mismo programa. En nuestro sistema existe una replicación de los procesos, de forma que todos los Nodos Procesadores (NPRO) tienen

todos los programas, estos programas pueden estar activos o no. En caso de fallo, durante la re-configuración del sistema, un nodo puede cambiar el papel que tenía asignado, esto se logra enviándole los datos necesarios, actualizando la tabla de configuración y activando el proceso correspondiente.

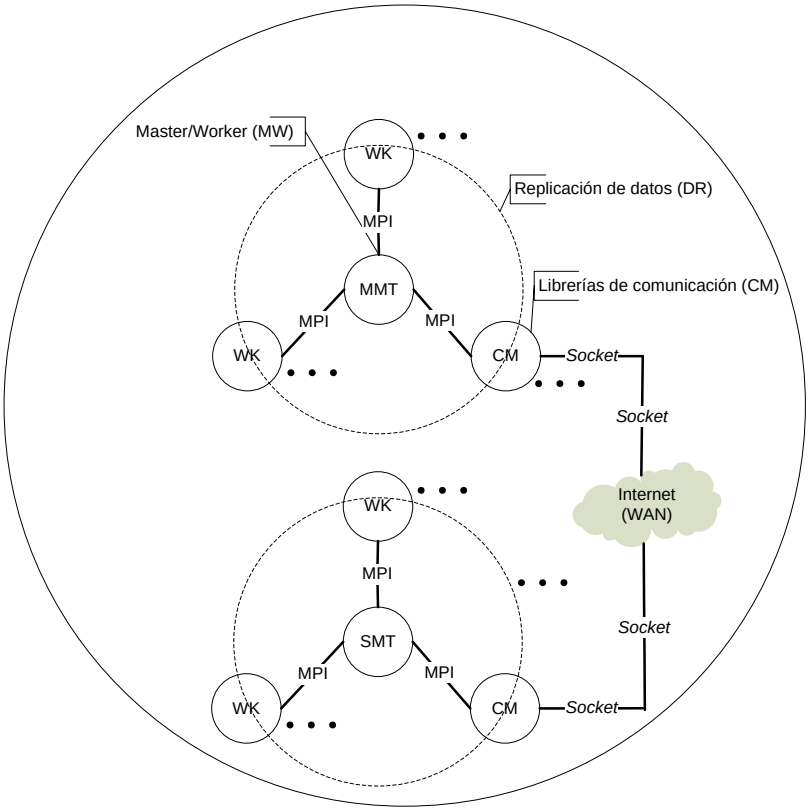

Figura 4-3: Capa 1 – Arquitectura

En la Figura 4-4 vemos los diferentes tipos de actuación de procesos que tendremos en un nodo concreto:

- Nodo *Master* activa el proceso de *Master*.
- Nodo *Worker* activa el proceso de *Worker*.
- Nodo *Worker* / Protector del *Master*: activa ambos procesos, el proceso *Worker* y el de Protector del *Master* (PMT).
- Nodo Gestor de Comunicación activa el proceso de Gestor de Comunicación.

Para gestionar todos los procesos, en cada nodo se instala un Controlador del Nodo (NC), que es el *software* encargado de la configuración/reconfiguración del nodo, es decir el que activa el proceso o procesos correspondientes al rol del nodo.

En los Subcluster ocurre lo mismo, pero se activa el proceso Submaster (SMT) en el nodo *Master*, que debe comenzar esperando los datos del *cluster* principal y a medida que

va recibiendo los resultados de los *Workers* debe reenviarlos al *cluster* principal.

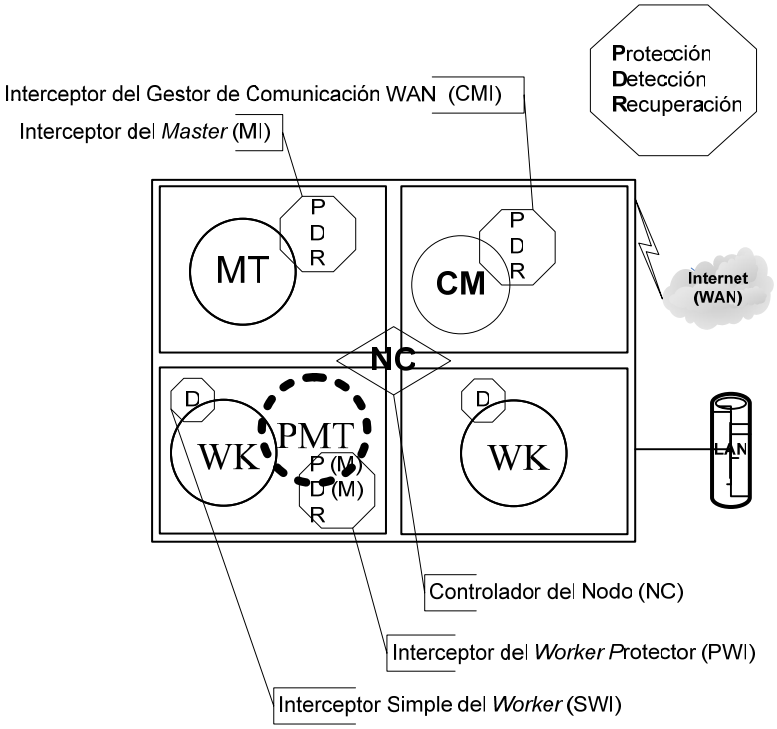

Figura 4-4: Nodo procesador (NPRO).

En el paradigma *Master/Worker*, solamente tenemos comunicación entre *Master* (MT) y *Workers* (WK), o sea, los *Workers* nunca se comunican directamente con otros *Worker*, en el caso de *Middleware* también se sigue el mismo paradigma de ejecución.

El *Master* es responsable de dividir el trabajo en pequeñas tareas y distribuir las tareas a los *Worker* en un esquema dinámico o bajo demanda (*on-demand)*, es decir, cuando acaban una tarea se le asigna la siguiente.

En el caso del Gestor de Comunicación se le asigna estáticamente toda la tarea que debe ejecutar el Subcluster, el Submaster hará la distribución entre los *Workers* del Subcluster. Tal y como hemos visto, los datos iniciales se enviarán al Subcluster, es decir, se realiza una Replicación de Datos en el Submaster para mejorar las prestaciones. En el Subcluster se activará también el sistema de tolerancia a fallos, por tanto, estos datos también serán replicados en el Subcluster. Los resultados se irán devolviendo a medida que van siendo generados y una vez en el *cluster* principal, serán tratados como el resto de resultados de los *Workers*, o sea, estarán replicados en el *cluster* principal, para evitar el *overhead* que supondría recuperar los datos a través de la WAN en caso de fallo del *Master*, en la fase de recuperación.

4.4. Operación

Al tratar el problema de la tolerancia a fallos es importante especificar frente a que fallos protegeremos, es decir, seleccionamos un conjunto de fallos que "pueden pasar" con cierta probabilidad, aunque esta sea pequeña, pero que pueden repercutir negativamente en la ejecución.

Tal y como hemos visto, es necesario especificar los fallos que se van a considerar, nuestro sistema está diseñado para tolerar fallos permanentes en los nodos de computo, se propone un mecanismo de tolerancia a fallos basado en la Replicación de Datos que permita acabar la ejecución de la aplicación aunque el multicluster esté funcionando con menores prestaciones. Para hacerlo transparente al usuario se dispone de un *Middleware* que se ejecutará junto con la aplicación. El *Middleware* diseñado, tal y como hemos visto, conserva la estructura *Master/Worker*.

Hemos visto que la prevención o protección frente a fallos se realizaba mediante la Replicación de Datos y que esta operación se consideraba una transacción con las características **ACID** (ver apartado 2.5).

Para la detección y diagnóstico se usa un mecanismo de *heartbeat*: es un mecanismo de diagnóstico de fallo que consiste en que los nodos del *cluster* envían un flujo periódico de mensajes de control otro. Si el flujo se interrumpe en algún momento, puede concluirse un fallo en algún nodo o en la red.

El *Middleware* en el *Master* es el encargado de la detección (recibiendo el *heartbeat* del resto de nodos del *cluster*) y diagnóstico de fallos en los W*orkers* y Gestores de Comunicación, de la protección del estado global del sistema y de la recuperación re-configuración en el caso de fallo de uno de estos nodos. En los *Workers* básicamente están encargados de actualizar su *heartbeat* para la detección de fallos

Para la protección del *Master*, se ha creado un proceso para la protección del *Master* (PMT), encargado de detectar y de diagnosticar los fallos en el *Master*, también almacena la replica del estado global del *cluster*, que le permite realizar la recuperación/re-configuración en el caso de fallo del *Master*.

Hay que tener en cuenta, como influye la Replicación de Datos en la relación cómputo/comunicación. Para que la Replicación de Datos no sea muy costosa la relación debe ser adecuada. Poca comunicación (de la que gasta tiempo del *Master*) y durante el cómputo de la aplicación, mientras se espera el cómputo, se realiza la Replicación de Datos.

La granularidad para relacionar los tiempos de cómputos y de comunicación puede permitir calcular si el *Master* estaba mucho tiempo libre, una vez envía datos, puede dedicar los recursos a la tolerancia a fallos. Como hemos visto en el Capitulo 3, donde se presentaba la arquitectura multicluster, para poder realizar el solapamiento, existen *buffers*, que permite este funcionamiento en pipeline.

Como hemos visto en el Capitulo 3, donde se presentaba la arquitectura, en los *Workers* (ver apartado 3.4) existe un doble buffer para la recepción de datos, para que se solape el cómputo de una tarea con la recepción de los datos de la siguiente (solapamiento cómputo/comunicación). Para controlar el *overhead* introducido por la tolerancia a fallos, también intentaremos solapar al máximo las tareas de tolerancia a fallos con las de cómputo, para ello el *Master* primero enviará trabajo a los *Worker*, y mientras estos computan realizará las tareas de tolerancia a fallos, tales como la Replicación de Datos y actualización de tablas. El principal problema que presenta esta alternativa, es que en caso de fallo, se puede perder más trabajo realizado, pero en el caso de funcionamiento sin fallo logramos minimizar el *overhead* introducido por la tolerancia a fallos.

4.4.1. Configuración

Inicialmente el usuario debe especificar un número *C* de fallos simultáneos frente a los que se quiere proteger, teniendo en cuenta, tal y como hemos visto, que el número de fallos que se presenta en un momento dado dependerá del tiempo medio entre fallos (MTBF). La probabilidad de que dos o más fallos estén presentes simultáneamente decrece. Normalmente, si las tareas no son críticas inicializaremos *C* a 1, de forma que sólo se realizará una replica de lo datos, el principal problema se presenta si ocurre un fallo mientras estamos en el proceso de recuperación, en una de las máquinas implicadas en el proceso de recuperación, ya que durante esa operación pueden existir datos no replicados.

Otro parámetro que puede configurar el usuario es el tiempo de *heartbeat*, este tiempo influye en el *overhead* del sistema, pero debemos tener en cuenta que tiempo muy grandes favorece que se incremente el tiempo de latencia hasta diagnosticar el error, y tiempos muy pequeños puede sobrecargar el envió/recepción de mensajes.

Respecto a la **configuración inicial** del sistema, debemos tener en cuenta, tal y como hemos visto, que todos los nodos tienen todos los programas, que pueden estar activos o no. El Controlador del Nodo (NC) (Figura 4-4) es el responsable de activar los procesos en cada nodo, de acuerdo con los esquemas de protección y de re-configuración especificados.

Inicialmente se especificará que nodos actuarán como *Worker* y como Gestor de Comunicación, en función del número de Subcluster que se conecten en el multicluster.

Una vez especificado el número de fallos simultáneos (*C*) que puede soportar el sistema, se realizará la configuración del multicluster, se configurará el número de procesos Protectores del *Master* (PMT) que estarán activos, por defecto consideraremos que como mínimo existe un PMT (*C*=1), se activarán las tablas de configuración y estado, se realizará la Replicación de Datos y la aplicación se ejecutará junto con el *Middleware* que irá activando dinámicamente las fases del esquema de FTDR:

- Protección utilizando mecanismos de Replicación de Datos
- Detección y diagnóstico, utilizando mecanismos de *heartbeat*.
- En caso de fallos se activará las fases de recuperación de datos (de la aplicación y de los mecanismos de tolerancia a fallos) y re-configuración del *cluster* donde se ha producido el fallo.

4.4.2. Protección

La organización jerárquica tiene ventajas desde el punto de vista de la protección, ya que permite que los *clusters* se protejan localmente, es decir, tal y como hemos visto, los Gestores de Comunicación comunican los *clusters* y junto con el Submaster que recibe una réplica de los datos iniciales que debe computar su cluster, permiten aislar los problemas en cada uno de los *clusters*.

La redundancia física o *hardware* de nodos es intrínseca en el *cluster*, puesto que existen varios nodos de procesamiento (NPRO). FTDR añade una redundancia funcional, protege todos los NPRO del *cluster* usando las máquinas existentes, o sea, no hay redundancia extra de *hardware*, sólo redundancia de información: replicación de procesos en todos los nodos y Replicación dinámica de Datos.

La protección de los *Workers* se realiza desde el *Middleware* en el *Master* a través de la creación de una tabla en la que se registra dinámicamente la configuración del *cluster* (la asignación de tareas) y el estado, durante la ejecución el *Middleware* es el encargado de la modificación de tablas para mantener el estado global del sistema y de la reasignación de trabajos a otro *Worker* en caso de fallo. Esta tabla se crea en el *Master* del *cluster* principal (MMT) y unas tablas similares se crean en cada uno de los Submaster (SMT) y son replicadas en los Protectores del *Master* (PMT).

El Fallo de un Nodo *Worker* (NodoWK) provoca un manejo parecido al manejo de

renombrado de registros (manejo del pool de recursos), es decir, durante la ejecución sin fallos, se guarda en la tabla de configuración de los mensajes: que tarea es y a que recurso (nodo) es asignada (Tabla 4-1), además del estado de dicha tarea, es decir, si se ha enviado y si se ha acabado, en cuyo caso el *Master* ya tiene los resultados.

Tabla 4-1: Tarea/recurso/estado

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**

Partimos, de que la tarea asignada a un *Worker* es una tarea atómica, que no puede ser reanudada en medio de la tarea, es decir, si existe un fallo debe comenzar de nuevo. La asignación de una tarea a un *Worker* es similar a una transacción, o agrupación de operaciones básicas: envío, proceso y recepción de resultados. Esta operación básica de una tarea comparte las propiedades de una transacción:

- Es una operación atómica: consideramos que o se realizan todas las operaciones de la transacción o no se realiza ninguna de ellas.
- Concurrente: pueden ejecutarse varias tareas a la vez, en cada uno de los *workers*.
- Independiente: las tareas concurrentes no interfieren entre sı ni pueden “ver” estados intermedios de otras tareas, puesto que los *workers* no pueden comunicarse entre si.
- Durable: una vez que la tarea se ha realizado sus resultados son permanentes.

Teniendo esto en cuenta, en la tabla de estado de las tareas/recurso/estado (Tabla 4-1), para cada una de las tareas consideramos que la tarea puede estar en diferentes estados, sin comenzar o enviada pero no acabada, por lo tanto si existe un fallo se debe ejecutar de nuevo entera; o enviada y acabada, en este caso estos resultados son permanentes.

Como las tareas son independientes podemos tener varias tareas enviadas y no acabadas pero no se interfieren entre si, si un nodo falla sólo hay que rehacer la tarea asignada a ese nodo.

La durabilidad de los datos del *Worker* se debe garantizar una vez terminada la tarea y se debe garantizar que los resultados calculados son permanentes, incluso cuando existan

fallos. Para conseguir atomicidad y durabilidad se usa protocolos de protección que usan Replicación de Datos en memoria estable al finalizar la tarea, de modo que si un nodo falla todos sus resultados están protegidos. Si un *Worker* falla mientras está procesando o enviando resultados al *Master*, se ignoran estos resultados, teniendo que reejecutarse la tarea entera.

- **Alternativas para la protección de datos**

Al comenzar la ejecución, el *Master* replica los datos iniciales en los nodos Protectores del *Master* (PMT). La protección de los resultados de la aplicación se realiza través de la Replicación de Datos que puede realizarse utilizando dos esquemas diferentes. Una alternativa es hacer una **Replicación de Datos Centralizada** en los PMT, la otra alternativa es que la **Replicación de Datos quede distribuida** entre los *Worker* y los Gestores de Comunicación.

Los Protectores del *Master* (PMT) en todos los *clusters*, son nodos que trabajan como *Workers* pero que están preparados para asumir las funciones de los *Masters* (*Master* o Submaster) en caso de detectar un fallo en uno de ellos. El PMT es un proceso (elemento lógico) encargado de la protección del *Master*. No existen nodos dedicados a esta tarea, sino que en nodos que alojan un *Worker*, en el que además de la tarea del *Worker*, hay otro proceso, el Protector del *Master* PMT, que hace la protección, la detección y diagnóstico del *Master*. Además en caso de fallo del *Master* este nodo encargado de la protección del *Master* sería el encargado de la recuperación del *Master*, la re-configuración del *cluster* y se convertiría en el nuevo nodo dedicado exclusivamente a las tareas de *Master*, por lo tanto deberá matar el proceso *Worker* en el nodo. Si por necesidades de prestaciones fuera necesario, el modelo también soportaría que un PMT también estuviera en un nodo dedicado que estuviera dedicado al Protector del *Master*.

En la Replicación de Datos centralizada, cada vez que llega un resultado al *Master* se envía una nueva tarea para el *Worker*, y luego se replica su resultado y se cambia el estado. Tal y como hemos visto, el enviar los datos antes de comenzar las tareas de protección permite solapar el cómputo en los *Workers* con las tareas de protección que se desarrollan en el *Master*. Una vez realizado, el *Middleware* en el *Master* (MIDMASTER) actualiza la tabla indicando el nuevo envío y replica los resultados en los Protectores del *Master* y la tabla de estado indicando la recepción de datos y replica la tabla de configuración/estado actualizadas.

Para garantizar la sincronización/congruencia de datos, solo se actualiza la tabla que

indica los datos ya procesados después de que exista la garantía de que los datos fueron almacenados en el *Master* y que fueron replicados con éxito en los Protectores del *Master* (PMT).

Esta forma de actuación de Replicación de Datos centralizada, presenta la característica de que en el caso de fallo del *Master*, la recuperación en el Protector del *Master* es muy rápida, de forma que el *overhead* de la recuperación será mínimo.

En la **Replicación de Datos distribuida**, se utiliza la memoria estable de los *Workers* y del Gestor de Comunicación, para hacer la Replicación de Datos, de forma que cuando un *Worker* o un Gestor de Comunicación tiene un resultado, lo almacena y lo envía al *Master*, de esta forma siempre hay una copia de los datos, pero en caso de fallo del *Master* o de un *Worker* hay que recuperar los datos que estaba almacenados en el nodo que ha fallado, a partir de la copia y hacer una nueva replica.

El funcionamiento de la Replicación de Datos distribuida requiere los siguientes pasos: Cada vez que se termina la ejecución de una tarea, los *Worker* o Gestor de Comunicación guardan sus propios resultados, además de enviarlos al *Master*.

En caso de fallo del *Master*, el nuevo *Master* pide los resultados de los trabajos ya realizados por los *Workers* para poder rehacer los resultados a partir de los datos replicados en los *Workers*.

En el caso del fallo de un nodo *Worker*, tenemos una copia de los resultados obtenidos por ese *Worker* en el *Master*, pero debemos tener en cuenta, que además del nodo y el trabajo que estuviera realizando en ese momento, se ha perdido la replicación de los datos que éste almacenaba, por lo tanto es necesario replicar los resultados de este *Worker* en otro *Worker*, en este caso, para que el sistema sea determinista, se elige el nodo en el que está el Protector del *Master* (PMT), de este modo tendremos los resultados en el *Master* y una replica en el nodo en el que está el PMT.

En el caso de los datos procesados por el Subcluster, la Replicación de Datos queda almacenada en el Gestor de Comunicación local, de forma que no se deben volver a solicitar su retransmisión a través de la WAN. En el caso de fallo del Gestor de Comunicación sus datos se replicarán también en el Protector del *Master*.

Tal y como hemos visto, después de enviar una nueva tarea para el *Worker*, el Middleware del *Master* actualiza la tabla de estado y la replica en los Protectores del *Master* para que estos siempre mantengan una copia del estado del proceso actualizada. Vemos que la Replicación de Datos distribuida, requiere tener en cuenta la información del “nodo” en

la tabla de estados y es que además de identificar el estado de la tarea, debemos especificar que *Worker* calculó los resultados y por lo tanto tiene la Replicación de Datos ya que en el caso de fallo del nodo, en el momento de la re-configuración, debemos realizar la Replicación de Datos en otro *Worker*, y se debe actualizar dicha tabla indicando de nuevo el nodo que tiene la Replicación de Datos.

Esta alternativa presenta menor *overhead* en la ejecución sin fallos, pero hace un poco más complejas, en caso de fallo, las fases de recuperación y re-configuración del *cluster*.

- **Protección del estado global de ejecución o del sistema**

La tabla de configuración es replicada inicialmente por el *Master* en los Protectores del *Master* (PMT) y cada vez que es actualizada es replicada. Después de enviar una tarea para un *Worker*, el MIDMASTER actualiza la tabla de estado y la replica en los PMT.

El MIDMASTER se crea en el momento de inicializar (*startup*) y replica los datos iniciales y la tabla de configuración (Tabla 4-1) en el Protector del *Master*. Siempre que un *Worker* o Gestor de Comunicación envía un resultado, el MIDMASTER devuelve una confirmación (*ack*) de recepción de resultado.

Para garantizar la protección del estado global durante la ejecución, hay que monitorizar la información necesaria desde el *Middleware* para poder actualizar la tabla de estado, también es necesaria la Replicación de los datos de las tablas de configuración/ estado, donde se especifica el trabajo asignado a cada nodo.

Para garantizar la sincronización /congruencia de datos, solo se actualiza la tabla de configuración, después que se pueda garantizar que el resultado fue almacenado en el *Master* y en el caso de una protección centralizada, que fueron replicados con éxito en los Protectores del *Master*.

Cuando el MIDMASTER replica los datos, los Protectores del *Master* devuelven una confirmación (*ack*) de recepción de datos, y en este momento se modifica el estado de dicha tarea. Esto es necesario para garantizar la sincronización/congruencia de datos, sólo se actualiza la tabla de estado después de que se garantice que los datos están almacenado en el *Master* y en el *Worker* o Gestor de Comunicación en caso de una protección de datos

$$t_{overhead}^{sf} = t_{MIDMASTER} + t_{RD}$$

$$t_{RD}^{sf} = t_{tconfig/estado} + t_{\mathrm{Re}sultWK}$$

Figura 4-5: *Overhead* - protección del estado global de ejecución o del sistema

distribuida, o que fueron replicados con éxito en los Protectores del *Master* en el caso de protección de datos centralizado.

La Figura 4-5, muestra el *overhead*, para protección del estado global de ejecución o del sistema.

4.4.3. Detección de fallos

La detección de fallos se basa en el uso de los *heartbeat.* El *Middleware* puede diagnosticar fallos en un nodo contando los sucesivos *heartbeat* perdidos, de este modo pretendemos no aislar nodos sanos, sólo porque existe un fallo de comunicación transitorio, y no un fallo en el nodo. Siempre que el número de *heartbeat* alcanza un límite previamente especificado, se considera que el nodo falló, este número, así como el tiempo de *heartbeat*, son parámetros que pueden ser configurados.

Debido a la latencia de red y para evitar "falsa alarma", es necesario que el usuario defina la frecuencia del *hearbeat*. Para definir esta frecuencia, se debe tener en cuenta, que tanto que dicha frecuencia incide en el *overhead* introducido por la detección, cuando el sistema está funcionando normalmente sin fallos, y que esto puede afectar al tiempo de ejecución sin fallo, como que también influye en la latencia del diagnóstico. Es decir, tenemos un compromiso entre la frecuencia y su influencia en el *overhead*, de ejecución sin fallos debido a los recursos que necesita, más uso de la red, y si enviamos *hearbeat* poco frecuentemente, puede influir negativamente en el tiempo que tardamos en diagnosticar el fallo o latencia del fallo.

El Middleware en los nodos *Worker* y Gestor de Comunicación envían los *heartbeats* al *Master* y el *Master* es el que debe realizar el diagnóstico, es importante definir, por lo tanto, de cuanto en cuanto tiempo el *Master* debe esperar recibir los *heartbeats* enviados por los *Workers* (frecuencia de *heartbeat*) y el número de *hearbeat* perdidos que se va a tolerar (latencia de diagnóstico). Como hemos visto, el usuario también deberá tener cuidado al definir el tiempo entre *heartbeats*, para evitar saturar al receptor, de forma que los mismos sean enviados antes de que el nodo que posee el interceptor tenga capacidad de contestar.

$$t^{sf}_{overhead_{WK}} = t_{heartbeat}$$

$$t^{sf}_{overhead_{MT}} = t_{ACK(WK)} + t_{heartbeat(PMT)}$$

Figura 4-6: *Overhead* – detección del fallo

Para poder detectara los fallos en el *Master,* éste envía los *hearbeat* al Protector del *Master* (PMT).

La Figura 4-6, muestra el *overhead*, para detección del fallo.

Vamos a analizar las diferencias en las detecciones de fallo en cada uno de los distintos tipos de nodo:

- **Detección de fallo en un Nodo *Worker***

Para detectar fallo en un Nodo *Worker* (Figura 4-2), cada cierto tiempo predefinido, el

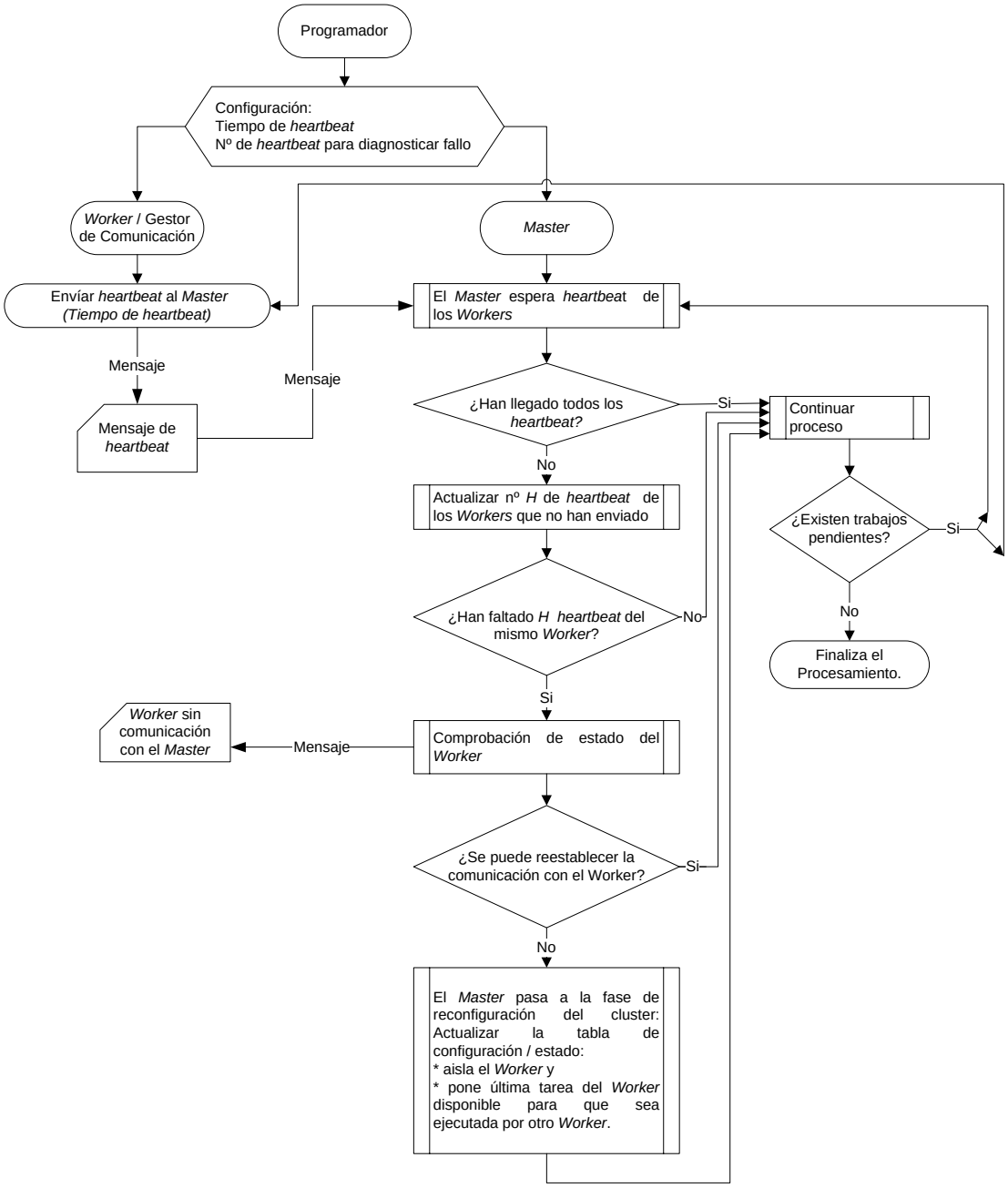

Figura 4-7: Detección de fallo en Nodo *Worker*

Middleware del *Worker* envía un mensaje (*heartbeat*) al *Middleware* en el *Master* indicando que está "vivo".

Para realizar el diagnóstico, si pasado determinado tiempo el *Master* no recibe un *heartbeat* de un determinado *Worker*, el *Master* pasa a la fase de recuperación/ re-configuración del *cluster*, aislando a dicho Worker y poniendo la última tarea como disponible para que sea ejecutada por otro *Worker* (Figura 4-7).

La latencia de la detección depende del tiempo que tengamos previsto desde que se produce el fallo hasta que se diagnostica y se pasa a la fase de recuperación/re-

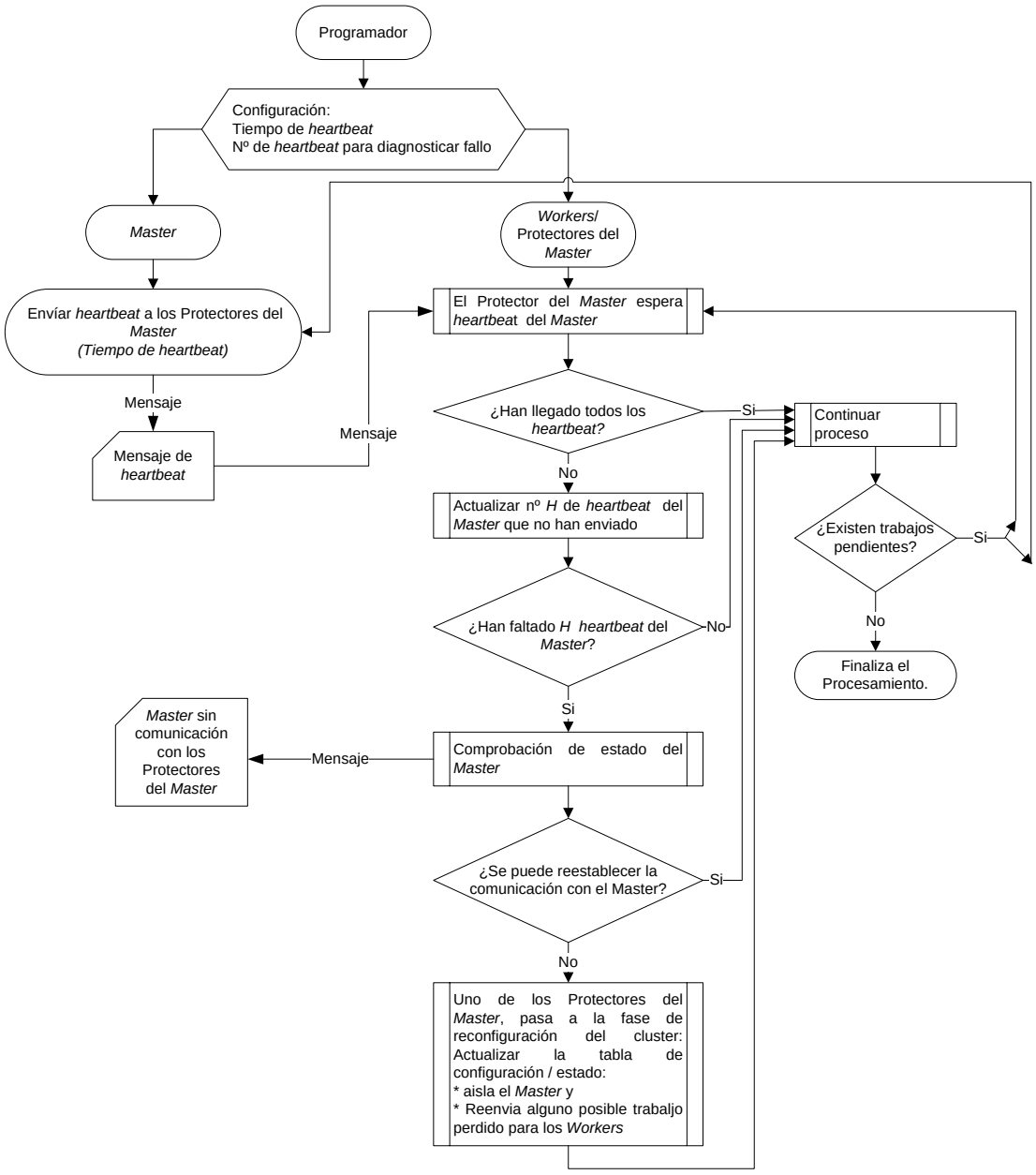

Figura 4-8: Detección de fallo en Nodo *Master*

configuración. Este tiempo está directamente relacionado con el número de *heartbeat* que consideramos que es adecuado para diagnosticar que existe un fallo, debido a que puede ocurrir que no se reciban *heartbeat* por otras causas, como la pérdida de mensajes en la red local (problema poco frecuente). Realmente este tiempo de latencia no es crítico, ya que una vez el nodo falla, sólo se le aísla y se reconfigura el *cluster* de forma que no se le vuelve a enviar trabajo.

- **Detección de fallo en Nodo *Master***

Para detectar fallo en un Nodo *Master*, (Figura 4-2), cada cierto tiempo predefinido, el *Master* envía un mensaje (*heartbeat*) a los Protectores del *Master* indicando que está "vivo".

Para realizar el diagnóstico, de nuevo consideramos que si pasado determinado tiempo el Protector del *Master* no recibe el *heartbeat* del *Master*, se pasa a la fase de recuperación / re-configuración del *cluster* (Figura 4-8).

Tal y como hemos visto en el apartado anterior, se debe garantizar que uno y sólo un Protector del *Master* (PMT) se encarga de la fase de recuperación/re-configuración, para ello es necesario configurar adecuadamente los tiempos de detección y diagnóstico de cada uno de los PMT.

- **Detección de fallo en un Nodo Gestor de Comunicación**

En el *cluster* principal la detección de fallos del Gestor de Comunicación (CM) es similar a la detección de fallos en un *Worker* (Figura 4-7), ya que para el *Master* principal el CM es un *Worker* especial que nos hace transparente a los Subcluster, que son considerados desde el nodo principal como *Workers* con un comportamiento diferente, disponen de una gran potencia de computo, pero también presentan una latencia de comunicación superior, por eso se les asigna estáticamente una gran carga de trabajo que el Gestor de Comunicación se encarga de enviar al Subcluster y recibir los resultados para ir enviándoselos al *Master*.

Para detectar fallo en un Nodo Gestor de Comunicación (Figura 4-2), cada cierto tiempo predefinido, igual que en los *Workers*, el Gestor de Comunicación (CM) envía un mensaje de *heartbeat* al *Master* indicando que está "vivo". Si pasado determinado tiempo el *Master* no recibe el *heartbeat* de un determinado CM, pasa a la fase de recuperación/re-configuración. Este *heartbeat* es interno del *cluster*.

En el caso de los Subcluster el Gestor de Comunicación (CM) es visto desde el Submaster como el sistema de E/S de datos, es el CM quien tiene los datos de entrada (no

están en el disco como en el caso de *Master*) y es donde se dejan los resultados de salida, para que sean enviados al *cluster* principal. También se utiliza el sistema de enviar un *heartbeat* al Submaster (SMT), si el SMT pierde la comunicación local con su CM, seleccionará otro Worker para la comunicación con el *cluster* principal.

4.4.4. Diagnóstico del fallo

Una vez detectado el fallo se debe diagnosticar el tipo de fallo, para no aislar nodos sanos, también es importante identificar, que funcionalidad tenía el nodo que ha fallado, para poder pasar a las fases de recuperación y re-configuración.

Si el fallo fue detectado por el MIDMASTER, entonces existen dos posibilidades:

- Hubo un fallo en un Nodo Worker. En este caso debemos distinguir si además del proceso *Worker* estaba activo en esa máquina el Protector del *Master* (PMT), es decir, el diagnóstico debe distinguir, para la recuperación / re-configuración, entre nodos con sólo tareas de *Worker* o nodos con *Worker* y con PMT.
- Hubo un fallo en un Nodo Gestor de Comunicación.

Si el fallo fue diagnosticado por el MIDWORKER de un Protector del *Master* (PMT), entonces el fallo fue en el Nodo Master. En el caso de tener varios Protectores del *Master* activos, debemos impedir que otro PMT diagnostique también el fallo y pase a la fase de recuperación y re-configuración, para ello debemos asignar distintos valores para la latencia del diagnóstico del fallo.

En el caso de que exista más de un PMT ($C>1$), sólo un PMT debe realizar la fase de diagnóstico, y recuperación/re-configuración, para ello cada uno de los PMT debe tener un tiempo de latencia de diagnóstico diferente. El Protector del *Master* con el tiempo menor será el que realizará la detección y diagnóstico, y se convertirá en el nuevo *Master*, para que el resto de PMT no empiecen este proceso deba enviarles un *hearbeat* indicando que el nuevo *Master* está vivo.

La Figura 4-9, muestra el *overhead*, para diagnóstico del fallo.

$$t^{sf}_{overhead} = t_{diagnóstico}$$

$$t^{cf}_{overhead} = t_{diagnóstico} + t_{latenciaerrror}$$

Figura 4-9: *Overhead* – diagnóstico del fallo

Vamos a analizar las diferencias en las detecciones de fallo en cada uno de los

distintos tipos de nodo.

4.4.5. Re-configuración

Una vez diagnosticado el fallo se debe reconfigurar el *cluster*. Esta re-configuración requiere la actualización de las tablas de configuración/estado e indicar a todos los nodos en el *cluster* la nueva configuración. Las acciones a seguir en la re-configuración dependen del tipo de nodo que falla.

- **Re-configuración después de un fallo en un Nodo *Worker***

Si hay un fallo en un Nodo *Worker*, el *Middleware* del *Master* (MIDMASTER) debe aislar el Nodo *Worker* que falló, para ello el MIDMASTER actualiza la tabla de configuración, de este modo ya no se vuelve a comunicar con este Worker. Mientras se están ejecutando las tareas de re-configuración el *Master* no atiende a los otros *Workers*. La re-configuración es una operación indivisible.

Cuando falla un Nodo *Worker*, el *Middleware* actualiza en la tabla de configuración, es decir, cambia la última tarea asignada a dicho *Worker* y la pone como libre, para que el *Master* lo vuelva a enviar a un nuevo *Worker*. Para ello las tareas o datos a enviar deben tener una doble marca sobre su estado: enviado, acabado. En la Tabla 4-2, vemos como se cambia el estado de la tarea 1, indicando que se ha había enviado al nodo 1 que ha fallado.

Tal y como hemos visto, en el momento de la re-configuración debemos volver a un estado consistente, es decir, debemos volver hacia atrás, para ello, las tareas asignadas al *Worker* que ha fallado cambian su estado; puesto que el *Worker* ya no tiene nada asignado, el *Master* no se volverá a comunicar con él. En el caso del ejemplo de la Tabla 4-2, el *Middleware* cambia el estado de la tarea T1, cuando ocurre un fallo en el *Worker* n1, la

Tabla 4-2: Doble marca

	Configuración		Estado		
Id	**Tarea**	**Nodo**	**Enviado**	**Acabado**	**Significado**
1	T1	n1	Si	No	Enviado
2	T2	n2	Si	Si	Acabado
3	T3	n3	Si	No	Enviado

En si falla un *Worker* se altera el estado de enviado:

	Configuración		Estado		
Id	**Tarea**	**Nodo**	**Enviado**	**Acabado**	**Significado**
1	T1	n1	No	No	Fallo
2	T2	n2	Si	Si	Acabado
3	T3	n3	Si	No	Enviado

tarea figura como una tarea no enviada a ningún nodo, es decir, cambia el estado de enviado y lo pasa a "no" enviado, de este modo queda en el estado de tarea no asignada (enviada = no; acabado = no) porque ha ocurrido un fallo en el nodo al que había sido enviada. Antes de acabar la ejecución esta tarea será reenviada a otro nodo.

Durante esta re-configuración, sólo es necesario parar al *Master*, no es necesario parar a todos los nodos del *cluster*, los nodos *Workers* no implicados en el fallo seguirán ejecutando su computación mientras se modifica la tabla de configuración/estado y se replica en el Protector del *Master*. Por lo tanto, el *overhead* de la re-configuración afecta sólo al *Master* y a los *Workers* que intentan comunicarse con el *Master*.

Si se estaba realizando una protección centralizada, una vez reconfigurado el *cluster*, sólo es necesario replicar la tabla de configuración/estado. Si se estaba realizando una distribuida, en la fase de recuperación, tendremos en cuenta, replicar los resultados que había computado el nodo que ha fallado en otro nodo y actualizar la tabla de configuración.

La Figura 4-10, muestra el *overhead*, para re-configuración.

$$t_{overhead}^{cf} = t_{MIDMASTER}$$

Figura 4-10: *Overhead* – re-configuración

- **Re-configuración después de un fallo en un Nodo Worker con Protector del Master**

En este caso, además de ser necesario realizar todas las operaciones de re-configuración vistas en el aparatado anterior, donde analizábamos la re-configuración necesaria cuando se pierde un *Worker*. Si falla un nodo con el Protector del *Master* (PMT), se debe reconfigurar el sistema para que se active otro PMT en otro nodo, esto implica cambiar la configuración inicial de papel de los nodos, puesto que tenemos replicación de procesos, y se pasa a la fase de recuperación de los datos y tablas.

El objetivo, de activar otro PMT es terminar con una configuración de protección frente a fallos igual a la inicial.

- **Re-configuración después del fallo en un nodo *Master***

Si se muere el nodo con el *Master*, se debe parar el *cluster* y reconfigurar el sistema, para no perder mensajes de los *Workers*, hasta indicar a todos los nodos quien es el nuevo *Master* con el que se tienen que comunicar.

El nuevo *Master* se recupera en el Protector del *Master* (PMT), además es necesario

reconfigurar la máquina activando otro PMT y desactivando el *Worker* que estaba activo en el nodo donde se ha activado el nuevo *Master*. La nueva máquina queda reconfigurada con un nuevo *Master* y un nuevo PMT, y disponemos de un *Worker* menos, puesto que ha fallado un nodo.

La recuperación, como analizaremos a continuación, dependerá del mecanismo de protección del *Master* utilizado, es decir, si existe una replicación centralizada de datos (replicación de resultados en el Protector del *Master*) o una replicación distribuida de datos en la que se marca que los datos están almacenados en el propio *Worker*.

4.4.6. Recuperación

Una vez diagnosticado el fallo y reconfigurado el *cluster*, se debe recuperar el trabajo a partir los datos replicados, volver a un estado consistente y proseguir la ejecución.

Debemos distinguir las actividades a realizar dependiendo la funcionalidad asignada al nodo que ha fallado y del tipo de protección centralizada o distribuida que se ha configurado.

- **Recuperación de fallo en Nodo *Worker***

Para recuperar el trabajo del *Worker* el MIDMASTER analiza el estado global del *Worker*. O sea, analiza en que punto estaba el *Worker* cuando falló, para ver si estaba enviándole datos, procesando o recibiendo los resultados. En el caso de un fallo, si estaba procesando o recibiendo resultados, se debe tener en cuenta que esta tarea no está terminada y se debe repetir entera como una transacción u operación atómica, por lo tanto, si había comenzado a recibir resultados debe desechar los datos recibidos y volver el estado de la tarea a un estado congruente, es decir, como vimos en la reconfiguración se vuelve al estado no enviado, no acabado, después del tratamiento del fallo. Una vez el MIDMASTER actualiza la tabla de estado del sistema debe replicar la tabla en el Protector del *Master*.

Vemos que el estado global del sistema vuelve a un estado congruente, recodemos que consideramos que el *Worker* es una máquina sin estado.

Si se está utilizando un esquema de **Replicación de Datos centralizada**, los datos que previamente habían sido procesador por el *Worker* que ha fallado están almacenados en el *Master* y replicados en el Protector del *Master.*

En el caso de estar utilizando un esquema de **replicación distribuida de datos** en el *Worker*, una vez el *Master* marca como libre la tarea perdida para que se reasigne a otro *Worker*, debe hacer una Replicación de Datos de los resultados que el *Worker* que había

fallado había computado, ya que además de perder la capacidad de computo del *Worker*, hemos perdido la memoria estable donde estaba la Replicación de Datos de las tareas que el *Worker* había finalizado y debemos garantizar la durabilidad de estos datos.

En este caso, de replicación distribuida de datos, es necesario enviar los resultados que estaban almacenados en este *Worker* a otro *Worker*, utilizando para ello la Replicación de Datos que existe en el *Master* y especificar en la tabla que estos resultados los tiene otro nodo; el *Worker* elegido es el que tiene el Protector del *Master* activo, ya que de este modo se facilita el *overhead* de recuperación en el caso de un fallo posterior del nodo *Master*.

- **Recuperación de fallo en un Nodo Worker con Protector del *Master***

Si se produce un fallo en un Nodo *Worker* con Protector del *Master* (PMT), el *Middleware* en este caso tiene que tratar tanto la perdida de un *Worker*, como la perdida del PMT. El tratamiento del *Worker* es el mismo que el analizado en el apartado anterior.

Una vez realizado la re-configuración y recuperación del *Worker* se debe activar otro Protector del *Master*. Para ello una vez el MIDMASTER actualiza las tablas, activa un nuevo Protector del *Master* y debe replicarle las tablas de configuración, los datos iniciales y los resultados de la aplicación.

En el caso del sistema de protección centralizado, debe replicar todos los resultados del *Master*.

Si el sistema de protección configurado es el distribuido, además debe replicar los resultados del nodo *Worker* que ha fallado.

- **Recuperación de fallo en un Nodo Gestor de Comunicación**

Si se produce un fallo en un Nodo Gestor de Comunicación (CM) debemos tener en cuenta que este fallo afecta al *cluster* local tanto como a la comunicación con el *cluster* remoto.

En el caso de fallo se debe seleccionar otro nodo para que actúe como Gestor de Comunicación y esto debe comunicarse al Subcluster correspondiente. En ese momento ambos Gestores de Comunicación deben comenzar a comunicarse a través de la red a larga distancia y deben sincronizar sus estados.

Una vez el *Master* aísla el Nodo Gestor de Comunicación que falló. Se debe actualizar la tabla de configuración para no volver a comunicarse con este Gestor de Comunicación. El Gestor de Comunicación como *Worker* que es, lo consideramos una máquina sin estado a pesar de que dispone de unas tablas internas para saber: el estado de la comunicación con el otro *cluster*, las tareas enviadas y los resultados recibidos. Esa información está replicada en

ambos Gestores de Comunicación (CM), por lo tanto el nuevo Gestor de Comunicación debe recuperar esta información del Gestor de Comunicación en el Subcluster y volver a un estado congruente. Por lo tanto, una vez reconfigurado el *cluster* y activado el nuevo Gestor de Comunicación (CM), éste se comunica con el otro CM para poder actualizar la tabla de configuración, de este modo conoce el estado de los datos en el otro *cluster* (Subcluster).

- **Recuperación de fallo en Nodo *Master***

Si el MIDWORKER del Protector del *Master* (PMT) activo, detecta un fallo en un Nodo *Master*, es decir, ha dejado de recibir los *heartbeat* del *Master*, tal y como hemos visto el *Middleware* debe realizar las siguientes acciones: activar el PMT para que actúe como *Master*, desactivar el "*Worker*" que está en el nodo junto con el Protector del *Master* y activar otro Protector del *Master* (PMT) que proteja al nuevo *Master*. Para ello ejecuta los siguientes pasos, una vez se ha diagnosticado el fallo por el Protector del *Master*:

1. El controlador del Nodo (NC) en el Nodo *Worker* que contiene el Protector del *Master*, activa la funcionalidad de *Master*. Como el Protector del *Master* tiene la copia de las tablas del *Master*, con los índices reales, tiene el estado del *cluster* actualizado. Si existe más de un Protector del *Master* activo hay que enviarles un *heartbeat* para que no pasen a las fases de recuperación y configuración.
2. El “Nuevo” *Master* aísla el Nodo *Master* que falló. El proceso "*Worker*", que estaba en el nodo que ahora asumirá las tareas del *Master*, tiene que ser desactivado, para ello actualiza la tabla de configuración, hace la recuperación del proceso *Worker* que ha desactivado tal y como hemos visto.
3. Reconfigura la máquina, avisando a los *Worker* y Gestor de Comunicación para que a partir de ahora se comuniquen con el nuevo *Master*. Para hacer esta re-configuración es necesario parar los nodos y re-configurar el *cluster*.
4. El “Nuevo” *Master* analiza el estado global de los *Worker* y del Gestor de Comunicación. O sea, analiza en que punto estaba el sistema cuando falló el *Master*, vuelve a un estado consistente
5. El “Nuevo” *Master* pide a los *Worker* y al Gestor de Comunicación, que reenvíen el trabajo que estaban realizando y que aún no había sido recibido por el *Master* que falló. A partir de aquí se continúa el procesamiento.
6. Activa un nuevo Protector del *Master* (PMT), envía una réplica de las tablas de estado y configuración al nuevo PMT. El nuevo *Master* una vez actualizada y replicadas las tablas comienza su funcionamiento normal. El MIDMASTER

escucha los *heartbeats* enviados por los "*Worker*", y envía sus *heartbeats* al nuevo Protector del *Master*.

7. Para la recuperación del *Master*, el *Worker* y el Protector del *Master* se deben tener en cuenta la política de protección que se estaba realizando, centralizada o distribuida. Como hemos visto en la recuperación del *Worker* y del Protector del *Master*, si la protección era distribuida es necesario también recuperar la Replicación de Datos, para mantener, a partir de la recuperación de este fallo, el mismo nivel de tolerancia a fallos que definió el usuario.

4.5. La tolerancia a fallos en el multicluster

Hasta ahora hemos visto como se tratan los fallos en cada uno de los *clusters*. Ahora vamos a analizar la influencia del fallo de un nodo en un *cluster* determinado, en el multicluster.

Cada uno de los *cluster,* como hemos visto, dispone de nodos suficientes para soportar una tolerancia a fallos local, también dispone de unos procesos dedicados a la comunicación entre *clusters* (Gestor de Comunicación). En caso de perdida de un *cluster* entero, el *Master* habría ido recibiendo el trabajo realizado y tendría los datos replicados en otro nodo, dependiendo del tipo de protección, centralizada (tendría una copia en el Protector del *Master*) o distribuida (la copia estaría en el Gestor de Comunicación local).

Si fallan todos los nodos de un *cluster*, el *Master* dispondría del trabajo recibido de dicho *cluster* hasta el momento en el que se produce el fallo y cuando se hayan procesado el resto de tareas, se analiza el estado global del sistema, se detecta que faltan las tareas asignadas a un *cluster* concreto, se verifica que no existe comunicación con dicho *cluster,* se diagnóstica el fallo, pasándose a asignar el resto del trabajo que no ha sido recibido a otros *Worker*.

Siempre que existen fallos en un *cluster* remoto, es decir, han fallado nodos en el *cluster* remoto, con la consiguiente perdida de capacidad de computación, puesto que la cantidad de trabajo inicial asignada estaba relacionada con la capacidad de cómputo, el sistema global se verá afectado, ya que la pérdida de prestaciones de cada uno de los *cluster* influye directamente en la pérdida de prestaciones del *cluster* global. Debemos distinguir la perdida de prestaciones de un *cluster* con la perdida total del *cluster*, para

no aislar a un *cluster* que no ha fallado, sino que han cambiado sus prestaciones debido al fallo de alguno de sus nodos.

4.6. Tablas de configuración/estado

Tal y como hemos visto, la protección de los *Worker* se realiza a través de la creación de una tabla de configuración /estado (Tabla 4-1), en la que se registra la asignación de tareas a los nodos (configuración del *cluster*) y el estado de dicha tarea.

Durante la ejecución el *Middleware* del *Master* (MIDMASTER), es el encargado de la modificación de tablas (para mantener la configuración y el estado global del sistema) y de la re-configuración o reasignación de tareas a otro *Worker* en caso de fallo.

Tal y como vimos para adaptar la aplicación al entorno multicluster es conveniente considerar la nueva heterogeneidad que añade la organización jerárquica del *Master/Worker*, se distingue la carga de trabajo para una tarea de "*Worker*", que se distribuye en el *cluster* local de una forma dinámica, de la carga de trabajo total que se tiene para distribuir para cada uno de los Subcluster, que se distribuye inicialmente de un forma semiestática. Desde el punto de vista del *Middleware* de tolerancia a fallos debemos tener en cuenta que puede existir este comportamiento de la aplicación en la que se definen estas dos granularidades, es decir, el funcionamiento normal de la aplicación que consideramos es que un *Worker* tiene una tarea, cuyos resultados envía al *Master* cuando termina una tarea completa. Un Subcluster tiene un trabajo, el Submaster de dicho *cluster*, debe enviar resultados al *Master* cada vez que termina una tarea y no debe esperar a realizar todo el trabajo.

En el momento de manejar las tablas de configuración, se considera sólo la tarea, en el caso de los Subcluster se asignan varias tareas (todas las que componen un trabajo) al Gestor de Comunicación, que será el encargado de enviarlo al Subcluster. (Tabla 4-3) Cuando se recibe un resultado se cambia el estado de dicha tarea, las recepciones siempre se van realizando tarea a tarea.

En caso de fallo de un *Worker*, o Gestor de Comunicación, en la fase de re-configuración del *cluster* se asignan todas las tareas que no han sido acabadas a otro nodo *Worker* o a otro Gestor de Comunicación.

Si tenemos una protección distribuida, en la fase de recuperación, se buscarán en la tabla todos los trabajos acabados por el *Worker* o Gestor de Comunicación y se enviará una copia de sus resultados al nodo donde está el Protector del *Master*. Debemos recordar que en el

caso de protección distribuida el nodo que figura en la tabla no es el nodo que realizó el cómputo, sino el nodo que contiene la Replicación de los resultados de dicha tarea.

Es importante que el *Middleware* pueda identificar las tareas de los *Worker* (datos enviados y datos recibidos para cada tarea para que se pueda actualizar el estado de las tareas)

Tabla 4-3: Configuración/estado

	Configuración		Estado	
Id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	T1	n1	No	No
2	T2	n2	Si	Si
3	T3	n3	Si	No

Significado
Fallo
Acabado
Enviado

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	T1	n4	Si	No
2	T2	n3	Si	Si
3	T3	n2	Si	Si
4	T4	n1	Si	No
5	T5	n1	Si	Si
6	T6	n1	Si	No
7	T7	n1	Si	Si

Hay que construir la tabla inicial con las identificaciones de tareas, esto lo debe tener en cuenta el programador en la aplicación. La inicialización de índices: para la generación de toda la tabla de tareas (esto se usará para el almacenamiento de datos y actualización de tablas). Se ejecuta en el *Master* para generar las tablas: es una función “INIT” que se ejecuta en todos: Ini tarea (índice); Fin tarea (índice); Ini resultado (índice); Fin resultado (índice).

4.7. Metodología

Una vez visto como funciona el esquema de tolerancia a fallos basado en Replicación de Datos para un multicluster *Master*/Worker, presentamos a continuación algunas consideraciones que se deben tener en cuenta para utilizar el *Middleware* implementado para soportar FTDR, es decir, proponemos a continuación la metodología que debe ser seguida por el programador para utilizar FTDR, para que partiendo de una aplicación con un modelo de ejecución *Master/Worker*, el programador pueda configurar y usar el mecanismo de tolerancia a fallos propuesto.

El programador inicialmente debe tener en cuenta la estructura global del multicluster y adecuar la aplicación a dicha estructura especificando las estructuras de datos que deben de ser replicadas y el *buffer* de datos, tal y como vimos en el capítulo de arquitectura.

Es necesario construir la tabla inicial con las identificaciones de las tareas, para ello es importante, para que el *Middleware* pueda gestionar la tolerancia a fallos de modo automático, etiquetar las tareas de los *Worker* (datos enviados y datos recibidos), porque tal y como hemos visto tenemos que reflejar el estado de cada tarea (enviado, acabado) y el nodo *Worker* involucrado, en la tabla de configuración/ estado, para ello se interceptan los mensajes enviados y recibidos (*send* y *receive*).

Interceptando los *send* y *receive* de cada *Master*, se identifica cuando se envía una tarea, además es necesario identificar el tipo de mensaje: iniciar tarea; finalizar tarea; recibir tarea, recibir fin de tarea. De este modo se puede actualizar correctamente la tabla y realizar su protección (Replicación de Datos), enviando las tablas a los Protectores del *Master*.
Tal y como hemos visto, tenemos dos alternativas para la protección, centralizada (la replicación de todos los datos del *Master* están en el Protector del *Master*) y la distribuida donde cada *Worker*, almacena una copia de sus resultados, de forma que en este caso se debe enviar los resultados al *Master*, y se deben interceptar para almacenar las copias, estas copias deben estar también etiquetadas, porque el Protector del *Master*, cuando diagnóstica un fallo del *Master*, en el momento de la recuperación, pide a los *Workers* el trabajo realizado, en base a esto restablecerá las tablas de resultados. Por lo tanto, también es necesario etiquetar los datos a almacenar.

4.7.1. Redundancia de procesos

Consideramos que existe una redundancia de procesos, es decir, la aplicación *Master*/*Worker* y el *Middleware*, todos son procesos residentes. En cada nodo del *cluster* existen dos procesos activos (Figura 4-2): el del usuario (*Master* o *Worker*), el del *Middleware* de gestión de las comunicaciones y de tolerancia a fallos. En caso de fallo el sistema se reconfigura y se define el rol de cada máquina, o sea sus procesos activos.

Para que esta redundancia de procesos se pueda utilizar efectivamente en la re-configuración del *cluster*, todos los nodos de procesamiento (NPRO) poseen dos placas de red. Una para la comunicación a la red local y otra para la conexión con la WAN. De este modo, como la imagen del *software* (*Master*, *Worker*, Gestor de Comunicación, *Middleware*) debe estar en todos los NPRO, todos los NPRO pueden asumir cualquier

papel.

Todos los NPRO deben poseer un controlador del nodo (Figura 4-4). El Controlador del Nodo (NC), es quien determina los procesos activos y es el mecanismo que permite configurar, reconfigurar cada NPRO. Este controlador es parte del *Middleware* y debe estar siempre activo.

El *Middleware* está formado por una serie de procesos diferenciados que también están replicados en todos los nodos. El *Middleware* ejecutando en Nodo *Master* (MIDMASTER), hace funciones de tolerancia a fallos de protección, detección, diagnóstico, recuperación y re-configuración.

El *Middleware* ejecutando en Nodo Worker (MIDWORKER) y en el Gestor de Comunicación (MIDCM), hacen detección y participa en la re-configuración en caso de fallo del Nodo *Master*. En caso de tener un esquema de protección distribuida, los *Worker* también participan en la fase de protección, ya que deben guardar una copia de los resultados, para que éstos estén replicados.

El Middleware en Gestor de Comunicación es el encargado de ser el Interceptor de comunicación WAN, que posee Gestor de Comunicación (CM), que hace control y retransmisión de mensajes entre los *clusters*.

El proceso del *Middleware* interceptor del *Master*, que posee el Interceptor del *Master* (MI), que gestiona: datos / tareas, Detecta los fallos en Nodo *Worker* y Nodo Gestor de Comunicación a través de la escucha de *heartbeat*, previene: almacena y replica en caso de utilizar una protección centralizada; reconfigura el *cluster* en caso de fallos en los *Workers* y gestiona las tablas de tolerancia a fallo.

El Middleware del *Worker*, que posee un Interceptor Simple del *Worker* (SWI), que para la fase de detección envía *heartbeat* al *Master* para indicarle que sigue vivo.

En el caso del Interceptor del *Worker* Protector (PWI), participa en diferentes fases de la tolerancia a fallos:

La **protección** del *Master*, es decir, se encarga de la Replicación de Datos iniciales del *Master*, de los datos de su *Middleware* (guardando copia de las tablas de estado y configuración) y, en el caso de que se utilice una protección centralizada una copia de los resultados recibidos o guardando la Replicación de Datos de los resultados que ha enviado un *Worker* que ha fallado, en caso de una protección distribuida.

La **detección** de fallo del *Master* a través de la escucha de *heartbeat* y el diagnóstico de fallo del *Master* cuando el número de *hearbeat* que han fallado está por encima del umbral

especificado en la configuración.

El Interceptor del *Worker* Protector (PWI) también hace la **re-configuración** en caso de fallo del *Master* y la **recuperación** de los datos, que depende del esquema de protección utilizado:

- Protección centralizada: las tablas en el *Master* y el Protector del *Master* deben estar actualizadas. Tienen los mismos datos (iniciales y resultados recibidos y copia de las tablas de configuración/estado).
- De forma distribuida, en este caso en la fase de recuperación el Protector del *Master* pide de los *Workers* y Gestor de Comunicación el resultado que tienen almacenado del trabajo ya realizado.

4.7.2. Requisitos de la aplicación

La aplicación debe estar escrita para un modelo de ejecución *Master*/*Worker* donde el número de nodos de procesamiento (NPRO) puede ser variable, o sea, debe tener la capacidad de permitir retirar nodos de forma dinámica.

Debe tener en cuenta que es un sistema heterogéneo y que se va a ejecutar en un entorno multicluster

Los Nodos pueden asumir distintas funciones (*Master* y *Worker*) e incorporar distintas funciones de *Middleware* (Protector del *Master* y Gestor de Comunicación), además de las funciones propias del nodo (MIDMASTER, y MIDWORKER). Por otro lado, como los *Workers* tienen distintas capacidades, pueden requerir diferentes cargas de trabajo (*workloads*).

El usuario al lanzar la aplicación debe configurar los parámetros de la tolerancia a fallos, tales como: el nivel de protección de fallos (número de fallos simultáneos en el mismo *cluster* que se quieren soportar), el tiempo para los procesos de detección de fallos, el número *C* de fallos simultáneos frente a los que se quiere proteger, la frecuencia del *hearbeat* y el tiempo entre *heartbeats*.

4.8. Prestaciones

La arquitectura está justificada desde el punto de vista funcional de la tolerancia a fallos (permite acabar las aplicaciones cuando fallo un nodo), además es importante tener en cuenta como afecta a las prestaciones.

En caso de fallo de un nodo, el coste no afecta sólo a la perdida del *hardware*, debemos

tener en cuenta que cuando una aplicación falla, el coste es mayor porque se puede perder la computación.

Existe un coste o pérdida de prestaciones debido a los recursos de computación utilizados por la arquitectura de tolerancia a fallos, esta pérdida de prestaciones debe ser el mínimo posible. Por otro lado, la tolerancia a fallos previene la pérdida de computación cuando existe fallo.

Existe un compromiso entre la tolerancia a fallo y la granularidad, o sea, nos preocupamos por la cantidad de paquetes que el *Worker* procesa y que perdida tendríamos en caso de fallo. Debemos tener un balanceo de carga correcto en función de lo que queremos.

La distribución inicial de trabajo entre los *clusters* es semiestática y está caracterizada, en función de las prestaciones de los nodos del *cluster*, es decir, se puede prever en función de las prestaciones de los nodos del *cluster*, pero se puede cambiar a lo largo de la ejecución, para adaptarnos a las características dinámicas del funcionamiento, como pueden ser pérdidas de nodos o variabilidad en el funcionamiento de la red. La carga de trabajo inicial se fija en función de las características de los nodos y se adapta en función de la ejecución en el multicluster.

Se utilizan como medidas de prestaciones el tiempo de ejecución, además se utiliza otras medidas de prestaciones adicionales como la alta disponibilidad (*high availability*) que está relacionada con la presencia de redundancia en el sistema (*software*) para reducir el tiempo de inactividad y la degradación de las prestaciones ante un fallo.

Consideramos el *overhead* introducido por el sistema de tolerancia a fallos, tanto en ausencia de fallos, como en presencia de fallos.

Tal y como hemos visto, las técnicas utilizadas para proveer los sistemas tolerante a fallos, provocan un coste (*overhead*) adicional. La técnica que nosotros utilizamos, también consume recursos computacionales y recurso de Entrada/Salida, reduciendo las prestaciones del sistema como un todo. El *overhead* añadido es debido a la cantidad extra de mensajes que es generada e introducida en el sistema durante la ejecución del modelo de tolerancia a fallo, aumentando la latencia de la red: tiempo de almacenamiento, tiempo consumidos por las rutinas de tolerancia a fallos de comprobación del estado de los nodos del sistema y detección de fallos. Además las prestaciones del *cluster* son menores cuando un nodo fallo ya que dispone de un recurso menos.

Comprobamos que a pesar del *overhead* añadido, es mucho mejor utilizar FTDR que

simplemente reiniciar la aplicación desde el principio.

4.9. Validación

El gran problema en el área de tolerancia a fallos es saber si la técnica implementada resulta realmente un aumento de la garantía de funcionamiento. Como en la mayor parte de los sistemas las tasas de fallos son bajas y los fallos suceden de forma aleatoria e incontrolable, el problema consiste en evaluar si la técnica diseñada está realmente tolerando los fallos para las cuales fue diseñada, sin necesidad de esperar meses o años para que los fallos realmente aparezcan (por ejemplo en aviónica, que es un sistema crítico, la tasa de fallo esperada es una en cada un poco más de millón de años). La principal herramienta para la validación es la Inyección de Fallos [4] [39] [54].

La Inyección de Fallos es relativamente popular en sistemas aislados, pero es reciente, el desarrollo de herramientas de Inyección de Fallos para sistemas distribuidos y sistemas paralelos y aún existe poca experiencia acumulado en el tema [5].

Para validar el modelo realizamos experimentación exhaustiva probando distintos tipos de fallos, inyectados de forma sistemática.

Ejecutamos experimentos con el objetivo de medir cuanto *overhead* es añadido al sistema (Figura 4-11) en las distintas fases de la tolerancia a fallos:

- Protección por Replicación de Datos (T1-T2)

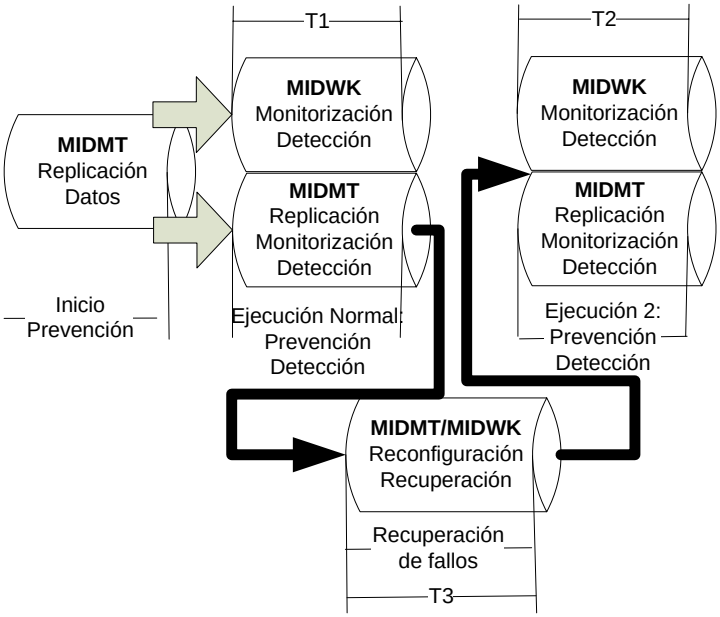

Figura 4-11: Validación del modelo

- Detección y diagnóstico (T1-T2)
- Recuperación y reconfiguración (T3) del *cluster* después de un fallo.

Cuando se diagnóstico un fallo la recuperación es prioritaria. Mientras el *Master* recupera/reconfigura, el ideal es que "todo" esté parado el mínimo tiempo posible. Es importante observar que el tiempo de recuperación es dependiente del número de nodos que se quedarán en el *cluster*.

Para que las pruebas sean sistemáticas, desde el punto de vista de la instrumentación/monitorización, incluimos en el sistema un supervisor/inyector de fallos (ver apartado 5.2.1), para desactivar nodos (provocar fallo) en puntos/momentos previamente determinados.

4.10. Conclusiones

En este capitulo se ha presentado el modelo de tolerancia a fallos usando Replicación de Datos para arquitecturas multicluster de computadores, geográficamente distribuidos.

Este modelo de tolerancia a fallos, denominado FTDR (*Fault Tolerant Data Replication*), no se utiliza componentes adicionales y previne fallos en cualquiera de los nodos de cómputo del sistema, generando para ello la redundancia necesaria para garantizar la finalización correcta del trabajo, aunque con perdidas de prestaciones debido al *overhead* añadido para la recuperación y re-configuración del sistema.

Trabajamos en un entorno multicluster y cada uno de los *cluster* utiliza recursos propios para la tolerancia a fallos, de forma que la tolerancia a fallos en un *cluster* se **detecta**, **previene**, **recupera** y **reconfigura** en el propio *cluster*, eso proporciona la escalabilidad

Tabla 4-4: Esquema de protección - replicación centralizada de datos

De	Acción	Para
MT	Protege	WK y PMT
MT	Replica	PMT
SC	Replica	CM
MT	Detecta	WK y PMT
PMT	Detecta	MT
MT	Diagnostica	WK y PMT
PMT	Diagnostica	MT
MT	Reconfigura	WK y PMT
PMT	Reconfigura	MT
MT	Recupera	WK y PMT
PMT	Recupera	MT

necesaria para que tengamos un sistema de tolerancia a fallos eficaz.

Podemos resumir nuestro esquema de tolerancia a fallo con replicación centralizada de datos en la Tabla 4-4 y con replicación distribuida de datos en la Tabla 4-5, donde nosotros mostramos quien sea el responsable (**De**) por cada una de las actividades (**Acción**) y quien es el destino (**Para**) de esta actividad.

Tabla 4-5: Esquema de protección - replicación distribuida de datos

De	**Acción**	**Para**
MT	Protege	WK y PMT
SC	Replica	CM
MT	Detecta	WK y PMT
PMT	Detecta	MT
MT	Diagnostica	WK y PMT
PMT	Diagnostica	MT
MT	Reconfigura	WK y PMT
PMT	Reconfigura	MT
MT	Recupera	WK, PMT y CM
PMT	Recupera	MT

Capítulo 5

Validación experimental del modelo

Este capitulo muestra las pruebas experimentales con replicación centralizada y replicación distribuida de datos para probar el funcionamiento en la práctica del modelo de tolerancia a fallos propuesto.

5.1. Introducción

En este capítulo se presentan los resultados de la evaluación experimental del modelo de tolerancia a fallos descrito en el Capitulo 4. Para realizar el análisis de la influencia del sistema FTDR propuesto, se ha diseñado un conjunto de pruebas que permite comprobar el *overhead* añadido, tanto en ausencia como en presencia de fallos. También es importante analizar la influencia de cada uno de los parámetros configurables por el usuario, tales como el tiempo de *heartbeat*, el número de fallos para el diagnóstico, el número de fallos soportados simultáneamente, el esquema de replicación de datos centralizado o distribuido.

Se han realizado pruebas experimentales con replicación centralizada y replicación distribuida de datos para probar el funcionamiento en la práctica de la propuesta realizada, las pruebas se han realizado de acuerdo con la metodología que se describirá a continuación.

Para que las pruebas sean sistemáticas se ha diseñado e implementado un sistema supervisor/inyector de fallos. Con este inyector podemos controlar los momentos en los que queremos provocar un fallo; una vez ha ocurrido el fallo, podemos validar el estado del sistema, ya que éste es conocido, puesto que tal y como hemos visto, el comportamiento del sistema frente a un fallo es predictivo.

Por otro lado, desde el punto de vista de prestaciones nos interesa cuantificar el *overhead* añadido. Para analizar el *overhead* de FTDR en ausencia de fallo, se ejecuta la aplicación sin el *Middleware* FTDR, midiendo el tiempo de ejecución (T_{AN}), esto se compara con la ejecución de la aplicación con el *Middleware* FTDR, sin inyectar fallos en ninguno de los nodos (T_{AN}^{sf}), en este caso están funcionando las fases de protección, basada en generar redundancia utilizando Replicación de Datos y la fases de detección basada en el envío de *heartbeat*.

Para analizar el comportamiento y *overhead* de FTDR en presencia de fallos en los distintos nodos del multicluster, además de hacer pruebas de ejecutar la aplicación con el *Middleware* FTDR, con fallos aleatorios provocados por desconexión de la placa de red del nodo, y comprobar que el sistema acaba correctamente. Se han provocado fallos sistemáticos, utilizando el inyector de fallos, en los distintos nodos después de enviar o recibir 50%, 75% y 95% de los resultados, hemos considerado que si el sistema está al inicio de la ejecución es mejor rearrancar la aplicación que poner el mecanismo de recuperación / re-configuración. Para ello se ejecuta la aplicación con el *Middleware* FTDR,

inyectando fallos simples (uno cada vez) en los nodos del *cluster* ejecutando diferentes roles (*Master*, *Worker*, Protector del *Master*, Gestor de Comunicación), cuando el Nodo *Master* (MMT) está en diferentes fases, para cada uno de los casos medimos el *overhead* introducido, (T_{AN}^{cf}), analizando dicho *overhead,* teniendo en cuenta la influencia de cada una de las etapas de la tolerancia a fallos y teniendo en cuenta la perdida de prestaciones por la perdida de un nodo ($T_{A(N-1)}$).

5.2. Multicluster utilizado para la validación experimental

El multicluster utilizado en las pruebas es el descrito en el Capitulo 3, disponiendo de un cluster en España y otro en Brasil, se han configurado cluster con un gran nivel de heterogeneidad y para medir prestaciones se han utilizando los computadores con características similares, de forma que el comportamiento del sistema no fuera tan dependiente de las características del nodo que falla.

Como aplicación de pruebas utilizamos un programa de multiplicación de matrices, que es un núcleo importante del álgebra lineal y fue seleccionado porque es un algoritmo escalable.

Para virtualizar la máquina paralela de paso de mensajes se ha utilizado MPI en una arquitectura, *Master/Worker* jerárquica geográficamente distribuidos.

Tal y como hemos visto en el Capitulo 3, esta arquitectura, además de los nodos *Master* y *Worker,* interconectados a través de una red local dedicada, para conectar *cluster* geográficamente distribuidos utiliza un elemento adicional que gestiona la comunicación (CM) entre el *cluster* principal y los Subcluster.

A continuación analizaremos algunas consideraciones que debemos tener en cuenta en la validación experimental.

Tal y como hemos visto en el Capitulo 2, **MPI tiene algunas restricciones para la tolerancia a fallos**. Una restricción que se debe tener en cuenta es que no se debe utilizar "MPI_Recv" en la aplicación, ya que, al ser ésta una función bloqueante, y en caso de que ocurra un fallo en un nodo emisor, o lo que es lo mismo, debido a una demora en la recepción de un paquete, la aplicación se quedará en *blocking* ("congelada") indefinidamente esperando ese dato.

Para resolver este problema es necesario utilizar la función MPI_Iprobe. Esta función es "no bloqueante" y funciona de la siguiente manera: verifica si hay un dato a ser recibido por

un determinado nodo (rank + communicator + *tag*). En caso de recibir un "*true*", significa que el MPI_Recv puede ejecutarse sin problemas (Figura 5-1).

```
...
while(!aux1)
  {
    MPI_Iprobe(MPI_ANY_SOURCE, TAG_DATA, MPI_COMM_WORLD, &aux1, &statusWorker);
  }
...
```

Figura 5-1: Función MPI_Iprobe

Para inyectar fallos aleatorios se puede quitar el cable que conecta la interface de red al *hub*/*switch* de un nodo o apagar la maquina.

Para inyectar fallos controlados en un nodo del *cluster*, a través del sistema operativo utilizamos la directiva de sistema "sudo /sbin/ifdown <interface>".

Estas dos formas de inyectar fallo en los nodos del *cluster*, no ocasiona la "muerte" del "Mundo MPI". Por otro lado, si "matamos" el proceso de un *Worker*, utilizando el comando "kill <proceso>" el "Mundo MPI" se "muere".

Como **aplicación de pruebas** utilizamos un programa de multiplicación de matrices cuadradas [10] [31], la distribución de datos se realiza en bloques de filas y columnas de forma dinámica. Así, dado dos matrices *A* y *B* como datos de entrada, después de ejecutar el algoritmo de multiplicación, obtendremos como resultado una matriz *C*, también cuadrada (Figura 5-2).

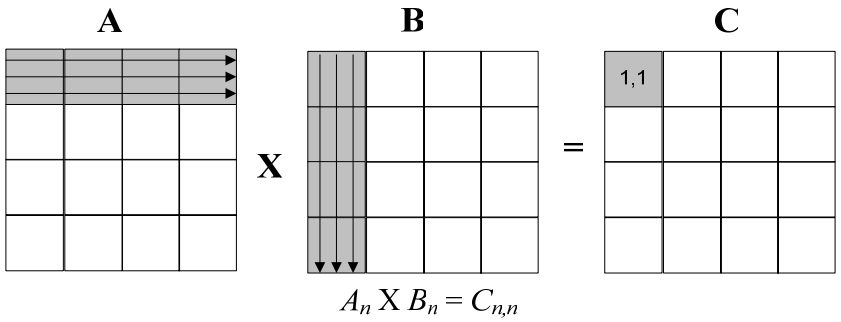

Figura 5-2: Matriz cuadrada

El *Master* envía *p* bloques de *n* filas de la matriz *A* y *q* bloques de *n* columnas de la matriz *B* a los *Workers* locales y un bloque mayor al Gestor de Comunicación local. El Gestor de Comunicación local envía los bloques al Gestor de Comunicación remoto, para que los envíe al Submaster (SMT), que distribuye el trabajo entre los *Workers* remotos.

A medida que los *Workers*, tanto locales como remotos concluyan sus trabajos, envían el

resultado al *Master*: Los *Workers* locales envían al *Master* Principal y los *Workers* remotos lo envían al Submaster (SMT), que utiliza el Gestor de Comunicación para enviar los resultados al *cluster* Principal (matriz *C*) (Figura 5-2).

El tamaño de las matrices utilizadas son: 1000x1000, 2000x2000 y 3000x3000, divididas en bloques de 200 filas X 200 columnas como tarea a enviar a cada *Worker*. Para calcular el trabajo que se debe enviar al Subcluster, se realiza en función de la relación computo/comunicación obtenida a través de la caracterización del multicluster. Tal y como hemos visto en el apartado 3.3, se ha estudiado la capacidad de computo que puede aportar el Subcluster en Brasil y en base a ello se ha decido dividir la carga de trabajo de forma que el 90% de los paquetes son procesados en el *cluster* local y el 10% restante se envía para ser computado en el *cluster* remoto, ya que tal y como hemos vistos la capacidad de computo en *cluster* remoto en este caso, es menor, y además tenemos que tener en cuenta la latencia de la comunicación.

5.2.1. Instrumentación / monitorización

Es importante inyectar fallos que permitan comprobar el funcionamiento correcto del sistema y que el sistema pueda tolerar fallos que ocurran en nodos que están en distintas fases tales como: realizando cómputo, enviando mensajes, recibiendo mensajes, realizando función de tolerancia a fallos (Replicación de Datos).

Para que las pruebas sean sistemáticas desde el punto de vista de la instrumentación/monitorización, incluimos en el sistema un supervisor/inyector de fallos, para desactivar nodos, es decir, provocar fallos de nodos, en distintos momentos, previamente determinados.

Respecto a las medidas de rendimiento analizadas, se ha monitorizado el tiempo de ejecución para analizar el *overhead* introducido por el sistema de tolerancia a fallos, También se ha monitorizado el funcionamiento global del sistema, monitorizando la configuración del *cluster* y el estado global del sistema.

Para determinar los momentos de Inyección de Fallos se ha seleccionado tres instantes, cuando la aplicación está al 50%, 75% y 95% del cómputo realizado. Se utilizan *logs* para visualizar los resultados de la monitorización.

5.2.2. Configuración del *cluster*

Antes de ejecutar una aplicación es necesario realizar la configuración del *cluster*, recordemos que los procesos están replicados en todos los nodos y es necesario activar en

cada nodo el papel concreto, por lo tanto es necesario antes del arranque de la aplicación, comienza ejecutándose el controlador del nodo (CN) que permite que el programador o administrador configure el *cluster,* en nuestro caso, utilizando los nodos descritos en el apartado 3.4 hemos utilizado diferentes configuraciones, con objetivo de tener un cluster altamente heterogéneo y otro lo más homogéneo posible en base a los nodos disponibles.

Para la configuración del *cluster* homogéneo se han utilizado los nodos mostrados en la Tabla 5-1.

Tabla 5-1: Configuración del cluster homogéneo

Nodo Local	**Función**	**Localización**	**Tag**
nl0	MMT	UAB	aoquir2
nl1	CM	UAB	aoquir12
nl2	WK/PMT	UAB	aoquir1
nl3	WK	UAB	aoquir7
nl4	WK	UAB	aoquir10
nl5	WK	UAB	aoquir11
Nodo Remoto	**Función**	**Localización**	**Tag**
nr0	SMT	UCSAL	infoquir5
nr1	CM	UCSAL	infoquir2
nr2	WK/PMT	UCSAL	infoquir3
nr3	WK/PMT	UCSAL	infoquir6
nr4	WK	UCSAL	infoquir1
nr5	WK	UCSAL	infoquir7

También se han realizado pruebas utilizando una configuración muy heterogénea (Tabla 5-2), donde la activación de procesos en nodos (*mapping* de procesos), es una tarea más crítica, ya que influye significativamente en el rendimiento del sistema, sobre todo teniendo

Tabla 5-2: Configuración del cluster heterogéneo

Nodo Local	**Función**	**Localización**	**Tag**
nl0	MMT	UAB	aoquir12
nl1	CM	UAB	aoquir2
nl2	WK/PMT	UAB	aoquir8
nl3	WK/PMT	UAB	aoquir3
nl4	WK	UAB	aoquir1
Nodo Remoto	**Función**	**Localización**	**Tag**
nr0	SMT	UCSAL	infoquir5
nr1	CM	UCSAL	infoquir2
nr2	WK/PMT	UCSAL	infoquir3
nr3	WK/PMT	UCSAL	infoquir6
nr4	WK	UCSAL	infoquir1
nr5	WK	UCSAL	infoquir7

en cuenta que esto requeriría un ajuste más fino de balanceo de carga.

En principio el criterio seguido tal y como comentamos en el capítulo de arquitectura (Capitulo 3), los nodos más potentes son utilizados como *Workers*, y los nodos con más capacidad de almacenamiento se utilizan para el *Master* y el Protector del *Master*.

Además de configurar el cluster se debe configurar el nivel de tolerancia a fallos, es decir, frente a cuantos fallos simultáneos que se desea proteger el sistema, por defecto este parámetro se inicializa a 1.

Se debe especificar el tiempo de *heartbeat*, que tal y como hemos comentado es necesario para definir el tiempo de detección de fallos, este parámetro tiene una influencia directa en la latencia de tratamiento del error.

También es necesario especificar si se desea una replicación de datos centralizada o distribuida.

5.2.3. Tolerancia a fallos: detalles de implementación

Tal y como hemos visto, el objetivo de la tolerancia a fallos es que cuando ocurra un fallo en un nodo, el procesamiento continúe hasta el final y, además, buscamos que del trabajo realizado hasta ese instante por el nodo que falla, se pierda el mínimo posible, para ello, se almacena en la tabla de configuración/estado la identificación de las tareas enviadas a los *Workers* y Gestor de Comunicación. Esta tabla es creada/mantenida dinámicamente por el *middleware* del nodo *Master*, almacenada en el nodo *Master* y replicada en los Protectores del *Master* (PMT). En la Tabla 5-3 se muestran los distintos estados de una tarea tal y como comentamos en el capítulo anterior.

Tabla 5-3: Combinaciones de estado

Estado		
Enviado	**Acabado**	**Significado**
-	-	No generado
Si	No	Enviado
No	No	Fallo
Si	Si	Acabado
No	Si	No se puede

Tal y como vimos, es necesario para la recuperación/re-configuración, identificar las tareas (*A* y *B*) responsables de generar un determinado resultado (*C*), para ello se "encapsulan" las tareas (*A* y *B*) e identifican por medio de un número secuencial (*id*), único

para cada tarea enviada a los *Workers* y Gestor de Comunicación (CM), de esta forma se

Tabla 5-4: Tabla de configuración/estado (0)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	nl2	Si	No

almacena este paquete en la tabla de configuración/estado (Tabla 5-4) en la que además, se especifica el nodo que está realizando el proceso y su estado (Enviado, Acabado, fallo o no generado) para todos los *Workers* del *cluster* local y para el Subcluster se indica cada una de las tareas que han sido enviadas al nodo Gestor de comunicaciones (CM: nl1). En la tabla de configuración/estado del *Master* (SMT) del *cluster* remoto (Subcluster) se especificará la asignación concreta de sus nodos.

Así, los *Workers* y Gestor de Comunicación recibirán las tareas (bloques de 25 filas de *A* y 25 columnas de *B*), junto con el *id* que les identifica de forma unitaria, los *Workers* y Gestor de Comunicación al enviar el resultado (*C*) también enviarán el *id* del paquete que lo generó, de esta forma, es posible identificar las tareas (*A* y *B*) que generaron un determinado resultado (*C)* y poder actualizar la tabla de configuración/estado.

En el momento de la ocurrencia del fallo quedarán especificados en la tabla los paquetes que aún no han sido procesados (Estado: Enviado = Si, Acabado = No) tanto sean locales como remotos (Tabla 5-4 de estado de las tareas enviadas hasta el momento y Figura 5-3

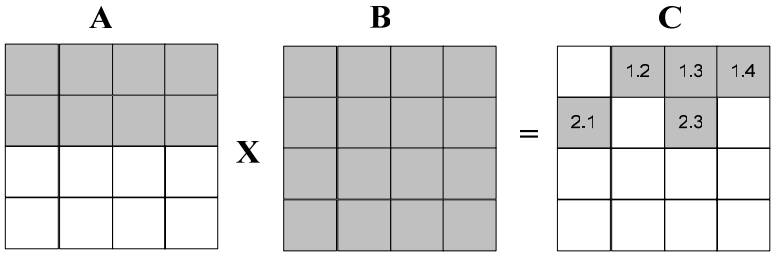

Figura 5-3: Resultado (0)

muestra la resultados que ya han sido calculados y almacenados en el *Master*).

El sistema de tolerancia a fallos está implementado utilizando un protocolo que requiere la confirmación explícita de la llegada de mensajes en la capa del *Middleware*. Siempre que un *Worker* o Gestor de Comunicación envía el resultado, el *Master* devuelve una confirmación (*ack*) de recepción de resultado. Siempre que el *Middleware* MMT replica la tabla de configuración/estado y la tabla de resultados, los Protectores del *Master* devuelven una confirmación (*ack*) de recepción de la tabla.

5.3. Replicación centralizada de datos

En el modelo centralizado de Replicación de Datos, se envían los datos al *Master* y la Replicación de Datos se centraliza en los Protectores del *Master* (PMT), por lo tanto existe una copia completa actualizada de datos en los *Master* y los PMT. Este Tipo de replicación no se utiliza para los nodos de cómputo, porque incrementaría el coste del sistema sin incrementar significativamente los beneficios.

Esta alternativa de replicación centralizada de datos, en caso de fallo de un nodo para la recuperación y re-configuración, requiere poco *overhead*, ya que la fase de recuperación es muy sencilla. En esa alternativa, el *Master* dispone de los datos iniciales que los replica en los PMT y a medida que recibe un resultado copia de todos los resultados y tablas en los PMT, que de esta forma disponen de una copia o replica de todos los datos iniciales, resultados y tablas de configuración/estado, teniendo un espejo en el PMT de los datos iniciales, resultados y tablas del *Master*.

Esta opción sin embargo, tal y como se muestra en la validación experimental, implica un gran *overhead* en ausencia de fallos, ya que requiere un gran uso de la red para la Replicación de Datos.

Para probar el funcionamiento en la práctica de la replicación centralizada de datos y medir el *overhead* en presencia y ausencia de fallo se realizaron las pruebas ejecutando la aplicación sin el *Middleware* FTDR, midiendo el tiempo de la aplicación con **N** nodos *Worker* (T_{AN}) y ejecutando la aplicación con el *Middleware* FTDR (T_{AN}^{sf}), sin inyectar fallos en ninguno de los distintos nodos, de este modo podemos analizar el *overhead* de hacer una copia a través de la red de todos los datos, durante la ejecución.

Para comprobar el tiempo de recuperación/re-configuración y poder compararlo con el sistema descentralizado se ejecuta la aplicación con el *Middleware* **FTDR** en el *cluster* **local**, inyectando **fallos simples** (uno cada vez) en distintos nodos del *cluster*, cuando el

Nodo *Master* (MMT o SMT) está **recibiendo resultados** de los *Workers*.

Se han provocado fallos en nodos con el proceso *Worker*, después de enviar 50%, 75 y 95% de los resultados para distintos tamaños de matrices. También se ha provocado un fallo en el nodo con el proceso *Worker* y Protector del *Master*, lo que requiere una recuperación de toda la Replicación de Datos. Estas pruebas se han realizado con diferentes tamaños de matrices.

A continuación, se muestra una descripción de funcionamiento para distintas situaciones y la validación experimental con replicación centralizada de datos que nos permite comprobar tanto el estado del sistema después de un fallo, y su comportamiento hasta el finalizar la ejecución, garantizando la correcta finalización, como el *overhead* introducido:

- Arranque y ejecución sin fallos: en este caso analizamos el comportamiento y el *overhead* introducido solo por las fase de protección y detección de fallos (ver apartado 5.3.1).
- Fallo simple en un Nodo *Worker* (NodoWK), en el que solo se está ejecutando un proceso *Worker* (WK): en este caso además de las fases de protección, detección, se activarán las fases de diagnóstico, reconfiguración (aislara al nodo que ha fallado) y recuperación de fallos (ver apartado 5.3.3).
- Un único fallo en un Nodo *Worker* (NodoWK) en el que se están ejecutando los procesos *Worker* (WK) y Protector del *Master* (PMT) – con las fases de protección, detección y recuperación de fallos (ver apartado 5.3.5).

5.3.1. Arranque y ejecución sin fallos – solo fase de protección y detección de fallos

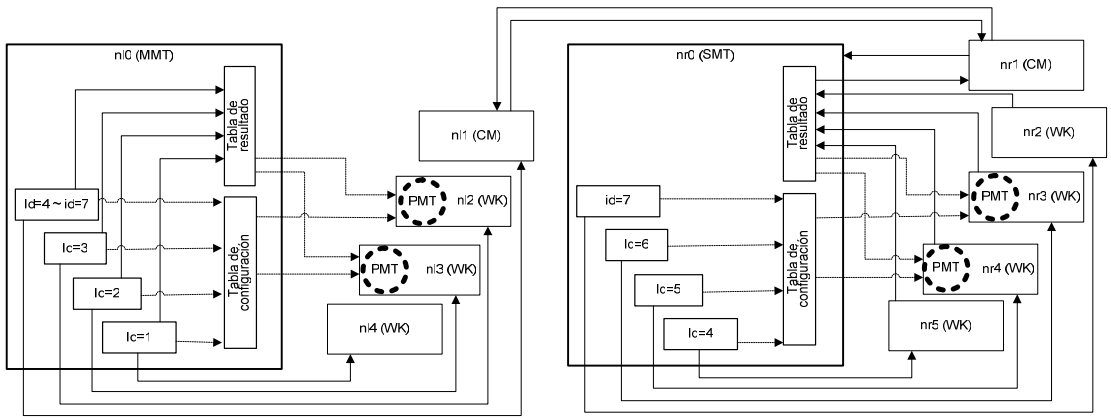

Figura 5-4: Arranque y ejecución sin fallos - replicación centralizada de datos

En este ejemplo ninguno de lo nodos del *cluster* fallan (Figura 5-4).En este caso se ha configurado el *cluster* para que exista protección frente a dos fallos simultáneos, por ello se configuran dos *Workers* con Protector del *Master* en cada uno de los clusters.

Para desarrollar el ejemplo utilizamos matrices divididas en 4x4 bloques, lo que genera un total de 16 tareas, de las cuales 4 se enviarán al subcluster (id 4 a 7), estos datos se enviarán al Gestor de Comunicaciones (CM) en el nodo local 1, al que en lugar de enviarle una tarea le enviamos todo el trabajo que tendrá que realizar el subcluster.

En el momento del arranque una vez configurado el cluster (Figura 5-4): el MMT (nl0) **inicializa** la máquina virtual, y el *Middleware* (nl0) crea una tabla de configuración (Tabla 5-5).

Tabla 5-5: Tabla de configuración/estado - replicación centralizada de datos (1)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**

Para la detección existe un proceso o *thread* en el *Middleware* (nl0 “MIDMASTER”) que se queda verificando si los nodos locales (WK y CM) están “vivos” y los *Middleware* (nl2 y nl3 “MIDWORKER”, que se han configurado como PMT) se quedan verificando si el nodo local (MMT) está “vivo” (**detección**).

- El *Middleware* (nl0 “MIDMASTER”) **puede** detectar fallos simples en un NodoWK (nl4) solo con el proceso WK (ver apartado 5.3.3), un NodoWK (nl3 y nl2) con el proceso WK y PMT (ver apartado 5.3.5) y un nodo Gestor de Comunicaciones CM (nl1).
- Los *Middleware* (nl3 y nl2 “MIDWORKER”) **pueden** detectar fallos en el NodoMT (nl0).

Si consideramos una ejecución que finaliza con todos los nodos vivos. Se irán ejecutando los siguientes pasos durante la ejecución de la aplicación si se utiliza el *Middleware* FTDR (ejecución con *N* nodos sin fallos (T_{AN}^{sf})) (Figura 5-4).

Una vez generadas las tablas y replicados los datos iniciales en el PMT, mientras el MMT (nl0) envía bloques de filas y columnas de la matriz (tareas) a los *Workers* locales (nl4, nl3 y nl2) y al CM (nl1) (Figura 5-5) , el *Middleware* en el nodo *Master* (nl0) escribe en la tabla de configuración (Tabla 5-6) a que nodo envió cada una de las tareas: id=1, id=2,

id=3, id=4, id=5, id=6 y id=7 (fase de **protección**).

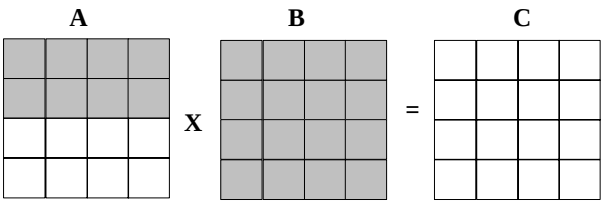

Figura 5-5: Resultado - replicación centralizada de datos (1)

El *Middleware* (nl0), mientras los nodos *Workers* están realizando el computo, replica la tabla de configuración (Tabla 5-6) en los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

Tabla 5-6: Tabla de configuración/estado - replicación centralizada de datos (2)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	No
3	1.3	nl2	Si	No
4	1.4	nl1	Si	No
5	2.1	nl1	Si	No
6	2.2	nl1	Si	No
7	2.3	nl1	Si	No

El MMT (nl0) recibe resultados de los *Workers* locales, supongamos que responden los *Workers* con las tareas id=2, id=3 y del CM se reciben los resultados de las tareas id=5, id=7, se almacenan estos resultados, en la tabla de resultado (matriz resultado "*C*") (Figura 5-6), y a continuación se deben replicar en los PMT.

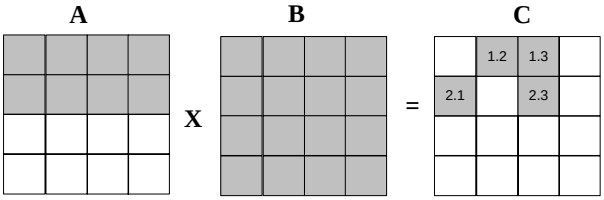

Figura 5-6: Resultado - replicación centralizada de datos (2)

EL *Middleware* (nl0) actualiza la tabla de configuración/estado, (**acabado=Si**) (Tabla 5-7), en la que se refleja de que nodo se ha recibido el resultado (**protección**).

El *Middleware* (nl0) va replicando la tabla de resultado (matriz resultado "*C*") y la tabla de configuración (Tabla 5-7) y (Figura 5-6) en los PMT que están localizados en los nodos nl3 y nl2 (**protección**). Para garantizar la fiabilidad de las replicas es necesario establecer

un protocolo recepción de los resultados.

Tabla 5-7: Tabla de configuración/estado - replicación centralizada de datos (3)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si

El MMT (nl0) envía bloques de filas y columnas de la matriz (tarea) para los *Workers* locales (nl3 y nl2) (Figura 5-7), no es necesario enviarlo al Gestor de Comunicación, ya que se le había enviado todo el trabajo que se había calculado que el subcluster podía realizar.

El *Middleware* (nl0) actualiza la tabla de configuración especificando a que nodo envió

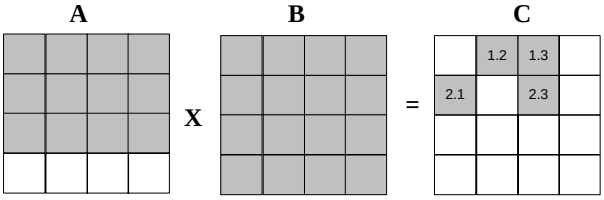

Figura 5-7: Resultado - replicación centralizada de datos (3)

cada una de las tareas: id=8, id=9 (**protección**).

El *Middleware* (nl0) va replicando la tabla de configuración y la tabla de resultado (matriz resultado "*C*") (Figura 5-7) para los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

El MMT (nl0) va recibiendo resultados de los *Workers* locales (id=1, id=8, id=9) y del CM (id=4) y va actualizando la tabla de resultado (matriz resultado "C") (Figura 5-8).

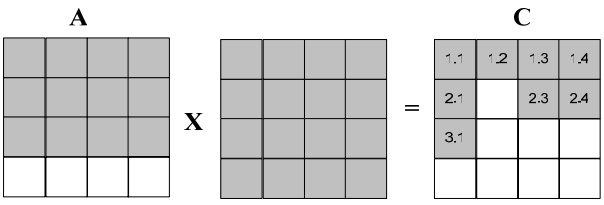

Figura 5-8: Resultado - replicación centralizada de datos (4)

A su vez el *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-8),

especificando de qué nodo ha recibido resultado (id=1, id=8, id=9, id=4) (**protección**), además va replicando la tabla de resultado (matriz resultado "*C*") (Figura 5-8) y la tabla de configuración (Tabla 5-8) en los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

Tabla 5-8: Tabla de configuración/estado - replicación centralizada de datos (4)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si

El MMT (nl0) envía bloques de filas y columnas de la matriz (tarea) para los *Workers* locales (nl4, nl3, nl2) (Figura 5-9)

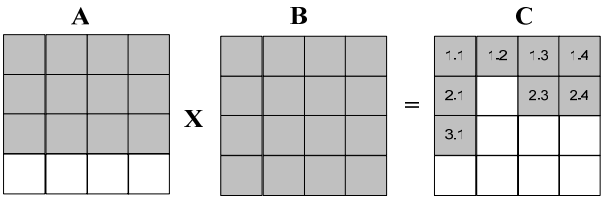

Figura 5-9: Resultado - replicación centralizada de datos (5)

EL *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-9) para donde envió

Tabla 5-9: Tabla de configuración/estado - replicación centralizada de datos (5)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si
10	3.2	nl4	Si	No
11	3.3	nl3	Si	No
12	3.4	nl2	Si	No

cada una de las tareas (id=10, id=11, id=12) (**protección**).

El *Middleware* (nl0) va replicando la tabla de configuración (Tabla 5-9) y la tabla de resultado (matriz resultado "*C*") (Figura 5-9) para los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

El MMT (nl0) recibe resultado de los workers locales (id=10, id=11, id=12) y del CM (id=6) y escribe en la tabla de resultado (matriz resultado "C") (Figura 5-10).

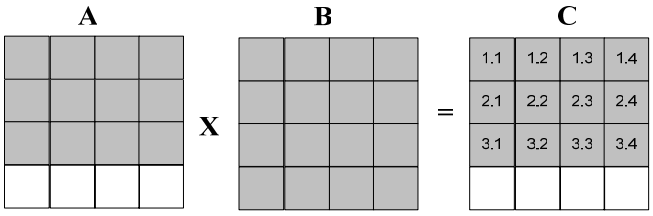

Figura 5-10: Resultado - replicación centralizada de datos (6)

Tabla 5-10: Tabla de configuración/estado - replicación centralizada de datos (6)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	Si
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si
10	3.2	nl4	Si	Si
11	3.3	nl3	Si	Si
12	3.4	nl2	Si	Si

EL *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-10) de quién ha recibido resultado (id=10, id=11, id=12, id=6) (**protección**). El *Middleware* (nl0) va replicando la tabla de configuración (Tabla 5-10) y la tabla de resultado (matriz resultado "*C*") (Figura 5-10 para los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

El *Middleware* (nl0 "MIDMASTER") se queda verificando si los nodos locales (WK y CM) están "vivos" y los *Middleware* (nl3 y nl2 "MIDWORKER") se quedan verificando si el nodo local (MMT) está "vivo" (**detección**).

- **Todos los nodos están vivos**.

El MMT (nl0) envía bloques de filas y columnas de la matriz (tarea) para los *Workers* locales (nl4, nl3, nl2, nl1). (Figura 5-11)

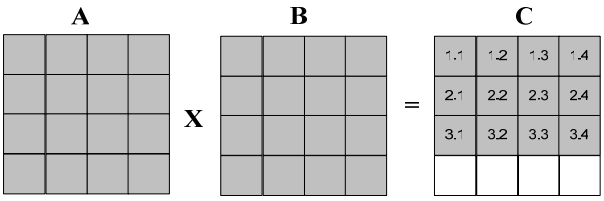

Figura 5-11: Resultado - replicación centralizada de datos (7)

EL *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-11) para donde envió cada una de las tareas (id=13, id=14, id=15, id=16) (**protección**).

El *Middleware* (nl0) va replicando la tabla de configuración (Tabla 5-11) y la tabla de resultado (matriz resultado "*C*") (Figura 5-11) para los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

Tabla 5-11: Tabla de configuración/estado - replicación centralizada de datos (7)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	Si
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si
10	3.2	nl4	Si	Si
11	3.3	nl3	Si	Si
12	3.4	nl2	Si	Si
13	4.1	nl4	Si	No
14	4.2	nl3	Si	No
15	4.3	Nl2	Si	No
16	4.4	nl1	Si	No

El MMT (nl0) recibe resultado de los workers locales (id=14, id=15, id=16) y escribe en la tabla de resultado (matriz resultado "*C*") (Figura 5-12). EL *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-12) de quién ha recibido resultado (id=14, id=15, id=16) (**protección**).

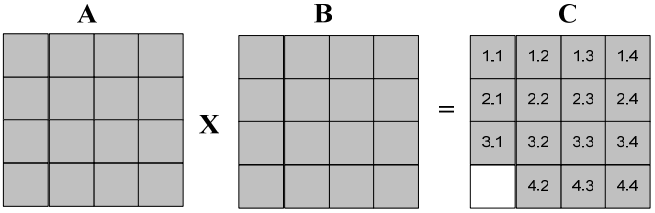

Figura 5-12: Resultado - replicación centralizada de datos (8)

El *Middleware* (nl0) va replicando la tabla de configuración (Tabla 5-12) y la tabla de resultado (matriz resultado "*C*") (Figura 5-12) para los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

Tabla 5-12: Tabla de configuración/estado - replicación centralizada de datos (8)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	Si
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si
10	3.2	nl4	Si	Si
11	3.3	nl3	Si	Si
12	3.4	nl2	Si	Si
13	4.1	nl4	Si	No
14	4.2	nl3	Si	Si
15	4.3	Nl2	Si	Si
16	4.4	nl1	Si	Si

El MMT muestra resultado. El MMT **Finaliza** procesamiento y la máquina virtual. El resultado de id=13, llega más tarde debido a que se está ejecutando en una máquina mas lenta. La Tabla 5-12 nos muestra el estado global de la máquina después de la ejecución.

En los pasos que ejecuta el *middleware* son relativos a las fases de protección y el paso de detección son los que ocasionan *overhead*, debidos a la tolerancia al fallo. Vemos que algunos pasos de la tolerancia a fallos se solapan con el cómputo de los *Workers*.

Una vez hemos visto el funcionamiento vamos a analizar el *overhead* que supone las fases de detección y protección.

5.3.2. Resultados replicación centralizada de datos en el *cluster* local heterogéneo

Para mostrar el *overhead* que introduce este esquema de tolerancia a fallos con una Replicación de Datos centralizada en los PMT se ha medido el tiempo de ejecución de la aplicación, para matrices cuadradas 1000, 2000 y 3000 elementos.

La gráfica nos compara el tiempo de ejecución de la aplicación (A) sin tolerancia a fallos con *N* nodos (T_{AN}) y con el tiempo de ejecución cuando se añade el *Middleware* FTDR, ejecución sin fallo (^{sf}) (T_{AN}^{sf}) en el *cluster* local, en las Figura 5-13, Figura 5-14 y

Figura 5-15 podemos ver estos tiempos para distintos tamaños de las matrices.

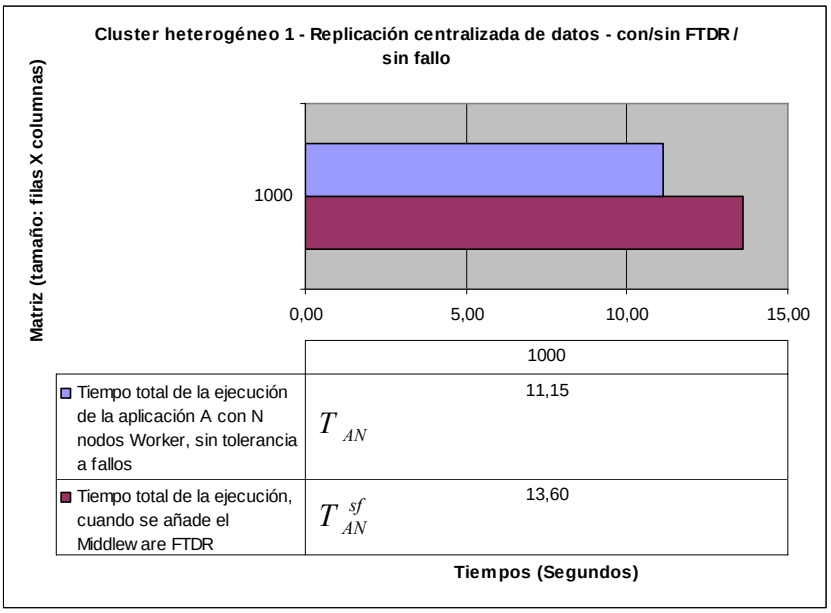

Figura 5-13: *Cluster* local heterogéneo - Replicación centralizada de datos: total de datos procesados (1000 X 1000), sin/con el *Middleware* FTDR, sin fallos

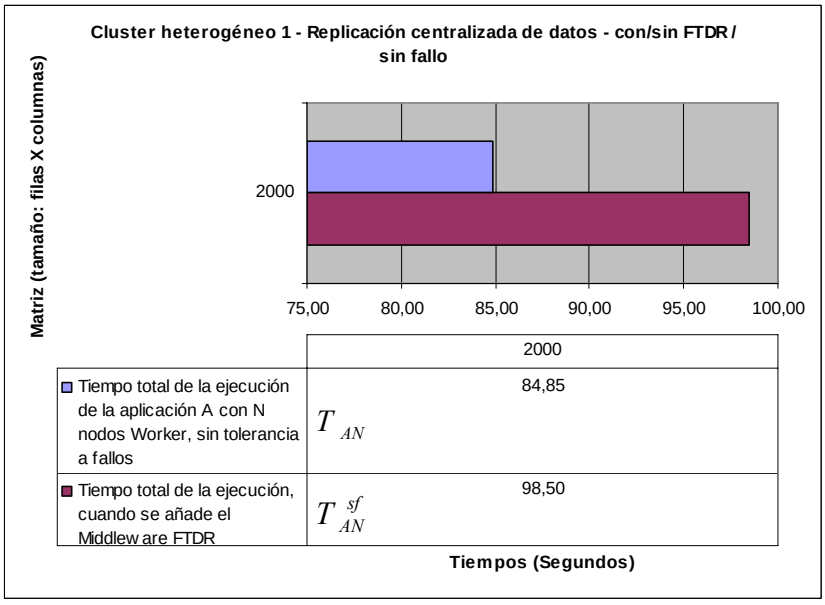

Figura 5-14: *Cluster* local heterogéneo - Replicación centralizada de datos: total de datos procesados (2000 X 2000), sin/con el *Middleware* FTDR, sin fallos

Vemos que realmente el *overhead* introducido por las fases de detección y protección de la tolerancia a fallos es del orden de 2, 45 segundos, 13,65 segundos, 30,81 segundos, pero

relativamente supone un aumento del 21,95% para matrices cuadradas de 1000, de 16,08% para 2000 y de 2,25% para 3000. El tiempo de ejecución aumenta, pero el porcentaje de tiempo gastado en tolerancia a fallos es relativamente menor.

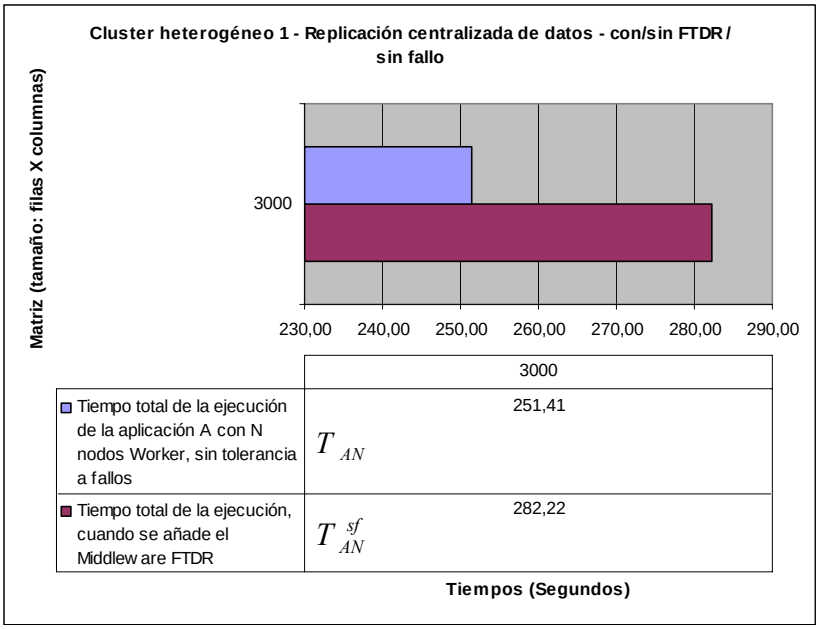

Figura 5-15: *Cluster* local heterogéneo - Replicación centralizada de datos: total de datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, sin fallos

De todos modos consideramos que la penalización sin fallos es importante utilizando un esquema centralizado de replicación de datos (se hace en 2 pasos: se envían los resultados al *Master* y después a 1 PMT), debido al volumen de datos que debe circular por la red.

Ahora vamos a analizar el comportamiento del sistema en caso de fallos.

5.3.3. Fallo simples en un Nodo Worker

En este caso analizaremos las fases de protección, detección y recuperación de fallos. Consideramos un nodo *Worker* que no tiene activado el papel del PMT, y que tenemos dos nodos *Worke*rs para la protección, de forma que podemos soportar fallos hasta de 2 nodos *Master* simultáneamente o de un *Master* y un PMT, el fallo de 2 Workers simultáneamente siempre lo puede soportar el sistema.

En este ejemplo vemos (Figura 5-16), el MMT (nl0) cuando está recibiendo resultados de los WK y del CM, detecta un fallo en un nodo local 4 (**nl4**), que está configurado como *Worker* (WK) y pasa a la fase de recuperación, es decir, marca en la tabla de configuración/estado la tarea enviada al nodo local 4 como tarea no enviada, de este modo

será reasignada a otro nodo *Worker*. En este caso como vemos la recuperación es sencilla, ya que los resultados previos de este nodo ya están en el *Master* y su Replicación de Datos está en el PMT, de forma que sólo se perderá el trabajo que se está computando en dicho nodo en el momento del fallo. A continuación se pasa a la fase de re-configuración en la que se aísla el nodo nl4, con lo cual pasamos a tener un *cluster* con (N-1) nodos *worker* para realizar el cómputo. En este caso, una vez modificada la tabla de configuración/estado en el *Master*, se hace la Replicación de Datos de la tabla en los PMT, aunque no se hayan recibido resultados.

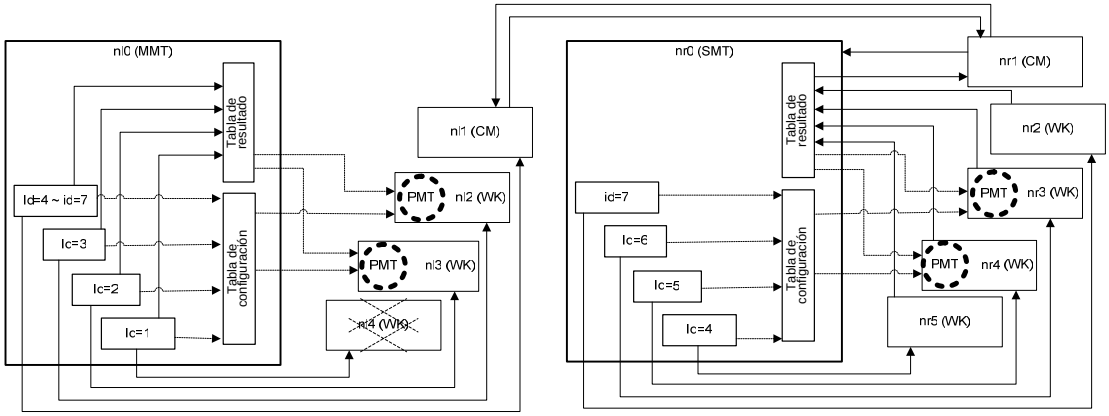

Figura 5-16: Fallo simples en un Nodo *Worker* (nl4) solo con el proceso *Worker* - replicación centralizada de datos

Tal y como hemos visto, existe un proceso en el *Middleware* dedicado a monitorizar a los *Worker* para detectar posibles fallos, (nl0 "MIDMASTER") verifica si los nodos locales (WK y CM) están "vivos", es decir, si envían periódicamente sus *heartbeat,* y los *Middleware* ("MIDWORKER") de los PMT en los nodos nl3 y nl2 se quedan verificando si el nodo local (MMT) está "vivo" (**detección**).

Cuando el *Middleware* (nl0) detecta que el nodo local **nl4** (WK) falló (Figura 5-16),

Tabla 5-13: Tabla de configuración/estado - replicación centralizada de datos (9)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	nl2	Si	No

durante la recepción de resultados (**detección**). Para realizar el diagnóstico el *Middleware* (nl0) verifica en la tabla de configuración (Tabla 5-13) si la tarea del nodo que falló (nl4) está acabada, como no es así, no tiene los resultados, es necesario cambiar sólo el estado.

Si hubieran llegado parte de los resultados, es decir, el fallo se produjo mientras se estaban enviando los resultados, es necesario borrar los resultados recibidos de dicha tarea y poner la tarea en un nuevo estado.

El *Middleware* (nl0) altera el estado del nodo nl4 en la Tabla de Configuración para "No" Enviado y "No" Acabado (Tabla 5-14), indicando máquina con fallo. La re-configuración del cluster se hace teniendo en cuenta que a partir de este momento no se envía mas trabajo porque "**No**" Enviado y "**No**" Acabado = Fallo. Una máquina que no ha acabado, no se puede enviar nuevo trabajo (**re-configuración**).

El *Middleware* (nl0) reasigna la tarea "1.1" (id=1) a otro *Worker*, en la tabla se ha reasignado al nodo nl3 (Tabla 5-14) (**recuperación**).

Tabla 5-14: Tabla de configuración/estado - replicación centralizada de datos (10)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	No	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	Nl2	Si	No
1	1.1	nl3	Si	No

El MMT (nl0) va recibiendo resultados de los *Workers* locales (id=8, id=9) y del CM (id=4) y escribe en la tabla de resultado (matriz resultado "C") (Figura 5-17).

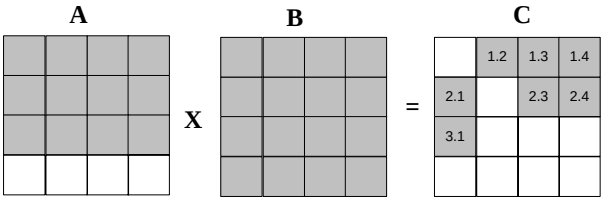

Figura 5-17: Resultado - replicación centralizada de datos (9)

EL *Middleware* (nl0) actualiza la tabla de configuración/estado (Tabla 5-15) para saber de quién ha recibido resultado (id=8, id=9, id=4) (**protección**) y el *Middleware* (nl0) replica la tabla de configuración y la tabla de resultado (matriz resultado ("C") (Figura 5-17) para

los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

Tabla 5-15: Tabla de configuración/estado - replicación centralizada de datos (11)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	No	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	Nl2	Si	Si
1	1.1	nl3	Si	No

Mientras no se detecta fallo, el procesamiento continua hasta el final, es decir, hasta que el MMT muestra resultado y el MMT **Finaliza** procesamiento y la máquina virtual (Tabla 5-16).

Tabla 5-16: Tabla de configuración/estado - replicación centralizada de datos (12)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	No	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	Si
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si
1	1.1	nl3	Si	Si
10	3.2	nl1	Si	Si
11	3.3	nl2	Si	Si
12	3.4	nl3	Si	Si
13	4.1	nl1	Si	Si
14	4.2	nl1	Si	Si
15	4.3	nl2	Si	Si
16	4.4	nl3	Si	Si

5.3.4. Resultados experimentales en el *cluster* local heterogéneo - WK

Una vez analizado el funcionamiento vamos a analizar el *overhead* que introduce la Replicación de Datos. Tal y como hemos visto cuando un nodo falla, tenemos una perdida de prestaciones tanto por el *overhead* introducido por el controlador de tolerancia a fallos como por la perdida de un nodo.

En las Figura 5-18, Figura 5-19 y Figura 5-21 vemos el *overhead* provocado por un fallo al 50% de la ejecución $T_{AN}^{cf}(50\%)$, es decir, durante el 50% del tiempo ejecutamos con N nodos y el otro 50% ejecutamos con N-1 nodos. En este caso estamos utilizando un *cluster* heterogéneo y el nodo que falla es nl4 (101,0874 MFLOPS).

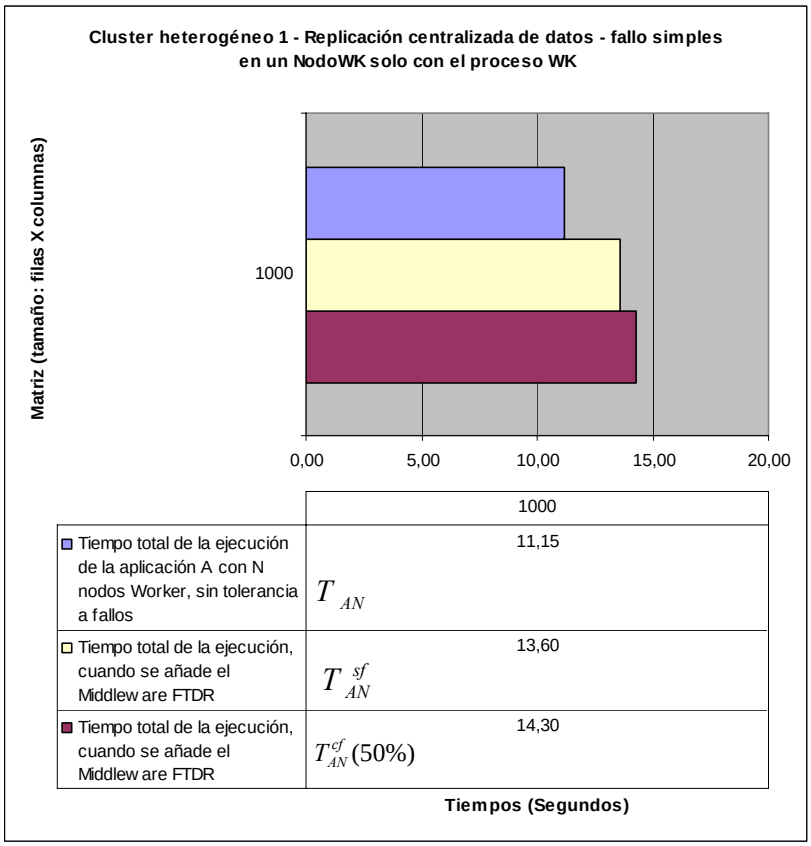

Figura 5-18: *Cluster* local heterogéneo - Replicación centralizada de datos: datos procesados (1000 X 1000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK solo con el proceso WK, después de 50% de datos procesados

El tiempo total de ejecución será:

$T_{AN}^{cf}(50\%) = T_{AN}^{sf}(50\%) + T_{A(N-1)}^{sf}(50\%) + T_{TFRR}$, siendo

$T_{AN}^{cf}(50\%)$ = Tiempo total de ejecución de la aplicación con N nodos y un fallo al 50%.

$T_{AN}^{sf}(50\%)$ = Tiempo de ejecución al 50% de la aplicación con N nodos y sin fallo.

$T_{A(N-1)(50\%)}^{sf}$ = Tiempo de ejecución de la aplicación con N-1 nodos al 50%, sin fallo y con FTDR (fases de prevención y detección).

T_{TFRR} = Tiempo de *overhead* de las fases de la tolerancia a fallos de recuperación y re-configuración.

Para tener una medida aproximada podemos utilizar $T_{AN}^{sf}(50\%) = T_{AN}^{sf}/2$ = Tiempo de ejecución al 50% de la aplicación con N nodos, con FTDR y sin fallo, considerando que existe una fase de inicialización y de finalización que no se está teniendo en cuenta.

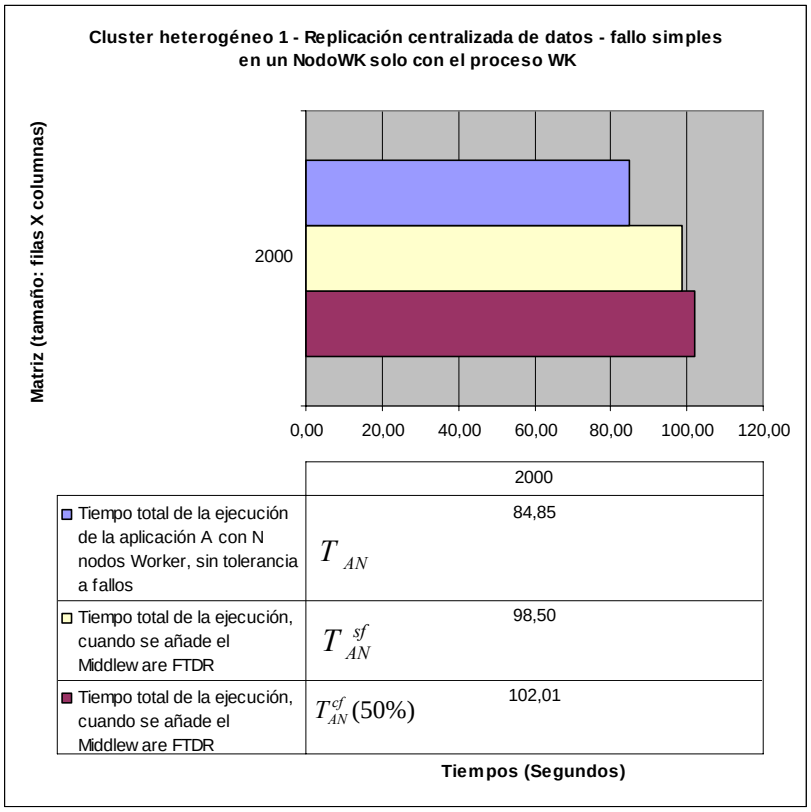

Figura 5-19: *Cluster* local heterogéneo - Replicación centralizada de datos: datos procesados (2000 X 2000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK solo con el proceso WK, después de 50% de datos procesados

Puesto que todos los tiempos de ejecución son conocidos podemos calcular el *overhead* introducido por la tolerancia a fallos en las fases de recuperación y re-configuración con un sistema de Replicación de Datos centralizado (Figura 5-20).

$$T_{TFRR} = T_{AN}^{cf}(50\%) - T_{AN}^{sf}(50\%) - T_{A(N-1)}^{sf}(50\%)$$

Figura 5-20: *Overhead* introducido por la tolerancia a fallos en las fases de recuperación y re-configuración con un sistema de Replicación de Datos centralizado

En la Tabla 5-17 tenemos los tiempos para matrices cuadradas de 1000, 2000 y 3000 elementos.

Tabla 5-17: Tiempos para matrices cuadradas de 1000, 2000 y 3000 elementos (1)

Matriz	T_{AN}	T_{AN}^{sf}	$T_{AN}^{cf}(50\%)$	T_{TFRR}
1000	11,15	13,60	14,30	3,15
2000	84,85	98,50	102,01	17,16
3000	251,41	282,22	289,38	37,96

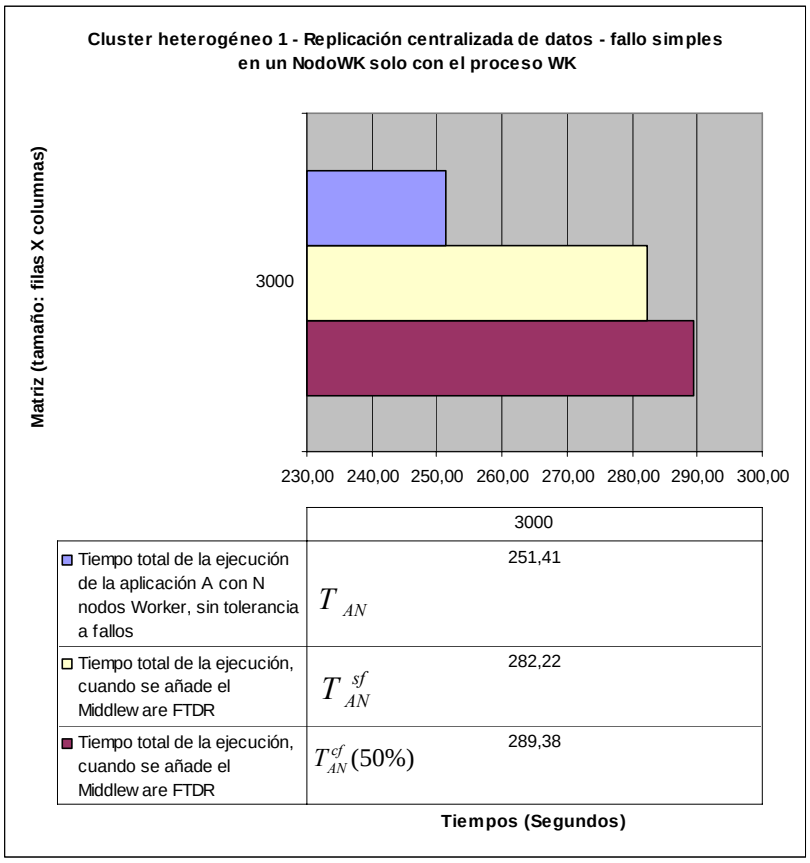

Figura 5-21: *Cluster* local heterogéneo - Replicación centralizada de datos: datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK solo con el proceso WK, después de 50% de datos procesados

5.3.5. Fallo simples en un Nodo Worker con PMT

Vamos a analizar el funcionamiento, cuando existe un fallo y por tanto se activan todas las con las fases de protección, detección y recuperación de fallos, cuando falla un nodo *Worker* que también tiene el proceso PMT. De forma que en la fase de

recuperación debemos tener en cuenta replicar de nuevo los datos del PMT, para poder mantener el mismo nivel de tolerancia a fallos hasta el final de la ejecución, esto hará que el *overhead* sea superior que en el caso de fallo de un *Worker* sin PMT.

En este ejemplo vemos (Figura 5-22), el MMT (nl0) cuando está recibiendo resultados de los WK y del CM, detecta un fallo en un nodo local (**nl3**) (WK/PMT) y pasa a la fase de recuperación: aísla el nodo nl3 y en este caso, envía el trabajo (WK/PMT) del nodo que falló (nl3) para el nodo nl2.

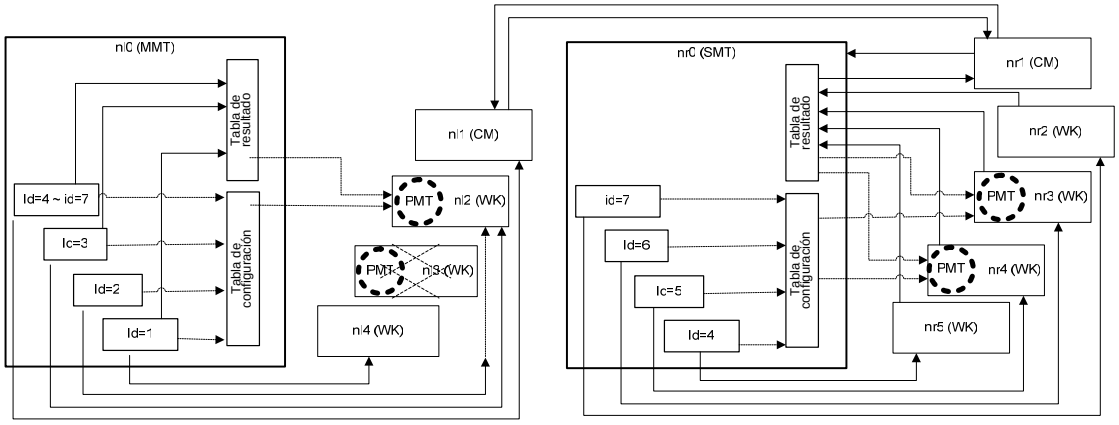

Figura 5-22: Fallo simples en un NodoWK (nl3) con el proceso WK y PMT - replicación centralizada de datos

Tal y como vimos el *Middleware* (nl0 "MIDMASTER") se queda verificando si los nodos locales (WK y CM) están "vivos" y los *Middleware* (nl3 y nl2 "MIDWORKER") se

Tabla 5-18: Tabla de configuración/estado - replicación centralizada de datos (12)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	nl2	Si	No

quedan verificando si el nodo local (MMT) está "vivo" (**detección**). El *Middleware* (nl0) **detecta que el nodo local nl3 falló** (Figura 5-22), durante la recepción de resultados (**detección**). A partir de aquí comienzan las fases de recuperación y re-configuración.

El *Middleware* (nl0) verifica en la tabla de configuración (Tabla 5-18) si la tarea del

nodo que falló (nl3) estaba acabada. En este caso no está acabada, por lo tanto debe ser reasignada.

El *Middleware*, debe reconfigurar el cluster, para ello (nl0) aísla el nodo nl3, o sea, altera el estado del nodo nl3 en la Tabla de Configuración/ estado y lo coloca en el estado "No" Enviado y "No" Acabado (Tabla 5-19) (máquina con fallo). A partir de este momento no se envía mas trabajo porque, tal y como hemos visto, "**No**" Enviado y "**No**" Acabado = Fallo. A una máquina que no ha acabado, no se le puede enviar nuevo trabajo (**re-configuración**).

Tabla 5-19: Tabla de configuración/estado - replicación centralizada de datos (13)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	No	No
9	3.1	Nl2	Si	No
8	2.4	nl1	Si	No

El *Master* (nl0) debe reasignar de nuevo la tarea "2.4" (id=8), en este caso se asigna al *Worker* del nodo nl1 (Tabla 5-19) (**recuperación**). Además debe activar un nuevo PMT en otro *Worker* y realizar una copia de las tablas de estado/configuración y de resultado en dicho *Worker*.

El *Master* debe configurar un nuevo PMT, para ello el *Middleware* del *Master* envía un mensaje al configurador del nodo 4 (CN), para que se activa el PMT, le envía los datos iniciales, la taba de resultados actual y la tabla de configuración/estado, mientras está haciendo esta reconfiguración no atiende a los otros *Workers*, aunque no los ha parado y éstos si que pueden haber terminado su cómputo.

El MMT (nl0) recibe resultado de los *Workers* locales (id=1, id=9) y del Gestor de Comunicación (*Communication Manager*: CM) (id=4) y escribe en la tabla de resultado (matriz resultado "*C*") (Figura 5-23). El *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-20), indicando de quién ha recibido resultado (id=1, id=9, id=4) y hace la copia en el nuevo PMT (**protección**).

El *Middleware* (nl0) a replica la tabla de configuración (Tabla 5-20) y la tabla de resultado (matriz resultado "*C*") (Figura 5-23) para los PMT que están localizados en el nodo **nl2** (**protección)**. Y en el nodo nl4 (nuevo PMT).

Tabla 5-20: Tabla de configuración/estado - replicación centralizada de datos (14)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	No	No
9	3.1	Nl2	Si	Si
8	2.4	nl1	Si	No

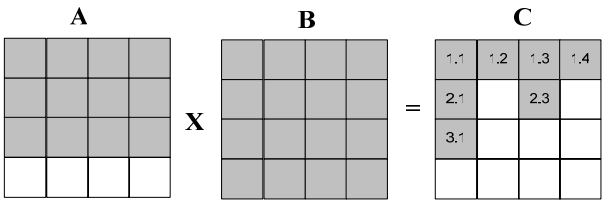

Figura 5-23: Resultado - replicación centralizada de datos (10)

Mientras no se detecta fallo el procesamiento continúa hasta el final, el MMT muestra resultado y **finaliza** procesamiento y la máquina virtual.

5.3.6. Resultados experimentales en el cluster local heterogéneo - WK/PMT

El tiempo de ejecución de la aplicación, matrices 1000, 2000 y 3000, replicación centralizada de datos, sin y con el *Middleware* FTDR, ejecución con fallo en un Nodo *Worker* con los procesos WK/PMT, después de 50% de datos procesados en el *cluster* local, pueden ser vistos en las Figura 5-24, Figura 5-25 y Figura 5-26. En este caso el nodo que falla es nl3.

En este caso para fallo en WK/PMT ($T_{AN}^{cf}(50\%)$), sale mejor tiempo que con el fallo de un WK, eso si debe ser que una máquina muy heterogénea antes se perdió un nodo con muchas prestaciones y ahora se ha perdido un nodo con pocas prestaciones.

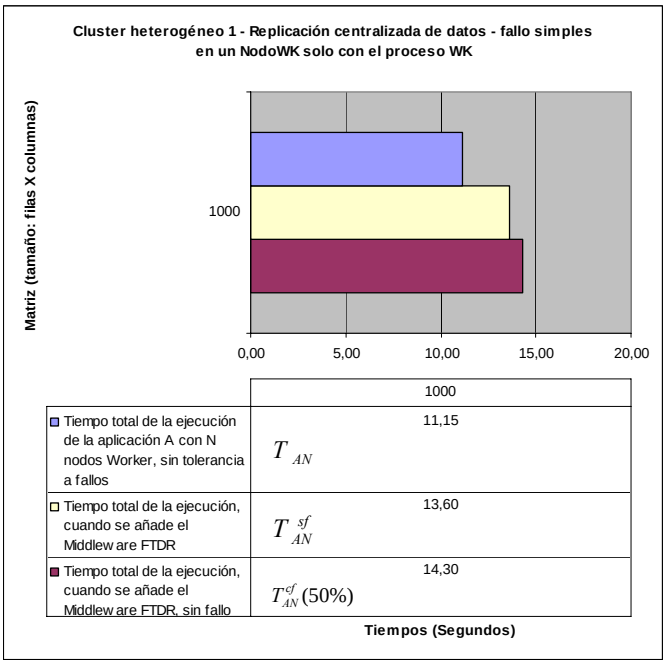

Figura 5-24: *Cluster* local heterogéneo - Replicación centralizada de datos: datos procesados (1000 X 1000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK con el proceso WK/PMT, después de 50% de datos procesados

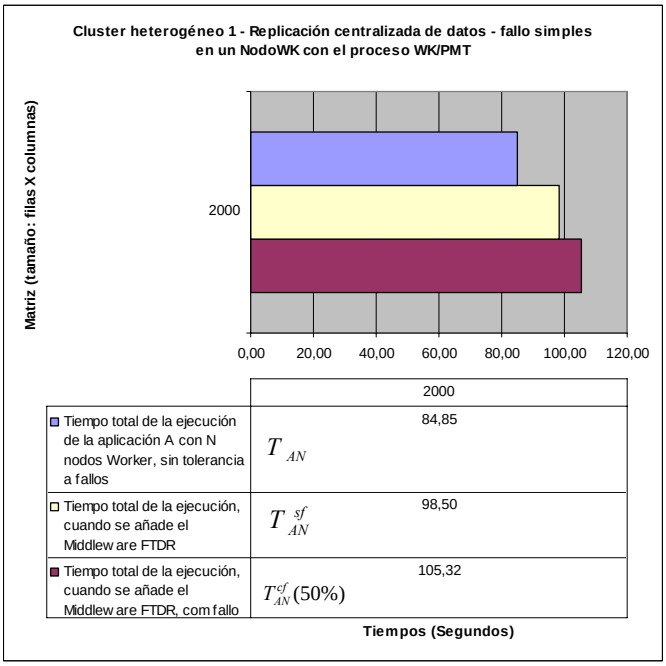

Figura 5-25: *Cluster* local heterogéneo - Replicación centralizada de datos: datos procesados (2000 X 2000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK con el proceso WK/PMT, después de 50% de datos procesados

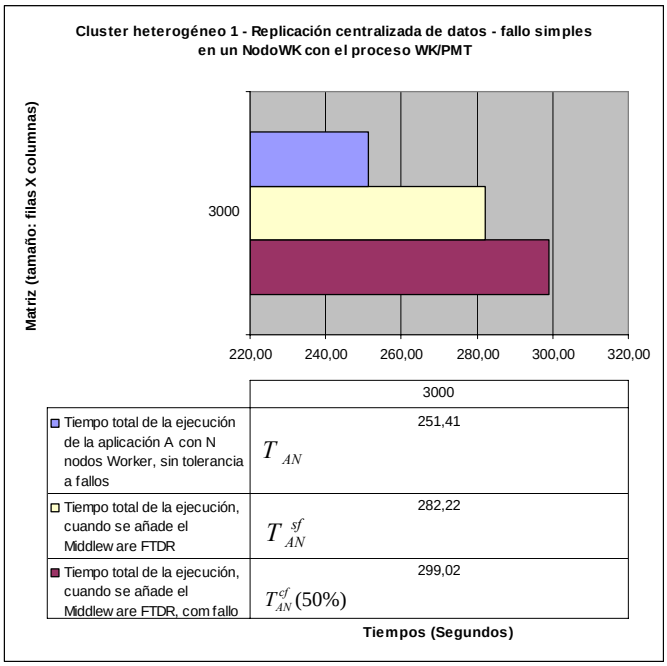

Figura 5-26: *Cluster* local heterogéneo - Replicación centralizada de datos: datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK solo con el proceso WK/PMT, después de 50% de datos procesados

En la Tabla 5-21 tenemos los tiempos para matrices cuadradas de 1000, 2000 y 3000 elementos.

Tabla 5-21: Tiempos para matrices cuadradas de 1000, 2000 y 3000 elementos (2)

Matriz	T_{AN}	T_{AN}^{sf}	$T_{AN}^{cf}(50\%)$	T_{TFRR}
1000	11,15	13,60	14,15	3,00
2000	84,85	98,50	15,32	20,47
3000	251,41	282,22	299,02	47,61

5.3.7. Consideraciones sobre los resultados experimentales en el *cluster* local heterogéneo

En estos experimentos mostrados utilizando replicación centralizada de datos utilizamos un *cluster* muy heterogéneo, por lo que además de la política de tolerancia a fallos, influyen mucho, el *mapping* realizado y las características del nodo que falla, siendo difícil aislar en los resultados la influencia de los diferentes factores. Otro problema asociado es que debido a fallos reales en los nodos, las características de los nodos del *cluster* también cambiaban

Basado en los resultados obtenidos con la replicación centralizada de datos, llegamos a

la conclusión que el *overhead* inyectado en el sistema sin fallos con esa forma de replicación era muy alto, por lo tanto, decidimos buscar una manera más eficiente de hacer esa replicación.

En la replicación distribuido de datos hemos realizado pruebas configurando un *cluster* local muy heterogéneo y otro más homogéneo, es decir que el grado de heterogeneidad no era tan elevado, esto es debido a que cuando inyectamos un fallo, no sólo depende si falla un nodo con un proceso *Worker*, sino que también influye que nodo *Worker* falla, lo cual complica la predicción de resultados a priori, ya que necesitaríamos tener el *cluster* caracterizado con N *Workers* y con N-1 *Workers* (teniendo en cuenta quitar uno a uno los diferentes *Workers).*

5.4. Replicación distribuida de datos

Tal y como vimos en el capítulo anterior, basados en que los *Workers* tienen los datos y pueden almacenarlos para tener una réplica de los resultados, esto reduce el tiempo de *overhead* de la red, de forma que la única información que necesitamos que circule por la red es la réplica de la tabla de estado/configuración. Decidimos optar por la replicación distribuida de datos, pues es mejor sin fallos, a pesar de que la re-configuración sea más lenta.

Ahora es importante conocer, además del estado de los procesos (enviados y acabados) y nodos (operativo o fallo), la historia de ejecución, para saber en que nodo está la réplica de los resultados procesados.

En esa alternativa, se envían y almacenan los resultados en el *Master* y la copia se almacena en los propios *Workers* que han realizado el cómputo. Las tablas de estado/configuración se replican en los PMT. En caso de fallo de un *Worker*, estos resultados se envían desde al *Master* a otro *Worker*, el *Worker* seleccionado es el que tiene el PMT, es decir, en caso de fallo de un nodo, los datos procesados por el *Worker* deben enviarse al PMT o a otro *Worker*, y se modifican las tablas indicando que *Worker* lo ha almacenado. El Gestor de Comunicación (CM) funciona de un modo similar almacenando la copia de los resultados del Subcluster.

En la Tabla 5-22 se muestra una visión global de cómo se distribuye la Replicación de Datos entre los nodos del *cluster*.

Debido a la distribución dinámica de datos en el *cluster* local, los datos iniciales son replicados en el PMT. En el caso de utilizar una distribución estática inicial, también se

podría conservar sólo la copia en los *Workers* y no hacer la replicación de los datos iniciales que se realiza en el arranque del sistema.

Tabla 5-22: Nodos que interviene en la replicación distribuida de datos

Tipo de información	Datos	Replicación de datos
Datos iniciales	*Master*	PMT
Resultados	*Master*	*Worker*
Tabla estado/configuración	*Master*	PMT
Resultado del Subcluster	*Master*	Gestor de Comunicación

Los *Workers* y Gestores de Comunicación, tienen capacidad de almacenamiento para tener la Replicación de Datos. Cuando un *Worker* hace un *send*, el MIDDLEWORKER debe almacenar también los resultados, así que en este caso se añaden tareas de la fase de protección en los *Worker*.

En caso de fallo de un nodo, los PMT tiene los datos iniciales y la tabla de estado/configuración con información para la re-configuración, que es necesaria hacer dependiendo del nodo que ha fallado.

Al igual que en el caso anterior, el PMT es el encargado de detectar si el *Master* falla, recibe los *hearbeat* del *Master* y si pasa un tiempo de *watchdog* sin que se reciba, se pasa a la fase de diagnóstico de fallo, para analizar o descartar un posible fallo del *Master*. En caso de fallo, asume las funciones del *Master*. Para recuperar la tabla de resultados, busca los resultados en los *Workers*, a partir de la tabla de configuración.

Para probar el funcionamiento en la práctica de la replicación distribuida de datos y el *overhead* producido por la Replicación de Datos se realizaron las diferentes pruebas.

Para medir el *overhead* de este modelo de tolerancia fallos hemos ejecutado la aplicación sin el *Middleware* FTDR (T_{AN}) y la aplicación con el *Middleware* FTDR, sin inyectar fallos en ninguno de los nodos (T_{AN}^{sf}), este *overhead* debe mejorar respecto a la replicación centralizada de datos, puesto que se evita el *overhead* de la transferencia de datos duplicada a través de la red.

Ejecutar la aplicación con el *Middleware* **FTDR** en el *cluster* **local** y **remoto**, inyectando **fallos simples** (un cada vez) en distintos nodos del *cluster*, cuando el Nodo *Master* (MMT o SMT) está **recibiendo resultados** de los *Workers*, después de enviar 50%, 75 y 95% ($T_{AN}^{cf}(50\%)$, $T_{AN}^{cf}(75\%)$, $T_{AN}^{cf}(95\%)$) de los resultados para distintos tamaños de matrices.

También se han realizado pruebas provocando los fallos en el ***cluster*** **remoto**, para

analizar la influencia que tiene en el *overhead* total del sistema, teniendo en cuenta que, en este caso, debido a la influencia de la red WAN de comunicación que no es predictivo, es complejo analizar la influencia de cada uno de los parámetros en las variaciones de los tiempos de ejecución.

A continuación, se muestra la simulación de funcionamiento para distintas situaciones de funcionamiento con replicación distribuida de datos. Igual que en el caso anterior, una vez presentado el modelo funcional, en el que se especifica la evolución del estado del sistema, comenzamos midiendo el *overhead* que introduce FTDR en un sistema sin fallos. Para ello se ejecuta la aplicación sin fallos, en la que influyen solo las fases de protección utilizando replicación distribuida de datos y detección de fallos (ver apartado 5.4.1).

Para medir el *overhead* introducido cuando existe un fallo, se inyectan fallos, se comienza realizando una inyección de fallos simples en un Nodo *Worker* solo con el proceso *Worker*, en este momento del controlador de tolerancia a fallos, además de las fases de protección, detección, se puede analizar el *overhead* introducido en la fase de recuperación de fallos, ya vimos que esta no es una medida directa (ver apartado 5.4.4). En este caso, en la fase de recuperación hay que tener en cuenta, además de reconfigurar la máquina, proteger a los datos calculados y de los que se ha perdido su Replicación de Datos por estar en el nodo que ha fallado, de este modo garantizamos el mismo nivel de protección hasta el final de la aplicación.

Si el nodo que falla es un Nodo *Worke*r que además actúa como PMT, además de todo el proceso anterior, es necesario activar un nuevo PMT, en este caso basta con replicar los datos iniciales y la tabla de estado/configuración.

Si inyectamos un fallo simple en el Nodo *Master*, es necesario activar la fase de recuperación de datos. El PMT coge el papel de *Master*, a partir de la tabla de estado/configuración recupera todos los resultados calculados por los *Workers*, desactiva el proceso *Worker* que estaba activo en su nodo, por lo que necesita también hacer el trabajo de recuperación de un *Worker*, y por último, antes de reanudar la ejecución debe activar un nuevo PMT en otro *Worker*. Vemos que se unen los *overhead* de todos los fallos anteriores, porque en la reconfiguración del *cluster*, se desactiva un *Worker* y es necesario generar un nuevo PMT.

5.4.1. Esquema de funcionamiento de una ejecución sin fallos

En este caso sólo se activarán las fases de protección y detección de fallos, en el

ejemplo (Figura 5-27), ninguno de lo nodos del *cluster* fallan.

Los siguientes pasos son ejecutados en el momento del arranque y ejecución de la aplicación (sin fallos) (Figura 5-27).

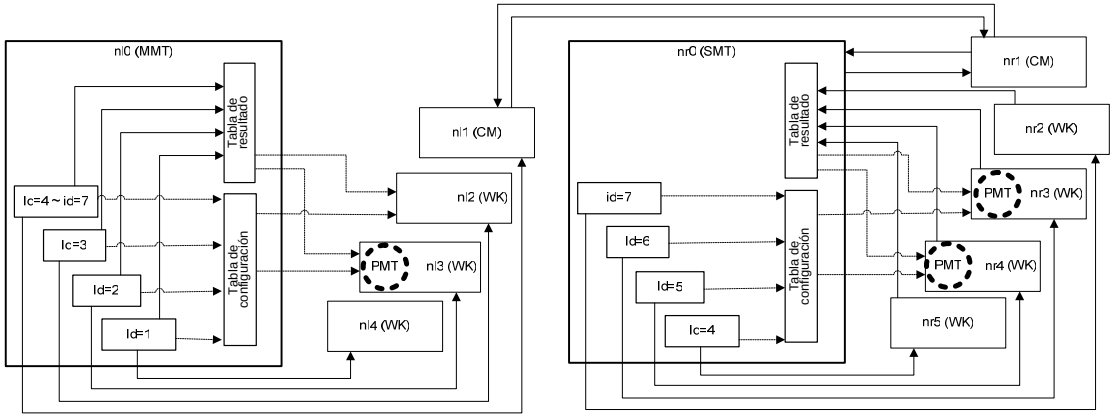

Figura 5-27: Arranque y ejecución sin fallos - replicación distribuida de datos

El MMT (nl0) **inicializa** la máquina virtual. Configura el sistema de tolerancia a fallos. EL *Middleware* del *Master* (nl0) crea una tabla de configuración/estado, vemos que para éste tipo de protección es básico saber donde se ha ejecutado una tarea (columna nodo de la parte de configuración), ya que este es el nodo *Worker* o Gestor de Comunicación (en caso de ejecutarse en un Subcluster), que tiene, la replicación de resultados (Tabla 5-23) (**protección**). Esta columna de configuración (tarea/nodo), no era necesaria en la replicación centralizada, aunque se mantiene para que las estructuras de datos no cambien.

Tabla 5-23: Tabla de configuración/estado - replicación distribuida de datos (1)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**

El *Middleware* (nl0 “MIDMASTER”) se queda verificando si los nodos locales (WK y CM) están “vivos” y el *Middleware* del PMT (nl3 “MIDWORKER”) se quedan verificando

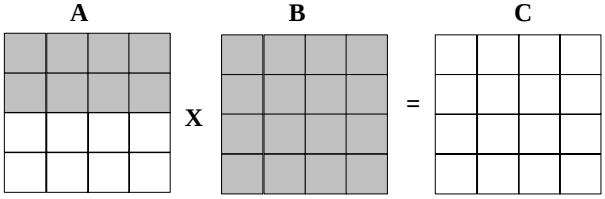

Figura 5-28: Resultado - replicación distribuida de datos (1)

si el nodo local (MMT) está "vivo" (detección). Los *Middleware* (nl3 "MIDWORKER") pueden detectar fallos simples en el Nodo *Master* (nl0). En este primer caso consideramos que todos los nodos están vivos.

El MMT (nl0) envía tareas (bloques de filas y columnas de la matriz) a los *Workers* locales (nl4, nl3 y nl2) y al CM (nl1) (Figura 5-28). Es importante que eL *Middleware* (nl0) actualice en la tabla de configuración (Tabla 5-24) donde envió cada una de las tareas: id=1, id=2, id=3, id=4, id=5, id=6 y id=7 (**protección**), para identificar los nodos en los que se

Tabla 5-24: Tabla de configuración/estado - replicación distribuida de datos (2)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	No
3	1.3	nl2	Si	No
4	1.4	nl1	Si	No
5	2.1	nl1	Si	No
6	2.2	nl1	Si	No
7	2.3	nl1	Si	No

replica cada bloque de resultados.

El MMT (nl0) recibe resultados de los workers locales con las tareas id=2, id=3 y del CM id=5, id=7 y actualiza en la tabla de resultado (matriz resultado "C") (Figura 5-29).

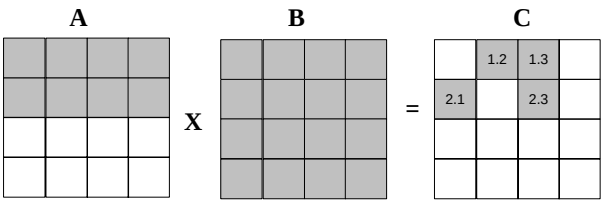

Figura 5-29: Resultado - replicación distribuida de datos (2)

EL *Middleware* en el nodo master (nl0) actualiza la tabla de configuración (Tabla 5-25) especificando de quién ha recibido resultados, cambiando su estado (**acabado=Si**)

Tabla 5-25: Tabla de configuración/estado - replicación distribuida de datos (3)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si

(**protección**).

En el caso de los *Workers* locales, como se va realizando una distribución dinámica de los datos el *Master* (nl0) envía nuevos bloques de filas y columnas de la matriz (tarea) a los *Workers* locales (nl3 y nl2). (Figura 5-31) y el *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-26) especificando donde envió cada una de las tareas: id=8, id=9 (**protección**).

Tabla 5-26: Tabla de configuración/estado - replicación distribuida de datos (4)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	nl2	Si	No

El MMT (nl0) recibe resultado de los *Workers* locales (id=1, id=8, id=9) y del CM (id=4) y escribe en la tabla de resultado (matriz resultado "C") (Figura 5-30) y el *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-27) y replica en el PMT.

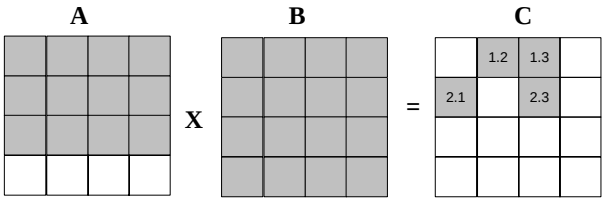

Figura 5-31: Resultado - replicación distribuida de datos (3)

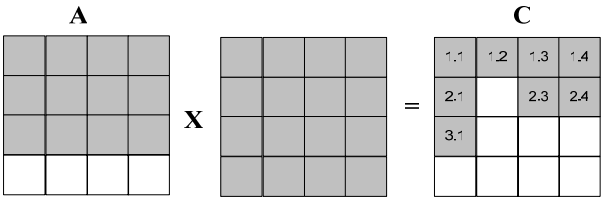

Figura 5-30: Resultado - replicación distribuida de datos (4)

El *Middleware* (nl0 "MIDMASTER") va verificando si los nodos locales (WK y CM) están "vivos" y el *Middleware* PMT (nl3 "MIDWORKER") se quedan verificando si el nodo local (MMT) está "vivo" (**detección**).

EL *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-28) especificando

donde envió cada una de las tareas (id=10, id=11, id=12) (**protección**), así va operando hasta terminar de ejecutar todas las tareas.

Tabla 5-27: Tabla de configuración/estado - replicación distribuida de datos (5)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si

Tabla 5-28: Tabla de configuración/estado - replicación distribuida de datos (6)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si
10	3.2	nl4	Si	No
11	3.3	nl3	Si	No
12	3.4	nl2	Si	No

Mientras no se detecta fallo. El procesamiento continúa hasta el final. El *Master* Principal (MMT) muestra resultado. El MMT Finaliza procesamiento y la máquina virtual.

5.4.2. Resultados replicación distribuida de datos – *cluster* local heterogéneo

Los nodos que configuran el cluster más heterogéneo son los mostrados en la Tabla 5-2. Los nodos utilizados para configurar un cluster con menor grado de heterogeneidad son los mostrados en la Tabla 5-1.

El tiempo de ejecución de la aplicación, matrices 1000, 2000 y 3000, replicación distribuida de datos, sin tolerancia a fallos (T_{AN}) y con el *Middleware* FTDR, ejecución sin fallo (T_{AN}^{sf}) en el *cluster* local heterogéneo, se muestran en las Figura 5-32, Figura 5-33 y Figura 5-34.

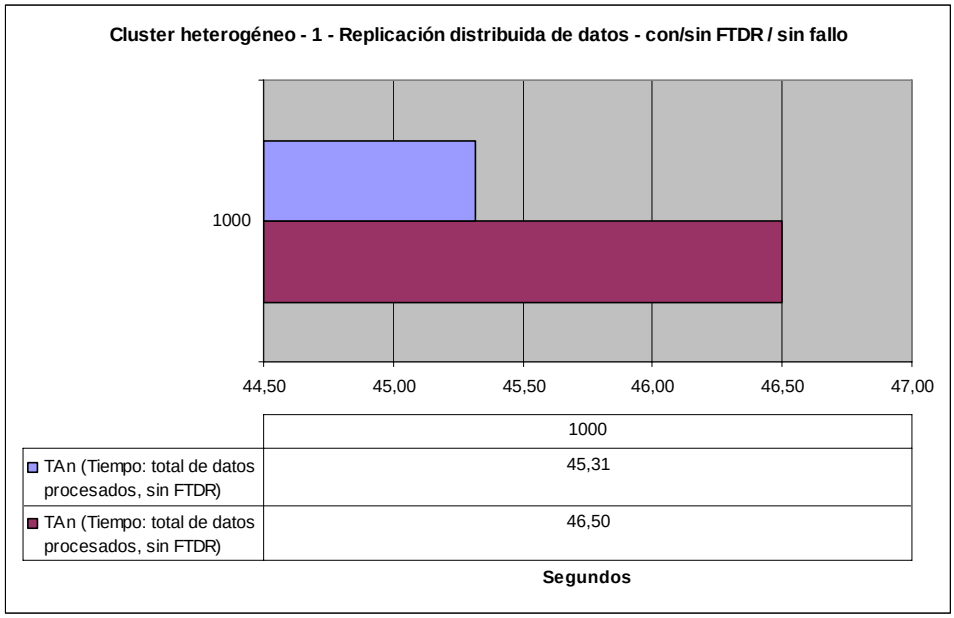

Figura 5-32: *Cluster* local heterogéneo - Replicación distribuida de datos: total de datos procesados (1000 X 1000), sin/con el *Middleware* FTDR

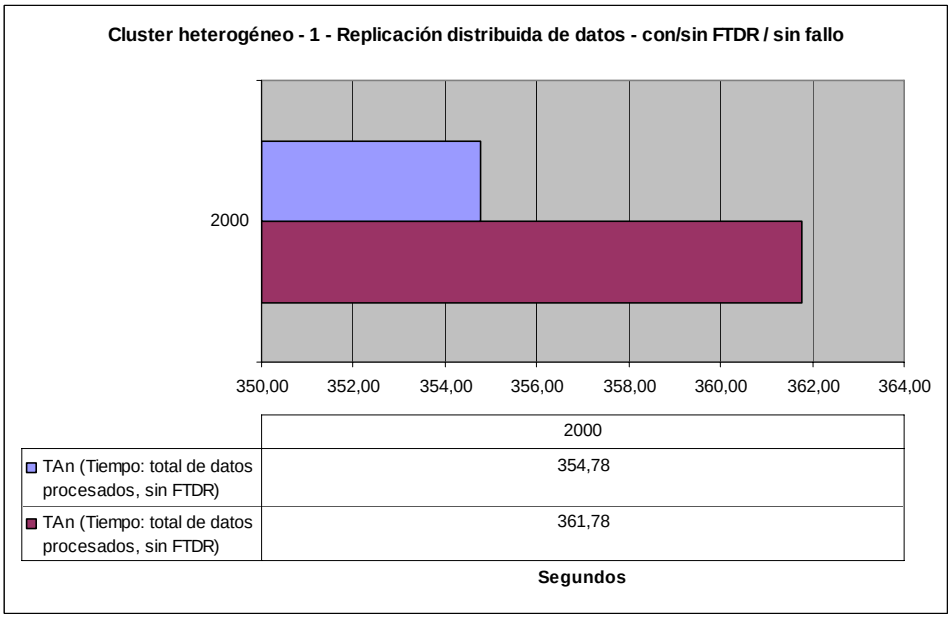

Figura 5-33: *Cluster* local heterogéneo - Replicación distribuida de datos: total de datos procesados (2000 X 2000), sin/con el *Middleware* FTDR

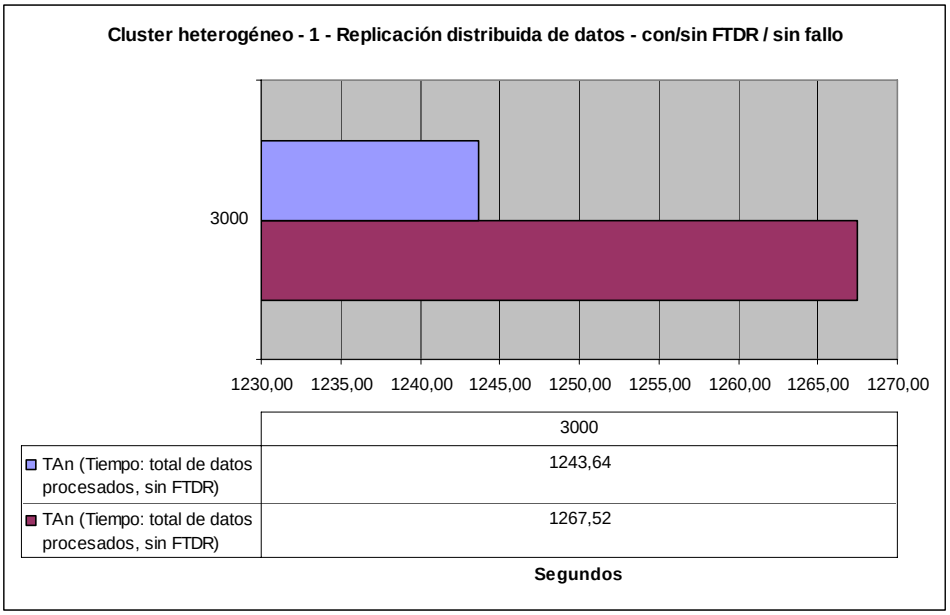

Figura 5-34: *Cluster* local heterogéneo - Replicación distribuida de datos: total de datos procesados (3000 X 3000), sin/con el *Middleware* FTDR

5.4.3. Resultados replicación distribuida de datos – *cluster* local homogéneo

El tiempo de ejecución de la aplicación, matrices 1000, 2000 y 3000, replicación distribuida de datos, sin (T_{AN}) y con el *Middleware* FTDR, ejecución sin fallo (T_{AN}^{sf}) en el *cluster* local homogéneo, cuya configuración podemos ver en la Tabla 5-1, como vemos este *cluster* tiene menos potencia de computo que el heterogéneo (antes para 1000x1000) se tardaban 45 segundos y ahora se tardan 49 segundos), pueden ser vistos en las Figura 5-35, Figura 5-36 y Figura 5-37.

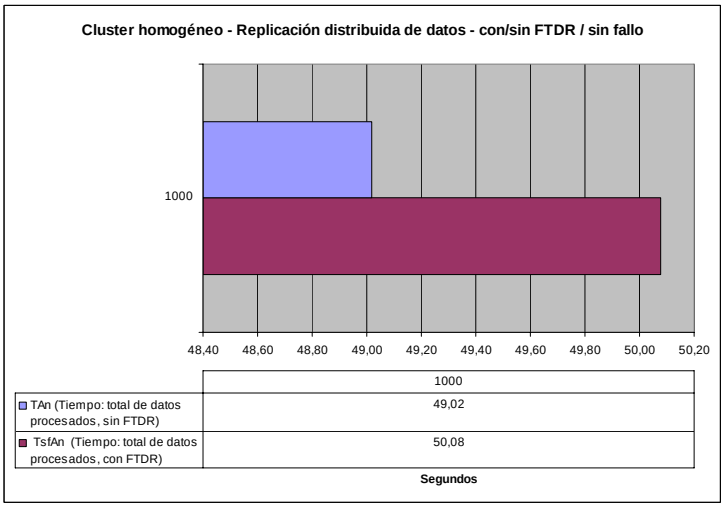

Figura 5-35: *Cluster* local homogéneo - Replicación distribuida de datos: total de datos procesados (1000 X 1000), sin/con el *Middleware* FTDR, sin fallos

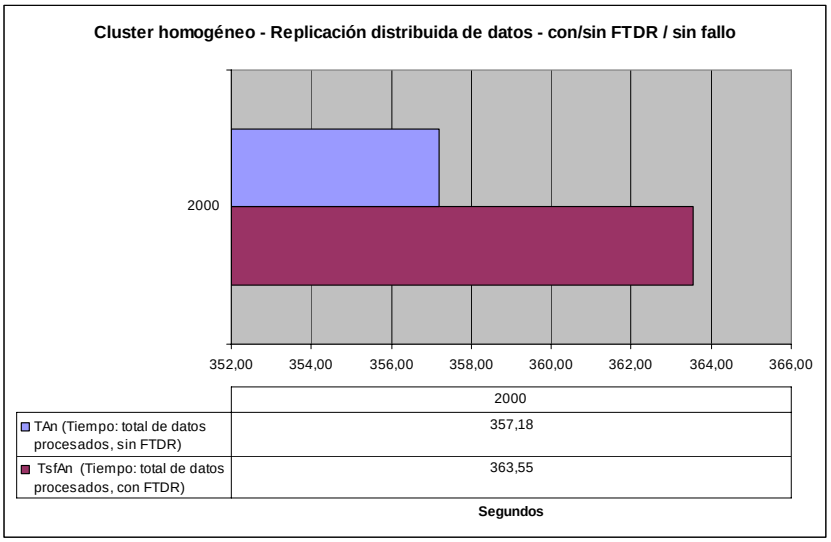

Figura 5-36: *Cluster* local homogéneo - Replicación distribuida de datos: total de datos procesados (2000 X 2000), sin/con el *Middleware* FTDR, sin fallos

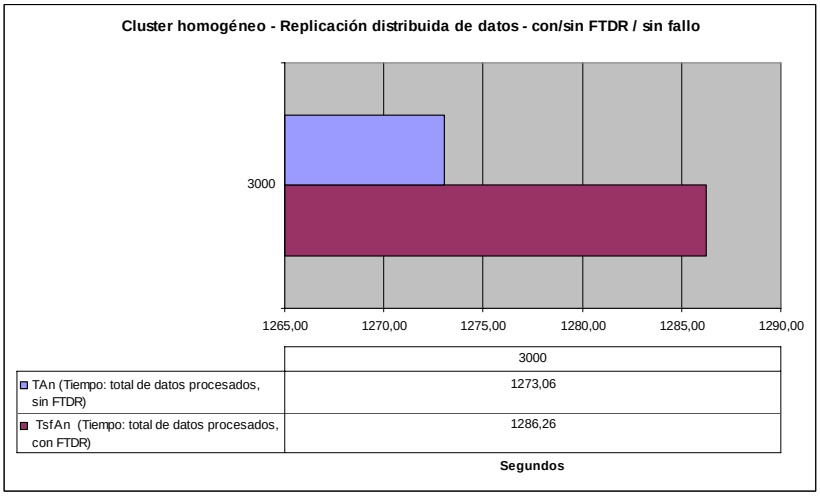

Figura 5-37: *Cluster* local homogéneo - Replicación distribuida de datos: total de datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, sin fallos

5.4.4. Fallo simples en un Nodo Worker

En este caso consideramos que estamos haciendo una Replicación de Datos distribuida y falla un nodo con solo con el proceso WK por lo tanto debemos tener en cuenta las fases de protección, detección, recuperación de fallos y reconfiguración del *cluster*. Pondremos especial interés en la recuperación de los datos replicados en otro *Worker*.

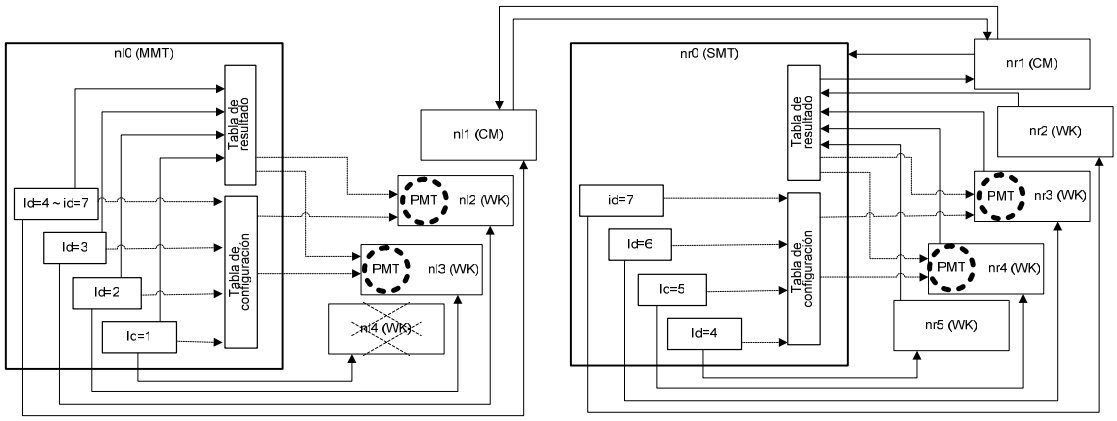

Figura 5-38: Fallo simples en un NodoWK (nl4) solo con el proceso WK - replicación distribuida de datos

El *Middleware* (nl0 "MIDMASTER") se queda verificando si los nodos locales (WK y CM) están "vivos" y los *Middleware* (nl3 y nl2 "MIDWORKER") se quedan verificando si el nodo local (MMT) está "vivo" (**detección**)

En este ejemplo vemos (Figura 5-38), el *Master* principal (MMT) (nl0) cuando está recibiendo resultados de los WK y del CM, detecta un fallo en un nodo local (**nl4**) (WK) y pasa a la fase de recuperación, o sea, aísla el nodo nl4, en este caso antes de acabar la ejecución deben enviar la tarea 1.1 del nodo que falló (nl4) a otro nodo.

El *Middleware* (nl0 "MIDMASTER") se queda verificando si los nodos locales (WK y CM) están "vivos" y los *Middleware* (nl3 y nl2 "MIDWORKER") se quedan verificando si

Tabla 5-29: Tabla de configuración/estado - replicación distribuida de datos (10)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	nl2	Si	No
			Si	

el nodo local (MMT) está "vivo" (**detección**). El *Middleware* (nl0) detecta que el nodo local **nl4** (WK) falló (Figura 5-38), durante la recepción de resultados (**detección**).

El *Middleware* (nl0) verifica en la tabla de configuración (Tabla 5-29) si la tarea del nodo que falló (nl4) fue acabada, como en este caso no ha sido acabada, deberá ejecutarla en otro nodo. El *Middleware* (nl0) aísla el nodo nl4, o sea, altera el estado del nodo nl4 en

la Tabla de Configuración para “No” Enviado y “No” Acabado (Tabla 5-30) (máquina con fallo). A partir de este momento no se envía mas trabajo porque “**No**” Enviado y “**No**”

Tabla 5-30: Tabla de configuración/estado - replicación distribuida de datos (11)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	No	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	Nl2	Si	No
1	1.1	nl3	Si	No

Acabado = Fallo. A una máquina que no ha acabado, no se le puede enviar nuevo trabajo (**re-configuración**).

El *Middleware* (nl0) reasignará la tarea “1.1” (id=1) al WK del nodo nl3 (Tabla 5-30) (**recuperación**). Y debe analizar si el nodo que ha fallado había acabado alguna tarea para replicar sus resultados en el nodo worker que tiene activo el PMT.

Tabla 5-31: Tabla de configuración/estado - replicación distribuida de datos (12)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	No	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	Nl2	Si	Si
1	1.1	nl3	Si	No

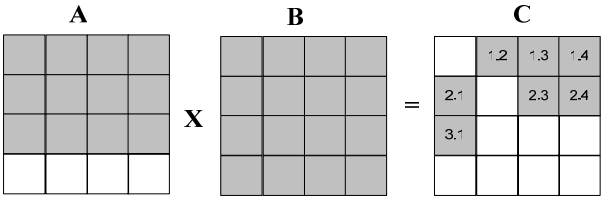

Figura 5-39: Resultado - replicación distribuida de datos (9)

El *Middleware* (nl0) replica la tabla de configuración (Tabla 5-31) y la tabla de resultado (matriz resultado ("C") (Figura 5-39) para los PMT que están localizados en los nodos nl3 y nl2 (**protección**).

Mientras no se detecta fallo el procesamiento continúa hasta el final, el MMT muestra resultado. El MMT **Finaliza** procesamiento y la máquina virtual.

5.4.5. Resultados replicación distribuida de datos – *cluster* local heterogéneo

Los tiempos de ejecución de la aplicación con matrices cuadradas de 1000, 2000 y 3000 elemento y replicación distribuida de datos, ejecutando en un cluster poco heterogéneo, sin tolerancia a fallos (T_{AN}) se muestra en la figura junto con el tiempo de ejecución cuando se incorpora el *Middleware* FTDR, ejecución con fallo en un NodoWK (T_{AN}^{sf}), también se muestra el tiempo de ejecución con fallo en un nodo solo con el proceso WK del *cluster* local heterogéneo, después de 50% de datos procesados ((T_{AN}^{cf}) (Tiempo: 50% de datos procesados)) en el *cluster* local, pueden ser vistos en las Figura 5-40, Figura 5-41, Figura 5-42.

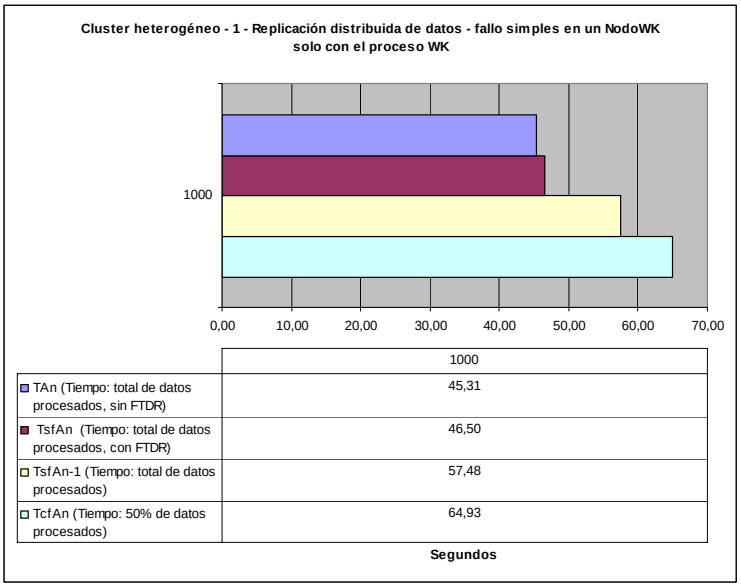

Figura 5-40: *Cluster* local heterogéneo - Replicación distribuida de datos: datos procesados (1000 X 1000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK solo con el proceso WK, después de 50% de datos procesados

Vemos que el *overhead* de incorporar FTDR es pequeño, por otro lado podemos observar que es mucho peor comenzar de nuevo la ejecución con un nodo de cómputo

menos que continuar la ejecución con un nodo menos si hemos incorporado el esquema de tolerancia a fallos, además relanzar la aplicación requiere la intervención del administrador y continuar la aplicación con FTDR es un proceso transparente al usuario.

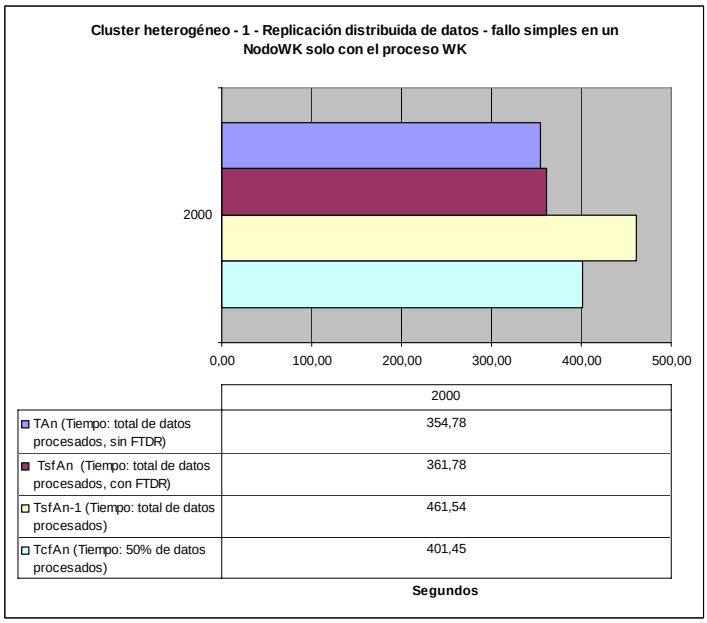

Figura 5-41: *Cluster* local heterogéneo - Replicación distribuida de datos: datos procesados (2000 X 2000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK solo con el proceso WK, después de 50% de datos procesados

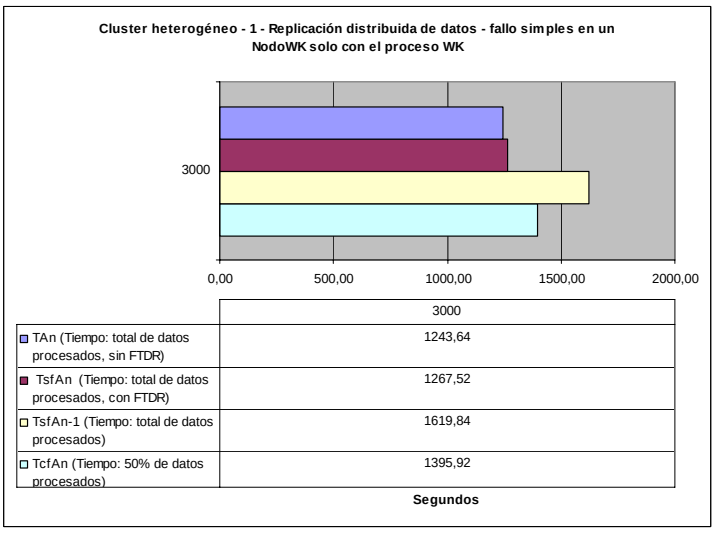

Figura 5-42: *Cluster* local heterogéneo - Replicación distribuida de datos: datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK solo con el proceso WK, después de 50% de datos procesados

5.4.6. Resultados replicación distribuida de datos – *cluster* local homogéneo

El tiempo de ejecución de la aplicación, matrices 1000, 2000 y 3000, replicación distribuida de datos, sin (T_{AN}) y con el *Middleware* FTDR, ejecución con fallo en un Nodo Worker del *cluster* local homogéneo solo con el proceso WK, después de 50% de

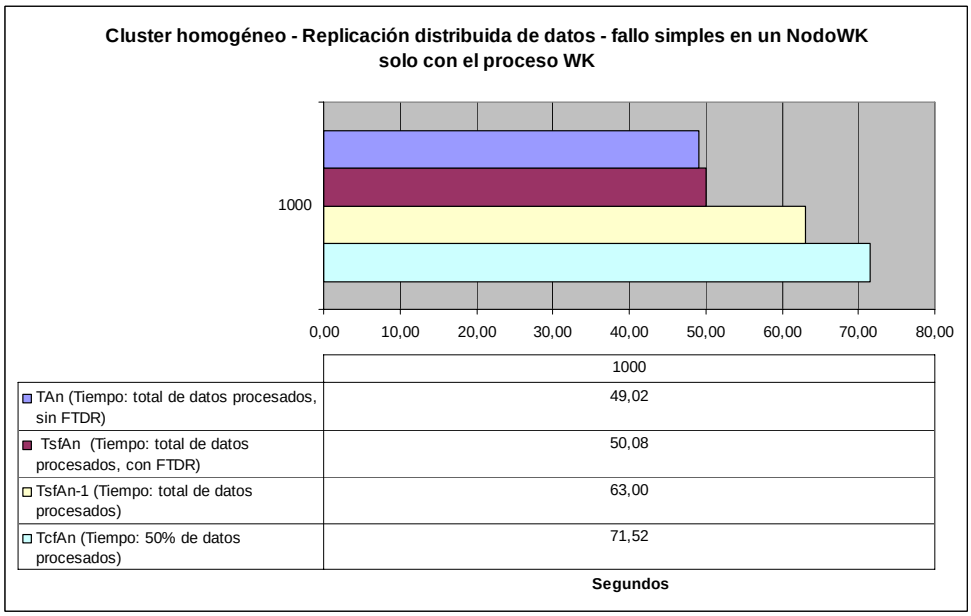

Figura 5-44: *Cluster* local homogéneo - Replicación distribuida de datos: datos

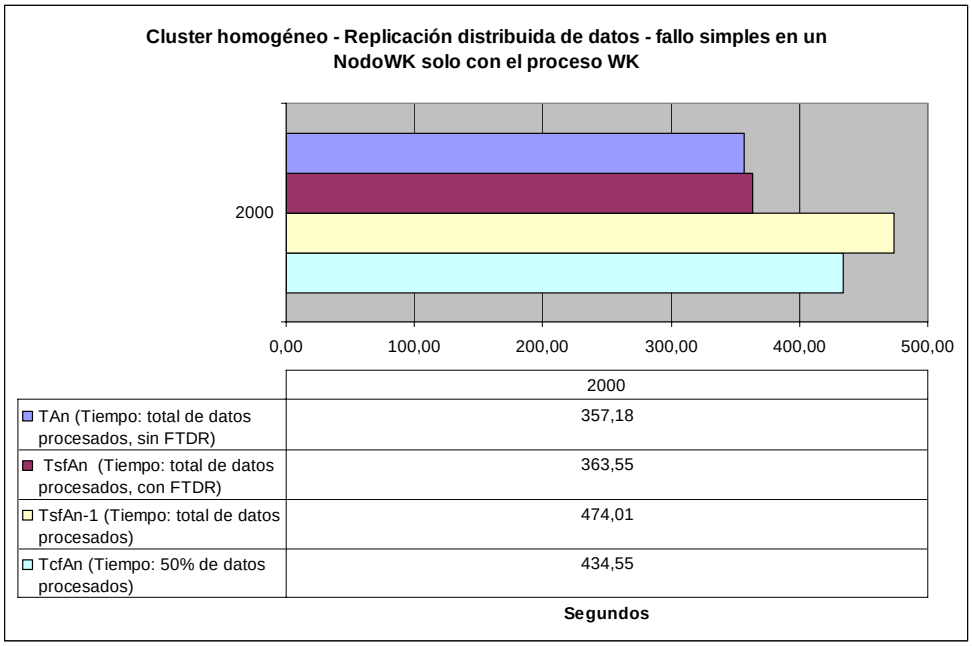

Figura 5-43: *Cluster* local homogéneo - Replicación distribuida de datos: datos

datos procesados ((T_{AN}^{cf}) (Tiempo: 50% de datos procesados)) en el *cluster* local, pueden ser vistos en las Figura 5-44, Figura 5-43 y Figura 5-45.

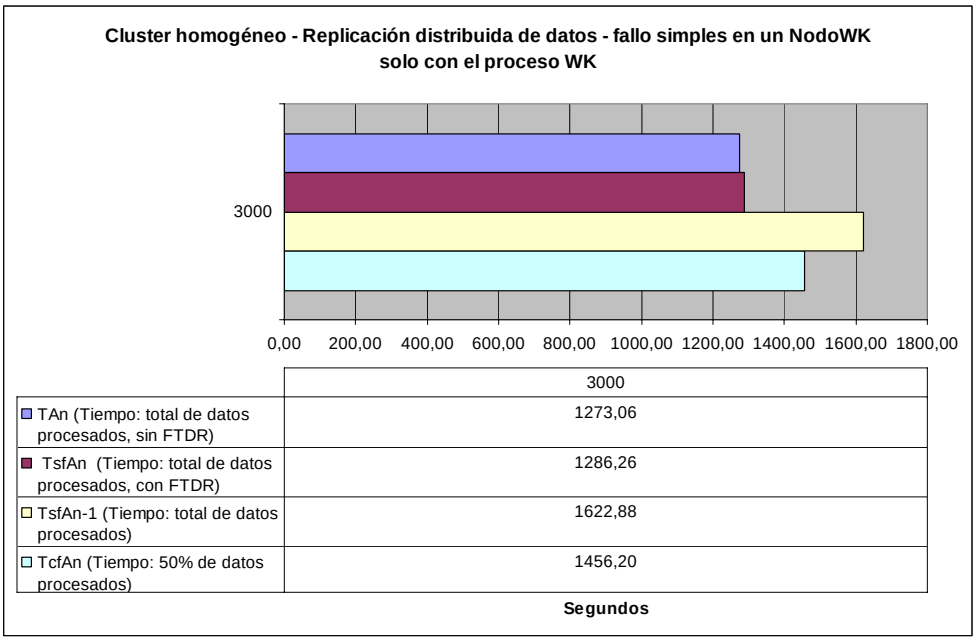

Figura 5-45: *Cluster* local homogéneo - Replicación distribuida de datos: datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK solo con el proceso WK, después de 50% de datos procesados

5.4.7. Fallo simples en un NodoWK PMT

En este ejemplo vemos (Figura 5-46), el MMT (nl0) cuando está recibiendo resultados de los WK y del CM, detecta un fallo en un nodo local (**nl3**) (WK/PMT) y pasa a la fase de recuperación: aísla el nodo nl3 y en este caso, envía el trabajo (WK/PMT) del nodo que falló (nl3) para el nodo nl2, siguiendo los siguientes pasos.

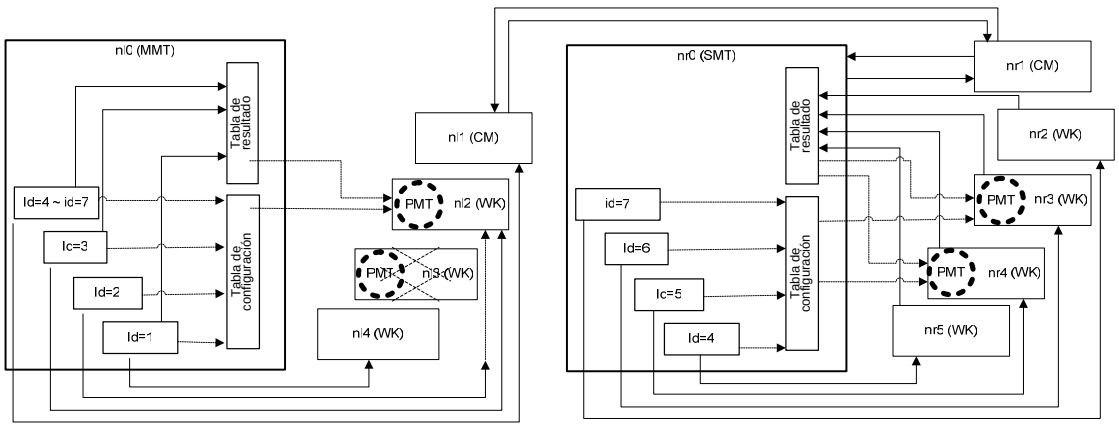

Figura 5-46: Fallo simples en un NodoWK (nl3) con el proceso WK y PMT - replicación distribuida de datos

El *Middleware* (nl0 “MIDMASTER”) se queda verificando si los nodos locales (WK y CM) están “vivos” y los *Middleware* (nl3 y nl2 “MIDWORKER”) se quedan verificando si el nodo local (MMT) está “vivo” (**detección**). El *Middleware* (nl0) detecta que el nodo local **nl3** falló (Figura 5-46), durante la recepción de resultados (**detección**). El *Middleware* (nl0) verifica en la tabla de configuración (Tabla 5-33) se la tarea del

Tabla 5-33: Tabla de configuración/estado - replicación distribuida de datos (13)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	nl2	Si	No

nodo que falló (nl3) fue acabada (**recuperación**). El *Middleware* (nl0) aísla el nodo nl3,

Tabla 5-32: Tabla de configuración/estado - replicación distribuida de datos (14)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	No	No
9	3.1	Nl2	Si	No
8	2.4	nl1	Si	No

o sea, altera el estado del nodo nl3 en la Tabla de Configuración para “No” Enviado y “No” Acabado (Tabla 5-32) (máquina con fallo). A partir de este momento no se envía mas trabajo porque “**No**” Enviado y “**No**” Acabado = Fallo. Una máquina que no ha acabado, no se puede enviar nuevo trabajo (**recuperación**). El *Middleware* (nl0) reasigna

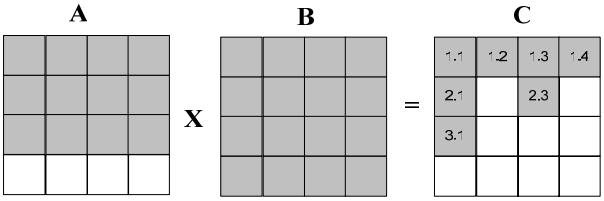

Figura 5-47: Resultado - replicación distribuida de datos (10)

la tarea “2.4” (id=8) para el WK del nodo nl1 (Tabla 5-32) (**recuperación**).

El MMT (nl0) recibe resultado de los workers locales (id=1, id=9) y del CM (id=4) y escribe en la tabla de resultado (matriz resultado “C”) (Figura 5-47).

EL *Middleware* (nl0) actualiza la tabla de configuración (Tabla 5-34) de quién ha recibido resultado (id=1, id=9, id=4) (**protección**).

El *Middleware* (nl0) a replica la tabla de configuración (Tabla 5-34) y la tabla de resultado (matriz resultado “C”) (Figura 5-47) para el PMT que está localizado en el nodo **nl2** (**protección**).

Tabla 5-34: Tabla de configuración/estado - replicación distribuida de datos (15)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	Si
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	No	No
9	3.1	Nl2	Si	Si
8	2.4	nl1	Si	No

El *Middleware* (nl0 “MIDMASTER”) se queda verificando si los nodos locales (WK y CM) están “vivos” y los *Middleware* (nl3 y nl2 “MIDWORKER”) se quedan verificando si el nodo local (MMT) está “vivo” (**detección**).

Mientras no se detecta fallo, el procesamiento continúa hasta el final. l MMT muestra resultado. El MMT **Finaliza** procesamiento y la máquina virtual.

5.4.8. Resultados replicación distribuida de datos – *cluster* local homogéneo

El tiempo de ejecución de la aplicación, matrices 1000, 2000 y 3000, replicación distribuida de datos, sin (T_{AN}) y con el *Middleware* FTDR, ejecución con fallo en un NodoWK del *cluster* local homogéneo con el proceso WK/PMT, después de 50% de datos procesados ((T_{AN}^{cf}) (Tiempo: 50% de datos procesados)) en el *cluster* local, pueden ser vistos en las Figura 5-48, Figura 5-49 y Figura 5-50.

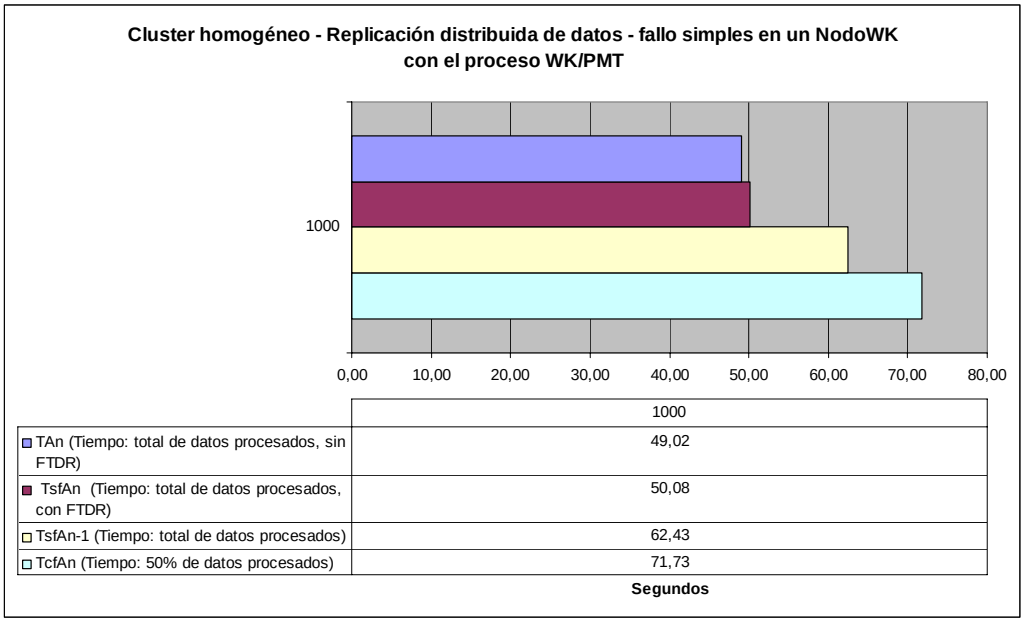

Figura 5-48: *Cluster* local homogéneo - Replicación distribuida de datos: datos procesados (1000 X 1000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK con el proceso WK/PMT, después de 50% de datos procesados

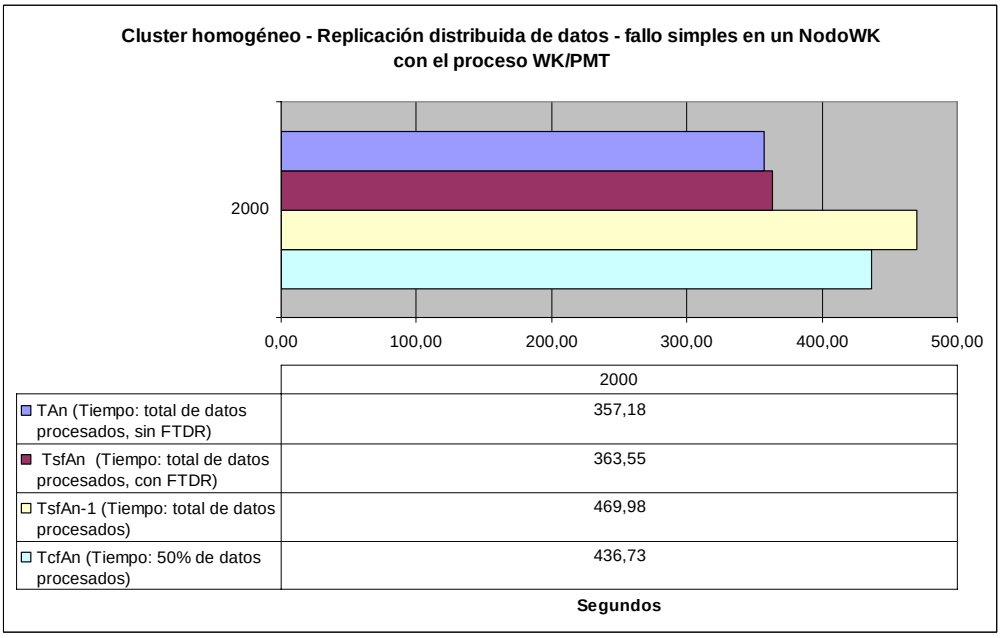

Figura 5-49: *Cluster* local homogéneo - Replicación distribuida de datos: datos procesados (2000 X 2000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK con el proceso WK/PMT, después de 50% de datos procesados

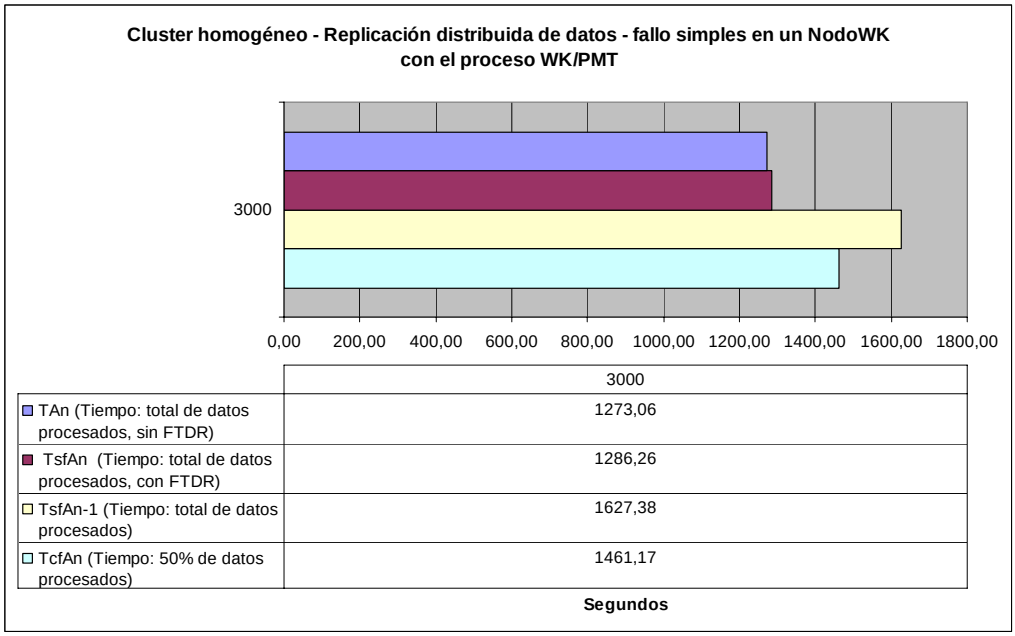

Figura 5-50: *Cluster* local homogéneo - Replicación distribuida de datos: datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, ejecución con fallo en un NodoWK con el proceso WK/PMT, después de 50% de datos procesados

5.4.9. Fallo simples en el Nodo Master

En este ejemplo vemos (Figura 5-51), el MMT (nl0) cuando está recibiendo resultados de los WK y del CM, falla y el nodo local **nl3** (PMT) detecta ese fallo y pasa a la fase de recuperación: el nodo local **nl3** (**nuevo**-**MMT**) asume las funciones de MMT, aísla el nodo

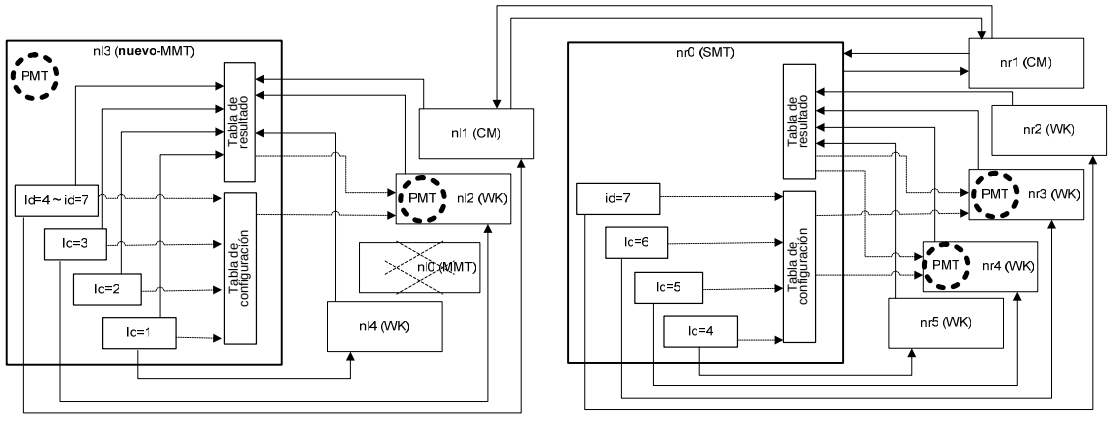

Figura 5-51: Fallo simples en el NodoMT (nl0) - replicación distribuida de datos

que falló nl0 (ex-MMT), siguiendo los siguientes pasos.

El *Middleware* (nl0 “MIDMASTER”) se queda verificando si los nodos locales (WK y CM) están “vivos” y los *Middleware* (nl3 y nl2 “MIDWORKER”) se quedan verificando si el nodo local (MMT) está “vivo” (**detección**). El *Middleware* (nl3 y nl2 “MIDWORKER”) detectan que el nodo local **nl0** (MMT) falló (Figura 5-51), durante la recepción de resultados (**detección**). El nodo local **nl3** (**nuevo-MMT**) asume las funciones de MMT (**recuperación**). El *Middleware* (nl3) aísla el nodo nl0. A partir de este momento no se envía mas trabajo al nodo que falló. Pues en el momento de la re-configuración del *cluster* no pondrá (reasignará) más esa máquina (**recuperación**). El *Middleware* (nl3), **nuevo-MMT,** reconfigura el *cluster* (**recuperación**).El *Middleware* (nl3) verifica en la tabla de configuración (Tabla 5-35) las tareas aún no acabadas e aún no enviadas (**recuperación**).El *Middleware* (nl3) pide que los workers locales (id=1, id=4, id=6, id=8, id=9) re-envíe sus resultados (**recuperación**).

Tabla 5-35: Tabla de configuración/estado - replicación distribuida de datos (19)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	No
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	No
9	3.1	nl2	Si	No

El MMT (nl3) recibe resultado de los workers locales (id=1, id=8, id=9) y escribe en la tabla de resultado (matriz resultado “C”) (Figura 5-52).

EL *Middleware* (nl3) actualiza la tabla de configuración (Tabla 5-36) de quién ha recibido resultado (id=1, id=8, id=9) (**protección**).

El *Middleware* (nl3) replican la tabla de configuración (Tabla 5-36) y la tabla de resultado (matriz resultado “C”) (Figura 5-52), para el PMT que está localizado en el nodo nl2 (**protección**).

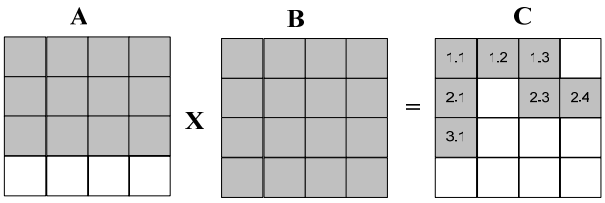

Figura 5-52: Resultado - replicación distribuida de datos (12)

El *Middleware* (nl3 “MIDMASTER”) se queda verificando si los nodos locales (WK y CM) están “vivos” y el *Middleware* (nl2 “MIDWORKER”) se queda verificando si el nodo local (MMT) está “vivo” (**detección**).

Mientras no se detecta fallo, el procesamiento continúa hasta el final. El MMT muestra resultado. El MMT **Finaliza** procesamiento y la máquina virtual.

Tabla 5-36: Tabla de configuración/estado - replicación distribuida de datos (20)

	Configuración		Estado	
id	**Tarea**	**Nodo**	**Enviado**	**Acabado**
1	1.1	nl4	Si	Si
2	1.2	nl3	Si	Si
3	1.3	nl2	Si	Si
4	1.4	nl1	Si	No
5	2.1	nl1	Si	Si
6	2.2	nl1	Si	No
7	2.3	nl1	Si	Si
8	2.4	nl3	Si	Si
9	3.1	nl2	Si	Si

5.4.10. Resultados replicación distribuida de datos – *cluster* local heterogéneo

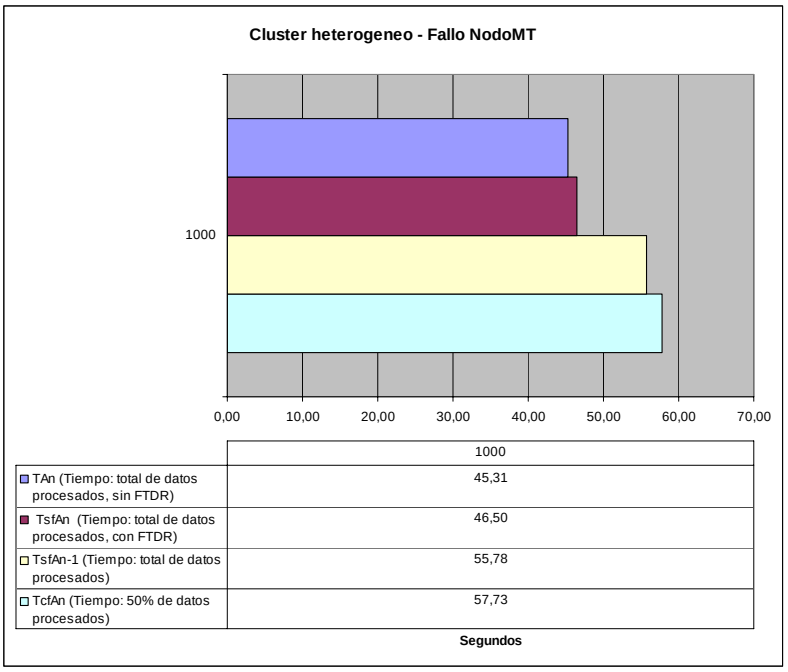

Figura 5-53: *Cluster* local heterogéneo - Replicación distribuida de datos: datos procesados (1000 X 1000), sin/con el *Middleware* FTDR, ejecución con fallo en el NodoMT con el proceso MMT, después de 50% de datos procesados

El tiempo de ejecución de la aplicación, matrices 1000, 2000 y 3000, replicación distribuida de datos, sin (T_{AN}) y con el *Middleware* FTDR, ejecución con fallo en el NodoMT del *cluster* local heterogéneo con el proceso MMT, después de 50% de datos procesados ((T_{AN}^{cf}) (Tiempo: 50% de datos procesados)) en el *cluster* local, pueden ser vistos en las Figura 5-53, Figura 5-54 y Figura 5-55.

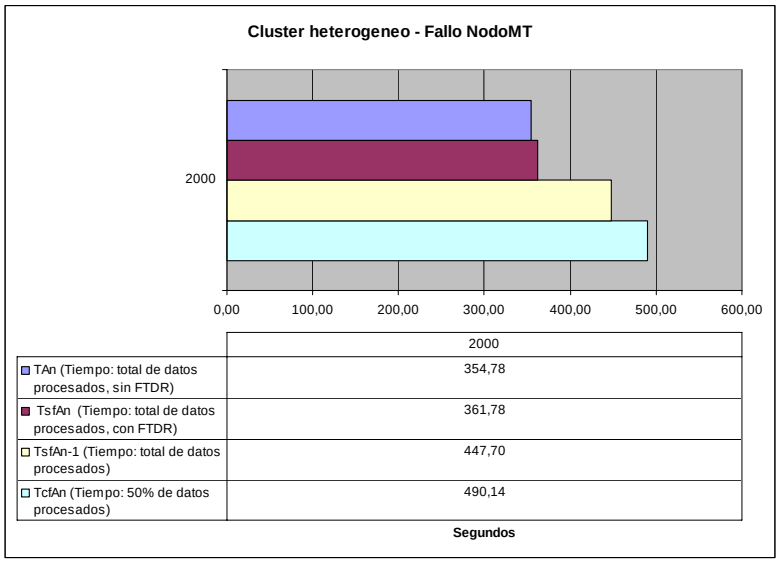

Figura 5-54: *Cluster* local heterogéneo - Replicación distribuida de datos: datos procesados (2000 X 2000), sin/con el *Middleware* FTDR, ejecución con fallo en el NodoMT con el proceso MMT, después de 50% de datos procesados

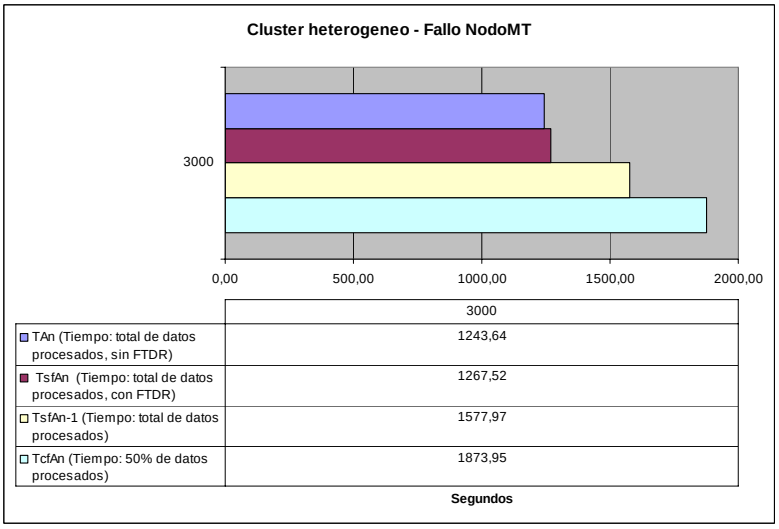

Figura 5-55: *Cluster* local heterogéneo - Replicación distribuida de datos: datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, ejecución con fallo en el NodoMT con el proceso MMT, después de 50% de datos procesados

5.4.11. Resultados replicación distribuida de datos – *cluster* local homogéneo

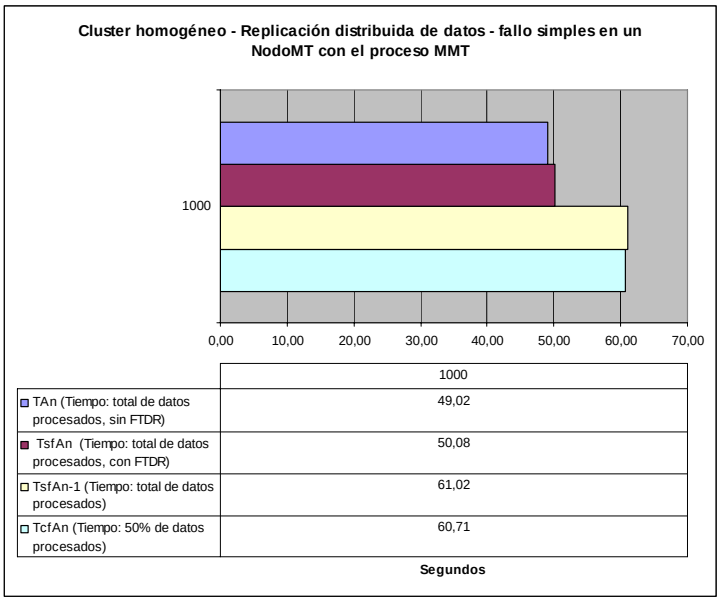

Figura 5-56: *Cluster* local homogéneo - Replicación distribuida de datos: datos procesados (1000 X 1000), sin/con el *Middleware* FTDR, ejecución con fallo en el NodoMT con el proceso MMT, después de 50% de datos procesados

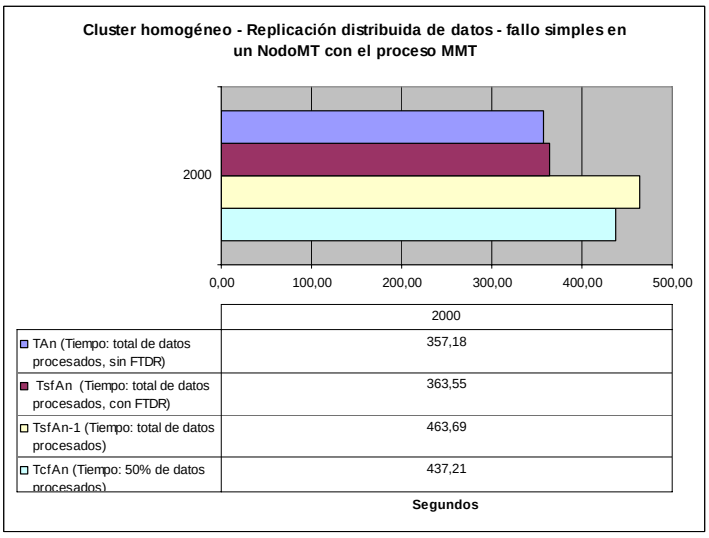

Figura 5-57: *Cluster* local homogéneo - Replicación distribuida de datos: datos procesados (2000 X 2000), sin/con el *Middleware* FTDR, ejecución con fallo en el NodoMT con el proceso MMT, después de 50% de datos procesados

El tiempo de ejecución de la aplicación, matrices 1000, 2000 y 3000, replicación distribuida de datos, sin (T_{AN}) y con el *Middleware* FTDR, ejecución con fallo en el NodoMT del *cluster* local homogéneo con el proceso MMT, después de 50% de datos procesados ((T_{AN}^{cf}) (Tiempo: 50% de datos procesados)) en el *cluster* local, pueden ser vistos en las Figura 5-56, Figura 5-57 y Figura 5-58.

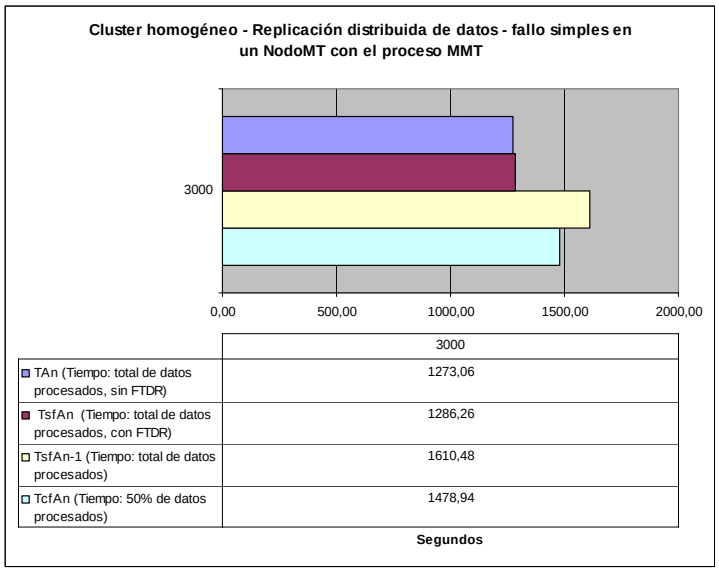

Figura 5-58: *Cluster* local homogéneo - Replicación distribuida de datos: datos procesados (3000 X 3000), sin/con el *Middleware* FTDR, ejecución con fallo en el NodoMT con el proceso MMT, después de 50% de datos procesados

5.4.12. Resultados replicación distribuida de datos – *cluster* local y remoto

El tiempo de ejecución de la aplicación, matriz 3000, replicación distribuida de datos, con el *Middleware* FTDR y ejecución con fallo en un NodoWK del *cluster* remoto con el proceso WK/PMT, después de 50% de datos procesados ((T_{AN}^{cf}) (Tiempo: 50% de datos procesados)), pueden ser vistos en la Figura 5-59.

El tiempo de ejecución de la aplicación, matriz 3000, replicación distribuida de datos, con el *Middleware* FTDR y ejecución con fallo en un NodoMT del *cluster* remoto con el proceso SMT, después de 50% de datos procesados ((T_{AN}^{cf}) (Tiempo: 50% de datos procesados)), pueden ser vistos en la Figura 5-60.

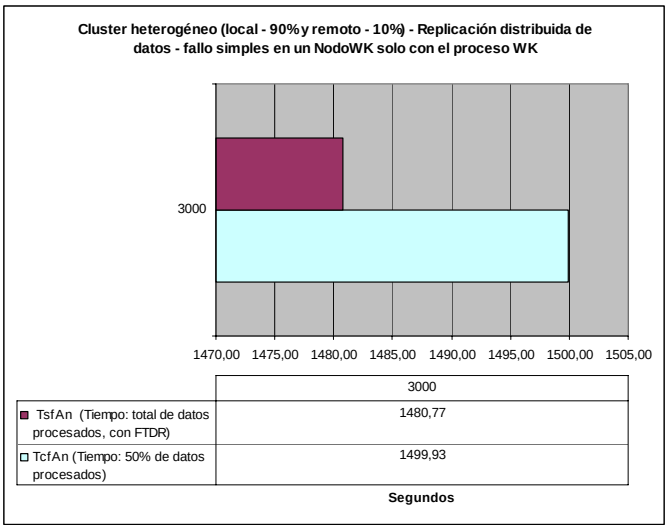

Figura 5-59: *Cluster* local y remoto - Replicación distribuida de datos: datos procesados (3000 X 3000), con el *Middleware* FTDR, ejecución con fallo en un NodoWK con el proceso WK/PMT, después de 50% de datos procesados

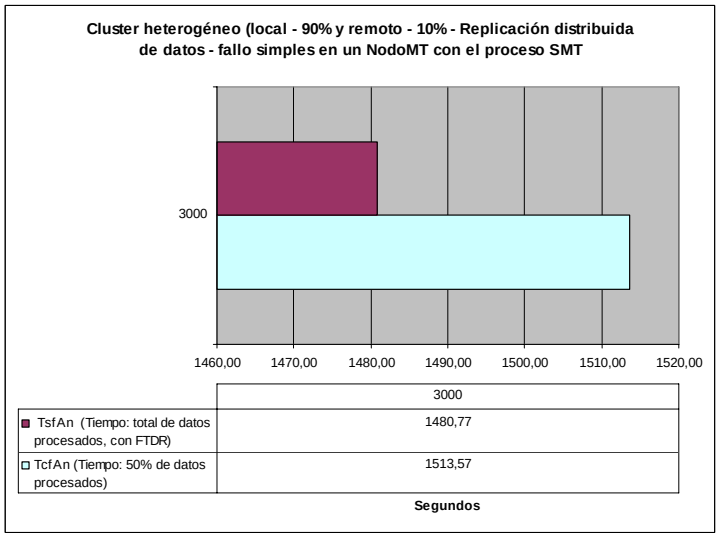

Figura 5-60: *Cluster* local y remoto - Replicación distribuida de datos: datos procesados (3000 X 3000), con el *Middleware* FTDR, ejecución con fallo en un NodoMT con el proceso SMT, después de 50% de datos procesados

5.4.13. Replicación centralizada X distribuida de datos

La Figura 5-62 muestra la comparación entre replicación centralizada X replicación distribuida de datos, sin (T_{AN}) y con el *Middleware* FTDR, ejecución sin fallo (T_{AN}^{sf}) en el *cluster* local heterogéneo.

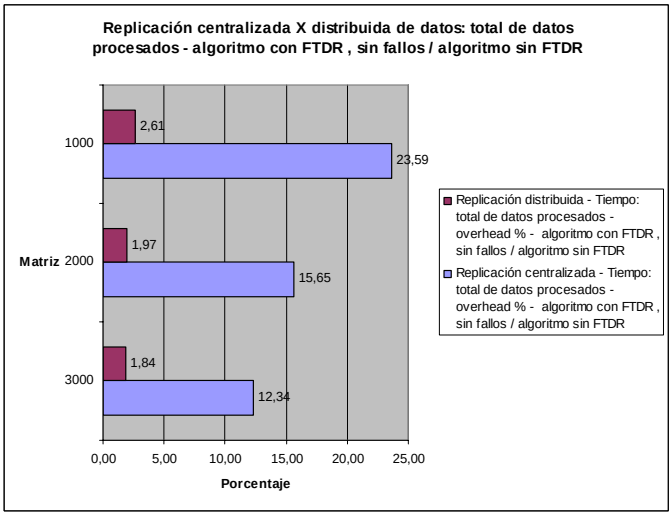

Figura 5-62: *Cluster* local heterogéneo - Replicación centralizada X distribuida de datos – Sin fallo

La Figura 5-61 muestra la comparación entre replicación centralizada X replicación distribuida de datos, sin (T_{AN}) y con el *Middleware* FTDR, ejecución con fallo en el

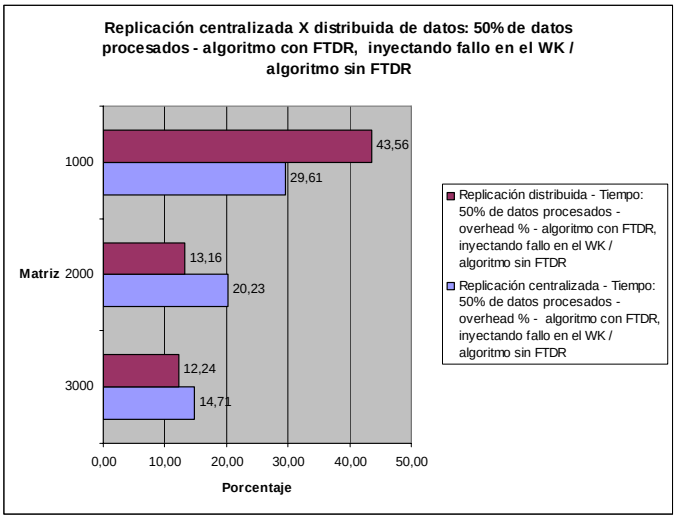

Figura 5-61: *Cluster* heterogéneo - Replicación centralizada X distribuida de datos: 50% de datos procesados - algoritmo con FTDR, inyectando fallo en el WK / algoritmo sin FTDR

NodoWK del *cluster* local homogéneo con el proceso WK, después de 50% de datos procesados ((T_{AN}^{cf}) (Tiempo: 50% de datos procesados)) en el *cluster* local.

5.5. Conclusiones

En este capitulo presentamos los resultados obtenidos a partir de la validación experimental del modelo de tolerancia a fallos propuesto. La interpretación y conclusiones obtenidas de estos datos pueden ser vistas en el capitulo Conclusiones y líneas abiertas.

Conclusiones y líneas abiertas

Conclusiones y principales aportaciones

El problema planteado es de gran vigencia debido al gran auge que tienen los computadores paralelos en la actualidad, teniendo en cuenta que la tecnología actual permite interconectar muchos nodos, que pueden estar distantes, que son dispositivos no especiales (muchas veces de bajo costo), para ejecutar aplicaciones que requieren un elevado tiempo de computación, todo esto hace que se incrementen los potenciales puntos de fallos, es decir, al ejecutar aplicaciones con gran cantidad de nodos, están sujetas a un número mayor de fallos (mayor probabilidad de fallos), cuanto mayor sea su tiempo de ejecución, exigiendo la utilización de esquemas de tolerancia a fallos que proporcionen la capacidad de terminar el procesamiento.

La Tesis presentada plantea un sistema de tolerancia a fallos para multicluster geográficamente distribuidos, con el objetivo de que en presencia de fallos de nodos, el trabajo total se ejecute correctamente, perdiendo el mínimo trabajo realizado posible antes del fallo del nodo.

Se tienen en cuenta la escalabilidad del sistema, el coste (recursos adicionales y *overhead* introducido), la transparencia (que el sistema termine la ejecución en presencia de fallos sin intervención humana), las prestaciones, teniendo en cuenta que la disminución de prestaciones debido al *overhead* introducido para tolerar fallos (generación de redundancia para la protección y detección de fallos) sea asumible por el usuario y que el *overhead* en caso de fallo, debido tanto al proceso de recuperación y reconfiguración como a la perdida de una parte del sistema, esté relacionada con la potencia de computo perdida.

El sistema desarrollado para la incorporación de la tolerancia a fallos en el entorno multicluster consigue un sistema más fiable, sin incorporar recursos *hardware* extra, de forma que partiendo de los elementos no fiables del *cluster,* permite proteger el cómputo realizado por la aplicación frente a fallos, de tal manera que si un ordenador falla otro se encarga de terminar su trabajo y el computo ya realizado está protegido por la Replicación de Datos.

El modelo de tolerancia a fallos está basado en la replicación inicial de procesos y una replicación dinámica durante la ejecución de los datos significativos, con el objetivo de preservar los resultados críticos. Está orientado a aplicaciones con un modelo de ejecución *Master/Worker*, desarrollándose un *middleware,* configurable por el usuario, para que se toleren los fallos de forma transparente durante la ejecución.

El sistema de tolerancia a fallos diseñado, implementa de forma transparente todas las fases de un sistema de tolerancia a fallos (detección de fallos, protección frente a fallos, diagnóstico del fallo, recuperación en caso de fallo y reconfiguración del sistema).

Cumple el requisito de escalabilidad, para ello, se propone básicamente una replicación *intracluster*. Respecto a la replicación entre *clusters*, el sistema está diseñado para que sólo los

datos iniciales necesarios para el cómputo y los resultados sean replicados a través de Internet. De esta forma la incorporación de nuevos *clusters* al multicluster con FTDR, es posible sin afectar significativamente en el *overhead* introducido por el sistema de tolerancia a fallos.

El sistema de tolerancia a fallos es configurable, existe una serie de parámetros que permiten configurar cuestiones tales como el número de fallos simultáneos y otros parámetros de la tolerancia a fallos que permiten controlar el *overhead* introducido por el sistema, estos parámetros permiten balancear la cantidad de recursos dedicados a la tolerancia a fallos. En este modelo de tolerancia a fallo, el *overhead* añadido representa el tiempo necesario para ejecución de la protección (configurable el esquema de protección centralizado o distribuido), detección (configurable el tiempo para el *heartbeat* y el número *heartbeat* para realizar un diagnóstico) de recuperación de fallos (configurable el esquema de protección centralizado o distribuido).

Los datos replicados se utilizan básicamente para tolerancia a fallos y no se utilizan para mejorar el acceso y distribución de los datos en el sistema.

Comenzamos trabajando con una replicación centralizada de datos y evolucionamos hacia una propuesta que utilizara una replicación distribuida de datos. Comparamos ambas propuestas. Cada una de estas opciones favorece el rendimiento en sistemas sin fallos o en presencia de fallos. Sin fallos es mejor una replicación distribuida de datos, ya que funcionando normalmente el *overhead* tal y como hemos visto es mínimo (por debajo del 3%), sin embargo cuando ocurre un fallo tenemos una fase de recuperación más costosa ya que se ha de re-construir el *Master.* Con fallo es mejor la replicación centralizada de datos, teniendo en cuenta que el *overhead* introducido durante la ejecución sin fallos, si no existe una aplicación computacionalmente costosa que permita solapar cómputo con replicación de datos, puede ser significativo.

La inserción del *Middleware* FTDR en aplicaciones con poco cómputo, incrementa el porcentaje de *overhead* introducido, mientras en las aplicaciones con mucho cómputo, disminuye proporcionalmente, en otras palabras, el *overhead* introducido se mantiene constante (o crece lentamente) y a causa de esto cuando el computo de la aplicación crece el *overhead* proporcionalmente se reduce.

La propuesta se ha validado experimentalmente y los resultados obtenidos tanto funcionales como prestacionales son adecuados al problema planteado. Hemos utilizado *clusters* muy heterogéneos, pero para realizar el análisis de la influencia del sistema de tolerancia a fallos, hemos utilizado un sistema más pequeño pero menos heterogéneo.

Los experimentos confirmaron que utilizar este sistema de Replicación de Datos (FTDR) en aplicaciones que son ejecutados en entornos CoHNOW Geográficamente Distribuidos, es una excelente opción para resolver grandes problemas computacionales.

Hemos valorado la ganancia que supone disponer de FTDR en comparación a no tener protección. Probamos experimentalmente que es mejor incluir el *Middleware* FTDR en las aplicaciones, que simplemente relanzar la ejecución del algoritmo desde el inicio. Los resultados experimentales muestran, que antes de llegar al 50% del tiempo de ejecución de la aplicación, si se produce un fallo, es menor el *overhead* introducido por el sistema de tolerancia a fallos propuesto, que relanzar de nuevo la aplicación, con la fuerte penalización que esto supone, además de requerir la intervención del usuario

Los resultados experimentales permiten afirmar que cuanto mayor es el cómputo en cada nodo *Worker*, menor es el coste porcentual de añadir las características de tolerancia a fallos contenidas en FTDR. Vemos que el coste (*overhead*) presentado por la solución debe ser evaluado en función del tamaño del problema que se quiere resolver y de las prestaciones que se desea obtener.

Líneas abiertas

El trabajo realizado plantea diversas líneas abiertas que van desde la detección de nuevos fallos, a la generalización a otras arquitecturas como arquitecturas GRID, o a otros paradigmas de ejecución.

El sistema actual detecta fallos permanentes en los nodos de cómputo, y los aísla. Una posible extensión consistiría en considerar otros tipos de fallos, tales como los fallos transitorios, lo cual permitiría aislar temporalmente un nodo y volverlo a incorporar posteriormente, tal y como se tratan los fallos en el sistema de comunicación. Esto también permitiría incorporar de nuevo nodos aquellos nodos que han fallado, una vez que hayan sido reparados.

Un entorno definido por un conjunto de *clusters* heterogéneos distribuidos y conectados por Internet, puede ser una típica infraestructura de tipo GRID o lo que nosotros denominamos un "*cluster* de *clusters*". En el entorno GRID también es importante garantizar que las aplicaciones finalicen correctamente, es necesario proporcionar una protección intrínseca contra fallos en sus nodos. Frente a los esquemas de tolerancia a fallos con protocolos de *Rollback Recovery* basadas en *checkpoint*, que pueden introducir un gran *overhead*, puede ser interesante estudiar como utilizar el modelo de tolerancia a fallos por replicación de datos (FTDR) para este tipo de arquitecturas.

También es interesante estudiar como aplicarlo a otros paradigmas de programación como por ejemplo el *pipeline,* en el que podemos considerar que cada nodo del *pipeline* funciona como *Worker* y *submaster* del siguiente elemento del *pipe*.

Este estudio para adaptarlo a otros paradigmas de programación y otras arquitectura aportan

la ventaja de ofrecer una solución más portable, por tanto menos dependiente del paradigma y de la arquitectura.

Elaborar un modelo analítico de predicción que contemple el sistema de tolerancia a fallos, ya que el sistema, si dispone del modelo de la aplicación, es bastante predecible, analizando tanto el modelo centralizado o distribuido, esto puede permitir aplicarlo a Sistemas en Tiempo Real (STR) donde la tolerancia a fallos es un factor importante y crítico.

Análisis y evaluación de diferentes opciones de diseño, a fin de ofrecer al usuario distintas posibilidades para configurar el sistema, considerando la aplicación y sus requerimientos. También se podrían elaborar, fruto de la experiencia, recomendaciones al usuario para realizar la aplicación, que pueden afectar a decisiones como el balanceo de carga, para que la aplicación aproveche al máximo el sistema de tolerancia a fallos introduciendo el mínimo *overhead*.

Otras líneas abiertas que completarían el sistema actual que consideramos interesantes, y serviría de base para alguna de las líneas propuestas, consistirían en facilitar al usuario la configuración de los parámetros tales como esquema centralizado/distribuido, o la selección de los tiempos de detección de fallos.

Referencias

1. Agbaria, A. & Friedman, R. (1999), 'Starfish: Fault-tolerant dynamic MPI programs on *clusters* of workstations', *8th IEEE International Symposium on High Performance Distributed Computing*.

2. Amdahl, G.M. (2000), *Readings in computer architecture*, Morgan Kaufmann Publishers Inc., chapter Validity of the single processor approach to achieving large scale computing capabilities, p.79-81.

3. Argollo, E.; Souza, J.R.; Rexachs, D. & Luque, E. (2004), 'Efficient Execution on Long-distance Geographically Distributed Dedicated *Clusters*', *Lecture Notes in Computer Science* **v.3241**, p.311-318.

4. Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.; Laprie, J.; Martins, E. & Powell, D. (1990), 'Fault Injection for Dependability Validation: A Methodology and Some Applications', *IEEE Trans. Softw. Eng.* **16**(2), 166--182.

5. Arlat, J.; Crouzet, Y.; Karlsson, J.; Folkesson, P.; Fuchs, E. & Leber, G.H. (2003), 'Comparison of Physical and Software-Implemented Fault Injection Techniques', *IEEE Trans. Comput.* **52**(9), 1115--1133.

6. Avizienis, A. (1967), 'Design of Fault-Tolerant Computers', *Fall Joint Computer Conference* **v.31**, p.733-743.

7. Avizienis, A. (1997), 'Toward systematic design of fault-tolerant systems', *IEEE, Computer* **v.30, Iss.4**, p.51-58.

8. Avizienis, A.; Laprie, J.; Randel, B. & Landwehr, C. (2004), 'Basic Concepts and Taxonomy of Dependable and Secure Computing', *IEEE Transactions on Dependable and Secure Computing* **v 1, n.1**, p.11-33.

9. Batchu, R.; Neelamegam, J.; Cui, Z.; Beddhu, M.; Skjellum, A.; Dandass, Y. & Apte, M. (2001), 'MPI/FT: architecture and taxonomies for fault-tolerant, message-passing *Middleware* for performance-portable parallel computing', *1st IEEE Int'l Syrup. of Cluster Computing and the Grid, IEEE CS Press*, p.26-33.

10. Beaumont, O.; Boudet, V.; Rastello, F. & Robert, Y. (2001), 'Matrix Multiplication on Heterogeneous Platforms', *IEEE Transactions on Parallel and Distributed Systems* **v12, n.10**, p.1033-1051.

11. Beaumont, O.; Legrand, A. & Robert, Y. (2003), 'The Master-Slave Paradigm with Heterogeneous Processors.', *IEEE Trans. Parallel Distrib. Syst.* **14**(9), 897-908.

12. Birman, K.P. (1997), Building Secure and Reliable Network Applications, *in* 'WWCA', pp. 15-28.

13. Bosilca, G.; Bouteiller, A.; Cappello, F.; Djilali, S.; Fedak, G.; Germain, C.; Herault, T.; Lemarinier, P.; Lodygensky, O.; Magniette, F.; Neri, V. & Selikhov, A. (2002), 'MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes', *SC '02: Proceedings of the Proceedings of the IEEE/ACM SC2002 Conference*, p.29.

14. Buyya, R. (1999), *High Performance Cluster Computing: Architectures and Systems, Volume 1*, Prentice Hall PTR.

15. Buyya, R. (1999), *High Performance Cluster Computing: Programming and Applications, Volume 2*, Prentice Hall PTR.

16. Chakravorty, S. & Kalé, L.V. (2004), 'A Fault Tolerant Protocol for Massively Parallel Systems', *18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 11. Publisher: IEEE Computer Society*, p.212.

17. Chan, A.; Gropp, W. & Lusk, E. (2003),'User's Guide for MPE: Extensions for MPI Programs', Technical report, Mathematics adn Computer Science Divison, Argonne National Laboratory, University of Chicago.

18. Chandy, K.M. & Lamport, L. (1985), 'Distributed snapshots: determining global states of distributed systems', *ACM Trans. Comput. Syst.* **v.3**(1), p.63-75.

19. Chen, Y.; Plank, J.S. & Li, K. (1997), 'CLIP: A Checkpointing Tool for Message-Passing Parallel Programs', *ACM/IEEE conference on Supercomputing*, p.1-11.

20. Condor (2005), 'Condor Project, online at http://www.cs.wisc.edu/condor, Computer Sciences Department, University of Wisconsin, USA'.

21. Coulouris, G.; Dollimore, J. & Kindberg, T. (2001), *Distributed Systems*, Addison Wesley.

22. Cristian, F. (1991), 'Understanding fault-tolerant distributed systems', *Communications of the ACM* **34**(2), 56--78.

23. Culler, D.E. & Singh, J.P. (1999), *Parallel Computer Architecture: a hardware/software approach*, Morgan Kaufmann Publishers, Inc.

24. Elnozahy, E.N.; Alvisi, L.; Wang, Y. & Johson, D.B. (2002), 'A Survey of Rollback-Recovery Protocols in Message-Passing Systems', *ACM Computing Surveys, ACM Press* **v.34**, p.375-408.

25. Fagg, G.E. & Dongarra, J.J. (2000), 'FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World', *Lecture Notes In Computer Science* **v.1908**, p.346-353.

26. Fagg, G.E. & Dongarra, J.J. (2004), 'Building and using a Fault Tolerant MPI implementation', *International Journal of High Performance Applications and Supercomputing* **v.18**, p.353-361.

27. Foster, I. (1995), *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Longman Publishing Co., Inc..

28. Fox, G.C.; Johnson, M.A.; Lyzenga, G.A.; Otto, S.W.; Salmon, J.K. & Walker, D.W. (1988), *Solving problems on concurrent processors. Vol. 1: General techniques and regular problems*, Vol. v.1, Prentice-Hall, Inc.

29. FSF (2005), 'The Free Software Foundation, online at http://www.fsf.org/, USA'.

30. Furtado, A.; Rebouças, A.; Souza, J.R.; Argollo, E.; Rexachs, D. & Luque, E. (2004), 'Improving performance of Long-distance Geographically Distributed Dedicated *Clusters*', *XXXI Seminário Integrado de Software e Hardware*

(SEMISH), SBC2004, p.22-22.

31. Furtado, A.; Rebouças, A.; Souza, J.R.; Rexachs, D. & Luque, E. (2002), 'Architectures for an Efficient Application Execution in a Collection of HNOWS', *Lecture Notes in Computer Science* **v.2474**, p.450-460.

32. Furtado, A.; Souza, J.R.; Rebouças, A.; Argollo, E.; Rexachs, D. & Luque, E. (2003), 'Application Execution over a CoHNOW', *International Conference on Computer Science, Software Engineering, Information Technology, e-business, and Application (CSITeA'03)*.

33. Geist, A. & Engelmann, C. (2002), 'Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors', *online at http://www.ornl.gov/~webworks/cppr/y2001/pres/115978.pdf, U.S. Department of Energy, USA*.

34. Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R. & Sunderam, V.S. (1994), *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, USA.

35. Gropp, W. & Lusk, E. (2002), 'Fault Tolerance in MPI Programs', *special issue of the Journal High Performance Computing Applications (IJHPCA)*.

36. Gropp, W.; Lusk, E.; Ashton, D.; Buntinas, D.; Butler, R.; Chan, A.; Ross, R.; Thakur, R. & Toonen, B. (2005), 'MPICH2 User’s Guide', Mathematics and Computer Science Division - Argonne National Laboratory, Version 0.4.

37. Gropp, W.; Lusk, E.; Doss, N. & Skjellum, A. (1996), 'A high-performance, portable implementation of the MPI Message Passing Interface standard', Parallel Comput. v.22(6), p.789-828.

38. Hennessy, J.L. & Patterson, D.A. (2003), *Computer architecture (3nd ed.): a quantitative approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

39. Hsueh, M.; Tsai, T.K. & Iyer, R.K. (1997), 'Fault Injection Techniques and Tools', *Computer* **30**(4), 75--82.

40. Hwang, K. (1992), *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill Higher Education.

41. Johnson, B.W. (1990), *Design and Analysis of Fault Tolerant Digital Systems*, Addison Wesley.

42. Johnson, B.W. (1990), *Design and Analysis of Fault Tolerant Digital Systems*, Addison Wesley.

43. LAM/MPI (2005), 'Official Site LAM/MPI Parallel Computing, online at http://www.lam-mpi.org/'.

44. Laprie, J. (1991), *Dependability: Basic Concepts and Terminology*, Springer-Verlag.

45. Laprie, J. (1995), 'Dependable Computing and Fault Tolerance: Concepts and Terminology', *Reprinted from FTCS-1.5, 1985, IEEE Proceedings of FTCS-25* **3**, p.2-11.

46. Lee, J.; Chapin, S.J. & Taylor, S. (2003), 'Reliable Heterogeneous Applications', *IEEE Transactions on Reliability* **v.52**(n.3), p.330-339.

47. Louca, S.; Neophytou, N.; Lachanas, A. & Evripidou, P. (2000), 'MPI-FT: Portable Fault Tolerance Scheme for MPI.', *Parallel Processing Letters* **v.10**(4), p.371-382.

48. MacRae, N. (1992), *John Von Neumann: The Scientific Genius Who Pioneered the Modern Computer, Game Theory, Nuclear Deterrence, and Much More*, Reprinted (1999) by the American Mathematical Society.

49. Maehle, E. & Markus, F. (1998), 'Fault-Tolerant Dynamic Task Scheduling Based on Dataflow Graphs', *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems. Kluwer Academic Publishers*, p.357-371.

50. MPI (2005), 'Official MPI (Message Passing Interface) standard, online at http://www.mpi-forum.org/, MPI Forum, USA'.

51. Pacheco, P. (1996), *Parallel Programming with MPI,* Morgan Kaufmann, Inc.

52. Patterson, D.A. & Hennessy, J.L. (1998), *Computer organization & design (2nd ed.): the hardware/software interface*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

53. Pierce, P. & Regnier, G. (1994), '*The Paragon (TM) Implementation of the NX Message Passing Interface*', *IEEE, High-Performance Computing Conference*, p.184-190.

54. Pradhan, D.K. (1996), *Fault Tolerant System Design*, Prentice Hall, New Jersey.

55. Rao, S.; Alvisi, L. & Vin, H.M. (1999), *Egida: An Extensible Toolkit For Low-Overhead Fault-Tolerance*, *in* 'FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing', IEEE Computer Society, p. 48-55.

56. Saito, Y. & Shapiro, M. (2005), 'Optimistic replication', *ACM Comput. Surv.* **37**(1), 42--81.

57. Skillicorn, D.B. (1988), '*A Taxonomy for Computer Architectures*', *IEEE COMPUTER* **v.21, Issue: 11**, p.46- 57.

58. Souza, J.R. (2000), '*Influencia de la comunicación en el rendimiento de un sistema de computación paralela*, basado en redes de estaciones de trabajo', *Master*'s thesis, Universidad Autónoma de Barcelona - UAB.

59. Souza, J.R.; Argollo, E.; Duarte, A.; Rexachs, D. & Luque, E. (2005), 'Fault Tolerant *Master-Worker* over a Multi-*cluster* Architecture', *ParCo2005: In Press*.

60. Souza, J.R.; Furtado, A.; Rebouças, A.; Rexachs, D. & Luque, E. (2002), 'Efficient Algorithm Execution in a Collection of HNOWS', *International Conference on Computer Science, Software Engineering, Information Technology, e-business, and Application (CSITeA'02)*.

61. Souza, J.R.; Rexachs, D. & Luque, E. (2000), '*Análise da distribuição de carga em cluster heterogêneo*', *VI Congreso Argentino de Ciencias de la*

Computación (CACIC'00).

62. Stellner, G. (1996), CoCheck: Checkpointing and Process Migration for MPI, *in* 'Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)', IEEE Computer Society Press, Honolulu, Hawaii, pp. p.526-531.

63. Sterling, T.; Salmon, J.; Becker, D.J. & Savarese, D.F. (1999), *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*, The MIT Press.

64. Sterling, T.L. (2002), Launching into the Future of Commodity Cluster Computing, *in* 'IEEE International Conference on Cluster Computing (CLUSTER 2002)', pp. 345-.

65. Strumpen, V.; Ramkumar, B.; Casavant, T.L. & Reddy, S.M. (1996), 'Perspectives for High Performance Computing in Workstation Networks', *Lecture Notes in Computer Science* **v.1067**, p.880–889.

66. Sun, H.; Han, J.J. & Levendel, H. (2001), 'A Generic Availability Model for Clustered Computing Systems, in 'PRDC '01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing', IEEE Computer Society, Washington, DC, USA, pp. 241-248.

67. Venugopal, S.; Buyya, R. & Ramamohanarao, K. (2005), 'A Taxonomy of Data Grids for Distributed Data Sharing, Management and Processing', Comment: 46 pages, 16 figures, Technical Report.

68. Weber, T.S. (2003), 'Tolerância a falhas: conceitos e exemplos'. Intech Brasil, Distrito 4 da ISA (The Instrumentation, System and Automation Society), 52, pp. 32-42.

69. Weissman, J.B. (1999), 'Fault Tolerant Computing on the Grid: What are My Options?', *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, p.351-352.

70. Weissman, J.B. (2000), 'Fault Tolerant Wide-Area Parallel Computing', *Lecture Notes in Computer Science* **v.1800**, p.1214-1225.