UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

POSGRADO EN CIENCIA E INGENIERIA
DE LA COMPUTACION

UN ESPACIO DE TUPLAS TOLERANTE A FALLAS.

T E S I S

PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS

P R E S E N T A :

JORGE ANDRES LOPEZ VELARDE AVILA

MEXICO, D.F. 2001

A Dwaia

A mis padres

A mi hermana

A mis tíos Gloria y Carlos

A mis tíos Silvia y Antonio

A mis táos: Guadalupe López Velarde Z., Silvia López Velarde Z., Martha Ávila P., Rosario del Pino, José Ávila P. y Luis Velázquez J.

A mis primos López Velarde: Carlos, Ana, Gloria, Rodrigo y Luis

A mis primos Ávila: Antonio, Silvia y Leonardo

A mis amigos: José Aranda F., Henry Pérez L., Moisés Bautista O., Pablo Castillejo G. y Blanca Gil D.

A la Sra. Ginette Elizalde

Al Dr. Alejandro Rodríguez

Prefacio

Este trabajo de tesis forma parte de los siguientes proyectos de investigación: "Ambiente Multipropósito de Edición Colaborativa en Internet e Intranet", con número de referencia: 400316-5-J320433-A del Conacyt y "Consistencia de la Información Distribuida y Problemas de Consenso en Sistemas Distribuidos a Gran Escala" del Conacyt-CNRS, con número de referencia: E130.811/200, de los cuales el Dr. Manuel Romero Salcedo es responsable.

Resumen

El objetivo de esta tesis es crear un nuevo servicio de espacio de tuplas que tolere básicamente los fallos[1] de caída y de omisión en la implementación de un espacio de tuplas. Con este fin, se adoptó una técnica de enmascaramiento de fallas que utiliza un par de implementaciones de un espacio de tuplas que funcionan como primario y de respaldo.

El resultado de esta tesis es un trabajo con los mismos alcances de los proyectos realizados anteriormente[2] y los cuales abordan el problema de construir una implementación toleraute a fallas de un espacio de tuplas. De hecho. cabe resaltar que si bien dichos proyectos son mas ambiciosos al abordar temas como escalabilidad y *performance*, la implementación de esta tesis ofrece un nivel muy similar de confiabilidad (*reliability*) y disponibilidad (*availability*).

Una aportación importante de esta tesis es el algoritmo de *recuperación de estado* y el cual sólo ha sido implementado de manera parcial por el desarrollo de Spring y Largsen [LS99]. Este algoritmo permite restaurar en tiempo de ejecución al espacio que haya experimentado anteriormente un fallo.

[1] A lo largo de la tesis se utilizará el término fallo como traducción de "failure" y el de falla como de "fault"

[2] El capítulo de Proyectos Relacionados hace un resumen de estos proyectos

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

En los último años, el uso y desarrollo acelerado de las redes de comunicación han permitido considerar con una mayor factibilidad el uso compartido de un mayor número de recursos computacionales. A su vez, esta mayor factibilidad de compartición de recursos promueve la construcción de aplicaciones distribuidas donde un conjunto de computadoras conectadas en red puedan ofrecer uu solo servicio. De esta forma, la investigación en nuevos lenguajes y abstracciones que faciliten la coordinación y sincronización entre las computadoras involucradas en una aplicación distribuida ha cobrado un auge singular.

Al mismo tiempo, el incremento en el número de computadoras potenciales en una aplicación distribuida aumenta significativamente la probabilidad de que algo funcione de manera inadecuada. Sin embargo, muchas veces la posibilidad de una interrupción o la pérdida completa de un trabajo computacional en el cual se han invertido una gran cantidad de recursos o tiempo es simplemente inaceptable.

En este contexto, existe entonces la necesidad de utilizar una abstracción que sirva como base para la comunicación y sincronización entre diferentes máquinas y que permita aumentar la probabilidad de que el trabajo que realicen en conjunto termine satisfactoriamente. El concepto de un espacio de tuplas tolerante a fallas es una propuesta de solución a este problema.

El concepto de espacio de tuplas definido originalmente en [CG86a] se clasifica estrictamente dentro de los sistemas de memoria compartida distribuida al proporcionar la abstracción de una memoria común sobre una base

distribuida. En general, el objetivo de un espacio de tuplas es resolver los problemas relativos a la persistencia de los datos y el diseño de protocolos para la comunicación y coordinación en un ambiente con varios procesadores distribuidos o en una sola máquina.

El costo tradicional asociado a una aplicación distribuida está en función del nivel de complejidad asociado a su programación. En este sentido, el paradigma asociado a un espacio de tuplas representa una solución atractiva por su reducido número de primitivas (out(), in(), rd(), inp() e rdp()) y sencilla incorporación a un lenguaje de programación (e.g., Java, C++, etc.). En particular, dicha incorporación permite introducir un base común para la programación de aplicaciones distribuidas y paralelas en distintos modelos como la programación orientado a objetos, la programación funcional o lógica [Bjo93]. Además, Linda permite un desacoplamiento en tiempo entre los procesos y que se traduce en que la comunicación entre los mismos no requiere su existencia simultánea.

Por otra parte, una implementación de un espacio de tuplas está expuesta a sufrir caídas o fallas que detengan completamente la sincronización y comunicación entre procesos. De esta forma, es necesario contar con una implementación que considere una política de tolerancia a fallas con el fin de aumentar la confiabilidad y la disponibilidad en el servicio que ofrece un espacio de tuplas. Esta última característica es deseable principalmente en aplicaciones donde el tiempo que toma una computación es considerable o la comunicación es primordial. Algunos ejemplos de este tipo de problemas son los cálculos de matrices, los sistemas de subasta en línea y, en general, problemas del cálculo paralelo donde trabajan en conjunto un número de procesos distribuidos [Bjo93].

El objetivo específico de esta tesis es entonces diseñar e implementar un espacio de tuplas que tolere básicamente la caída en una versión centralizada de un espacio de tuplas. El método utilizado en el desarrollo de esta tesis consistió inicialmente en un estudio del paradigma de espacio de tuplas y de su semántica de operación. A continuación, fue obligatorio investigar los distintos mecanismos de tolerancia a fallas en sistemas distribuidos existentes y analizar las ventajas y desventajas particulares de su adopción.

El conocimiento a profundidad del paradigma de espacio de tuplas y del tema de tolerancia a fallas en sistemas distribuidos permitió entonces realizar el diseño de un espacio de tuplas tolerante a fallas. Este diseño consiste

específicamente en la utilización de una técnica de enmascaramiento de fallas que utiliza dos implementaciones de un espacio de tuplas redundantes. Es decir, ambos espacios almacenan eventualmente los mismos datos y siguen la misma secuencia en las modificaciones de estos. De esta forma, el sistema completo es capaz de funcionar con uno solo de los espacios y proporcionar de esa forma una mayor disponibilidad en el servicio.

Además, el diseño propuesto permite recuperar y actualizar al espacio de tuplas que haya experimentado un fallo e incrementar así la confiabilidad del sistema.

La evaluación de la propuesta de este trabajo empleo básicamente dos aplicaciones de prueba características de la utilización de un espacio de tuplas y se dividió en términos cuantitativos y cualitativos. La evaluación cualitativa incluyó la verificación del cumplimiento práctico de los requerimientos establecidos en el diseño; mientras que la cuantitativa se refirió a su confiabilidad y disponibilidad teóricas.

Este trabajo incluye también un investigación sobre los proyectos que han abordado el problema de la tolerancia a fallas en un espacio de tuplas y la cual sirve para comparar las características del diseño propuesto. En este aspecto, entre los trabajos previos más importantes se citan los de Xu y Liskov [XL89] del MIT y el sistema Globe de Larsen y Spring de la universidad de Copenhagen [LS99], entre otros. Las propiedades principales que permiten distinguir la propuesta de esta tesis de las demás son una vista transparente de los mecanismos de tolerancia a fallas empleados, la utilización por primera vez del esquema primario-respaldo y el algoritmo de recuperación de un espacio caído o con una falla.

1.1 Organización de la tesis

A continuación, se presenta una descripción de los capítulos restantes de la tesis.

Capítulo 2. Este capítulo trata el concepto de espacio de tuplas como un modelo para la construcción de sistemas distribuidos, sus características principales así como su utilización.

Capítulo 3. El tema de la tolerancia a fallas en los sistemas distribuidos

es abordado en este capítulo. El principio del mismo presenta algunas definiciones básicas, las cuales servirán de marco contextual para los siguientes apartados. A continuación, se abarcan las técnicas principales de tolerancia a fallas junto con los problemas básicos de su implantación en sistemas distribuidos. El capítulo, por último, presenta los conceptos de transacciones y grupos; así como sus ventajas y desventajas para la construcción de sistemas.

Capítulo 4. Este capítulo trata de manera particular el diseño conceptual de la propuesta de un espacio de tuplas tolerante a fallas de este trabajo, denominada `FT-JavaSpaces`. En general, el propósito de este capítulo es presentar las abstracciones principales utilizadas en su implementación. De manera particular, muestra las partes principales del sistema propuesto, sus funciones y la manera en que trabajan e interactúan.

Capítulo 5. El objetivo de este capítulo es presentar la implementación de `FT-JavaSpaces` y describe la estructura y funcionamiento de sus partes desde el punto de vista de las herramientas de programación utilizadas.

Capítulo 6. Este capítulo presenta la evaluación correspondiente a la implementación de `FT-JavaSpaces`. El propósito de esta evaluación consiste principalmente en presentar las característica logradas en términos de sus especificaciones y de las propiedades reales del sistema. La evaluación se divide esencialmente en términos cualitativos y cuantitativos.

Capítulo 7. En este capítulo se tratan los proyectos más importantes que han buscado la implementación de un espacio de tuplas tolerante a fallas, sus características principales y su clasificación. La propuesta de esta tesis, `FT-JavaSpaces`, es comparada con todas las propuestas en un tabla de clasificación al final del capítulo.

Finalmente, se presentan las conclusiones y perspectivas que se obtuvieron en el desarrollo de esta tesis.

Capítulo 2

Espacio de Tuplas

Este capítulo trata el concepto de espacio de tuplas ("Tuplespace") como un modelo para la construcción de sistemas distribuidos, sus características principales así como su utilización.

2.1 Introducción

Un sistema distribuido consta de un conjunto de computadoras autónomas conectadas mediante una red y equipadas con un software distribuido [ea95a]. Este software distribuido esta formado por procesos que comparten información con el fin de realizar un trabajo en común. Una característica importante de estos sistemas es que sus procesos permiten a otros utilizar todos, o algunos, de sus recursos individuales.

En general, de acuerdo a la forma en que interactúan sus procesos, los sistemas distribuidos pueden clasificarse dentro de uno de dos modelos: paso de mensajes o memoria compartida distribuida [ea95a].

En el modelo de paso de mensajes, la única forma en que dos procesos distribuidos pueden interactuar es mediante el intercambio de mensajes a través de una red. Las primitivas básicas de este modelo son *enviar* y *recibir*. *Enviar* transmite un mensaje con datos de un proceso a otro. *Recibir* lee un mensaje transmitido por otro proceso.

El objetivo de la memoria compartida distribuida ("Distributed Shared Memory"), o DSM por sus siglas en inglés, es implementar el modelo de memoria compartida de los sistemas fuertemente acoplados donde todos los

procesadores comparten una sola memoria[1] en un sistema con memorias físicamente distribuidas donde cada procesador posee su propia memoria de trabajo[2] [ea98a].

La memoria compartida distribuida permite sumar las ventajas de un sistema con múltiples procesadores y una sola memoria compartida, como son la comunicación a través de la memoria y una mayor facilidad en la programación, con la escalabilidad [3] de los sistemas formados por distintas computadoras conectadas en red [ea98a].

Los sistemas basados en DSM pueden entonces compararse con los sistemas que emplean paso de mensajes en distintos aspectos:

- En el caso del paso de mensajes, una transferencia de datos entre dos procesos requiere de la existencia al mismo de tiempo de ambos. En DSM, en cambio, el intercambio de datos se realiza a través de variables compartidas ("shared variables").

- El paso de mensajes excluye la posibilidad de que un proceso dañe el espacio de memoria de otro proceso. En el caso de DSM, cualquier proceso puede modificar una variable compartida, lo cual puede solucionarse utilizando un proceso monitor que la encapsule para coordinar todas las operaciones sobre la misma. En conclusión, la desventaja en DSM consiste ahora en que se depende en todo momento de un proceso monitor.

- La sincronización de las operaciones en un sistema de paso de mensajes requiere de primitivas como multicast ordenado, relojes lógicos, etc. ...; mientras que en DSM se pueden tener dos tipos de métodos de sincronización: exclusión mutua, "mutual exclusion" o sincronización condicional ("condition synchronization").

- La mayor eficiencia de uno u otro modelo no está siempre probada a favor de alguno, sino que depende mucho del tipo de problema y ámbito en el cual se trabaje. Esta afirmación queda apoyada por la siguiente cita de George Coulouris [ea95b]:

[1] "Tightly coupled multiprocessor".

[2] "Loosely coupled system"

[3] Entendiéndose escalabilidad como la capacidad de sumar tanto elementos de software como de hardware al sistema sin necesidad de una reconfiguración.

> "No existe una respuesta definitiva a si DSM es preferible al paso de mensajes para una aplicación en particular. DSM es una herramienta prometedora cuyo status último depende de la eficiencia con la que pueda implementarse."

- La memoria compartida permite una comunicación entre procesos donde no es necesario la existencia simultánea del proceso que envía un dato y de quien lo recibe. Por el contrario, en el paso de mensajes es necesaria la existencia de ambos en un mismo lapso de tiempo.

En un nivel de abstracción superior, los sistemas de memoria compartida distribuida y de paso de mensajes sirven de base para la definición de modelos de programación de sistemas formados por procesadores conectados en red y con memorias individuales.

En particular, la memoria compartida distribuida da origen a modelos como el de espacio de tuplas [Gel85] o el de memoria virtual compartida [LS]. Por su parte, el ejemplo más representativo de la programación de sistemas paralelos y distribuidos con el modelo de paso de mensajes es el de PVM [GS].

El modelo orientado a objetos es también otra alternativa para la construcción de sistemas distribuidos donde la comunicación y sincronización están bajo el control de uno o varios objetos y sus métodos asociados. Este modelo guarda en común con el paso de mensajes la necesidad de conocer la identidad y localización de un objeto remoto. Por otra parte, conserva una similitud con la memoria compartida distribuida cuando una vez obtenida la referencia a un objeto remoto puede tratarlo como si se tratara de un objeto local. Un ejemplo representativo de este modelo es el de RMI (Remote Method Invocation) de Sun Microsystems y parte del lenguaje de programación Java [Mic99].

Por último, existen también los modelos de programación funcional [Gol88] y lógica [eab] para la programación de sistemas concurrentes en ambientes paralelos y distribuidos.

2.2 Operaciones básicas

El concepto de espacio de tuplas nace con el proyecto *Linda* de la Universidad de Yale [Gel85]. En la actualidad existen diferentes versiones de Linda, en-

tre las que destacan JavaSpaces [ea99], TSpaces [ea98e] y PageSpace [ea96b].

Un espacio de tuplas consiste en un conjunto de tuplas. Una tupla es una estructura de datos y específicamente un vector consistente de uno o mas campos de algún tipo ("typed fields"), de los cuales al menos uno cuenta con un valor. Un ejemplo de tupla es el siguiente:

("Lucia", 9, true)

donde la tupla tiene tres campos: una cadena de caracteres Lucia, un entero 9 y un valor Booleano True.

Una tupla existente en el espacio es independiente del proceso que la creó al igual que lo es de otros procesos. Un conjunto de tuplas puede también formar una estructura de datos que comprenda uno o mas espacios.

Las operaciones con que se puede acceder a un espacio de tuplas constituyen un número reducido y pueden agregarse a un lenguaje de alto nivel (e.g., C, Fortran, Java) para crear un lenguaje de programación paralela. Un ejemplo de esto son los lenguajes C-Linda y Fortran-Linda [CG86a].

Linda cuenta con tres operaciones *atómicas*[4] fundamentales para insertar, remover e inspeccionar (leer) una tupla de un espacio: **out()**, **in()** y **rd()**, respectivamente. En este contexto, un espacio de tuplas es *distribuido* por que puede ser utilizado por cualquier proceso de un sistema, inclusive por aquellos localizados en máquinas distintas a donde se encuentra el espacio. En consecuencia, un número de operaciones puede ejecutarse de manera concurrente sobre un mismo espacio. La atomicidad de estas operaciones permite también que el retiro de una tupla del espacio sea una acción exclusiva de un solo proceso (*"mutual exclusive"*). Esta última aseveración significa que no existen accesos exclusivos a un espacio completo, sólo sobre tuplas individuales [LS99].

A continuación, se muestra una tabla con las 3 operaciones fundamentales y una descripción de la semántica de las mismas desde el punto de vista de un proceso que interactúa con el espacio.

Enseguida se presenta la definición de las operaciones básicas de Linda de acuerdo con David Gelernter en [Gel85].

[4]Es decir, que se realizan adecuadamente o simplemente no se llevan a cabo.

Operación	Descripción
out	Inserta una tupla en el espacio de tuplas.
in	Retira una tupla del espacio de tuplas.
rd	Lee –hace una copia– de una tupla en el espacio sin retirarla.

Tabla 2.1: Operaciones fundamentales sobre un espacio de tuplas.

Identificadores o nombres de tuplas

Cada tupla contiene un campo del tipo "*nombre*" (N) que sirve como identificador. Este identificador consiste normalmente de un variable o una constante, del tipo *string*. Sin embargo, N puede ser también del tipo numérico.

out()

La operación out() se escribe

out(N, P_2, ..., P_j),

donde P_2, ..., P_j es una lista de parámetros llamados comúnmente *patrón* o "*template*". Cada uno de estos puede ser *actual* (tener un valor) o *formal* (comodín, en el sentido de referirse un tipo de datos determinado). Por otra parte, N es normalmente un parámetro actual. Si se asume que todos los elementos P_j del *patrón* son parámetros formales, la ejecución de la operación **out()** provoca la inserción de la tupla N, P_2,..., P_j en el espacio de tuplas. Un proceso que ejecuta **out()** no se bloquea, sino que continua inmediatamente después de su invocación (ver figura 2.1).

in()

La operación in() se escribe

in(N, P_2, ..., P_j),

donde P_2, ..., P_j es un "*template*" donde, al igual que en out(), cada uno de estos parámetros puede ser *actual* o *formal*. Como demostración, se asume por el momento que todos los elementos P_j del *patrón* son parámetros formales. En el momento de ejecutarse **in()**, si existe una tupla cuyo primer elemento sea del mismo tipo y valor al del identificador N, entonces esta tupla es retirada del espacio y sus valores actuales son asignados a los formales del *patrón* de la operación **in()**. Un proceso que ejecuta **in()** se

bloquea hasta después del momento de realizar la asignación de los valores actuales de la tupla extraída. En caso de no encontrarse ninguna tupla que concuerde ("match") con el *patrón*, **in()** queda suspendida hasta encontrar una disponible y continúa como se describe anteriormente (ver figura 2.1).

rd()

La operación rd() se escribe

$$\text{read}(N, P_2, \ldots, P_j),$$

y es idéntica a **in()** excepto en que, al encontrarse una tupla concordante con el *patrón*, la asignación de parámetros actuales a formales se realiza sin retirar la tupla del espacio (ver figura 2.1).

inp() y rdp()

Linda ofrece también una versión adicional de las operaciones **in()** y **read()** llamadas **inp()** y **readp()**, respectivamente. Estas últimas operaciones se caracterizan por que en caso de no encontrarse una tupla en el espacio que coincida con el *patrón*, los procesos que las ejecutan no se bloquean sino que continúan con su operación normal. Un valor booleano *false* o cero, dependiendo de la implementación, es devuelto por estas operaciones cuando no existe una tupla coincidente en el espacio.

La ejecución de estas operaciones involucra la inspección del estado "presente" de un espacio de tuplas. Sin embargo, en el caso de contar con un espacio de tuplas conformado por varios espacios de tuplas individuales, ubicados en máquinas distintas, la definición del estado "presente" queda ligada a la del modelo de memoria empleado en la misma implementación distribuida [LS99].

eval()

La operación eval() inserta *tuplas activas* en el espacio de tuplas a diferencia de las llamadas *tuplas pasivas* que inserta **out()**. Una *tupla activa* define básicamente una computación a realizar. El propósito de **eval()** es que el espacio realiza una cierta acción definida en una *tupla activa* y que ésta misma se convierta en una *tupla pasiva* como resultado de dicha acción [CG89].

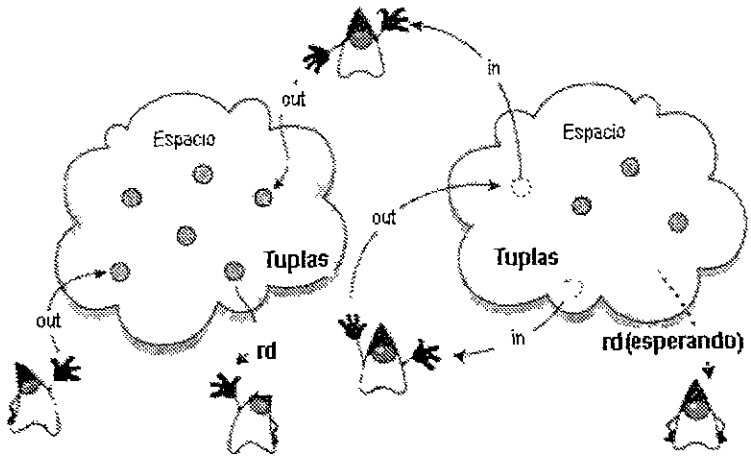

Figura 2.1: Operaciones básicas en un espacio de tuplas.

Eval(t), en otras palabras, adiciona una tupla sin **evaluar** ("unevaluated tuple") al espacio [ea85]. La evaluación de una tupla comienza en el momento que ingresa al espacio, en forma concurrente al proceso que ejecutó eval(), y sus campos son evaluados en un orden arbitrario. La *tupla pasiva* resultante puede ser leída o retirada posteriormente del espacio utilizando las operaciones rd() o in(), respectivamente.

Por ejemplo, la ejecución de la instrucción

eval("M", f(x))

inserta una *tupla activa*

(*eval*("M"), *eval*(f(x)))

en el espacio. Si f(x), por ejemplo, da como resultado un entero j, la *tupla activa* (*eval*("M"), *eval*(f(x))) se convierte en la *tupla pasiva*

("M", j)

cuando termina la evaluación.

La utilidad de eval() se da en el hecho de que un proceso puede generar otro definiendo una *tupla activa* e insertándola en el espacio.

No determinismo y coincidencia de múltiples tuplas con un *patrón*

En el caso de existir mas de una tupla coincidente con un *patrón*, las operaciones **rd()** e **in()** eligen solamente una. La semántica original de Linda obliga a que la tupla devuelta como resultado de estas operaciones se escoja de manera arbitraria sobre el conjunto de tuplas coincidentes. Esta situación plantea un problema cuando algún proceso requiere la lectura o el retiro de todas las tuplas concordantes con un mismo patrón.

En este sentido, las soluciones se dividen en aquellas que buscan incluir nuevas operaciones al modelo de Linda y otras donde las aplicaciones mismas son las encargadas de resolver el problema.

En [RW96] se incluyen las operaciones `copy-collect` y `collect` para copiar y retirar, respectivamente, todas las tuplas coincidentes con un patrón específico. Ambas utilizan otro espacio de tuplas para almacenar el conjunto de tuplas resultantes.

El uso de un índice es el método más utilizado por las aplicaciones cuando necesitan leer o copiar todas las tuplas que concuerdan con un patrón. En estos casos, el orden y número existente de un cierto tipo de tuplas es responsabilidad de un proceso monitor. El manejo de un índice conlleva la existencia de un mecanismo de iteración; sin embargo, el modelo de Linda no incluye dicho mecanismo por diversas razones. En primer lugar, la iteración sobre un conjunto de tuplas puede no arrojar el mismo resultado en una misma iteración subsecuente debido a que durante el transcurso de las mismas pueden ingresar o desaparecer tuplas coincidentes con el patrón. Inclusive, el tiempo de ejecución de un mecanismo de iteración puede ser indefinido cuando ingresan continuamente nuevas tuplas al espacio. La otra opción consiste en bloquear por completo a otros procesos el acceso al espacio de tuplas. La iteración puede comenzar una vez logrado dicho bloqueo, pero las tuplas que pretendan ingresar al espacio durante la misma serán invisibles.

El peor caso en el no determinismo del resultado surge cuando un proceso necesita copiar o retirar todas las tuplas de un espacio. El uso de funciones de asociación ("hashing") es una de las alternativas con mayor potencial para la solución de este problema; además, este tipo de funciones son también una fuente para el mejoramiento del rendimiento [Nit98].

Un resumen de distintas soluciones al problema del no determinismo en la elección de una tupla se presenta en [LS99].

2.3 Nombres estructurados

Cada tupla tiene un nombre estructurado formado por el valor (siempre actual) del identificador de tupla N junto con la lista $F_2, \ldots, F_k$ de valores actuales de $P_2, \ldots, P_j$[5]. Por ejemplo,

```
in(P, i:integer, FALSE)
```

busca extraer una tupla con un nombre estructurado igual a "P,,FALSE".

La forma en que la tupla anterior es ingresada al espacio es mediante la operación

```
out(P, i:integer,FALSE)
```

aunque en este caso su nombre estructurado carece de importancia.

Los nombres estructurados permiten resultados similares al de una operación *select* de SQL y contar con un espacio de tuplas donde las tuplas sean referenciadas ("addresable") por su contenido [Gel85].

2.4 Reglas de coincidencia entre una tupla y un *patrón*

Una tupla T y un *patrón* P coinciden cuando se cumplen las siguientes condiciones [Lei89]:

- T y P tienen el mismo número de campos.
- Los campos correspondientes son del mismo tipo.

[5] $P_2, \ldots, P_j$ puede incluir valores tanto actuales como formales.

- Cada par de campos correspondientes C_T y C_P coinciden de alguna de las siguientes formas:

  - Si ambos son actuales, C_T y C_P coinciden si sus respectivos valores son iguales en el contexto del lenguaje utilizado.

  - Si C_P es formal y C_T actual, ambos coinciden. En el caso de que *todos* los campos coincidan, el valor de C_T se asigna a la variable especificada en el *patrón*, si existe.

  - Si C_P es actual y C_T es formal, ambos coinciden incondicionalmente. El valor actual de C_P es descartado.

  - Si tanto C_T como C_P son formales, siempre coinciden.

Por ejemplo, la tupla ("Lucia", i:integer) coincide con ("Lucia",9), pero no con ("Lucia",15.3), ("Jorge",9) ni ("Lucia", i:float) o ("Lucia",9,7).

2.5 Orden de las operaciones

El orden de las operaciones que se realizan en un espacio de tuplas puede clasificarse en dos ramos [LS99]: *globalmente desordenadas* y *parcialmente ordenadas*.

Las operaciones realizadas por procesos independientes y que no guardan ninguna sincronización entre ellos se clasifican como *globalmente desordenadas*.

Un orden entre diferentes procesos o hilos de control ("threads") sólo es posible mediante la cooperación entre ellos y la utilización de un protocolo especializado para la sincronización de sus acciones. Además, dicho protocolo debe obedecer a los requerimientos de una aplicación específica.

Desde el punto de vista del orden en que un proceso realiza sus operaciones, éstas pueden clasificarse como *parcialmente ordenadas* si cumplen con una secuencia predeterminada en el programa.

2.6 Actualización de tuplas

Una de las características de un espacio de tuplas es la ausencia de un operador de asignación ("="). Esto significa que las tuplas no pueden ser modificadas dentro del espacio, sino que deben de seguirse los siguientes tres pasos:

1. Remover la tupla del espacio utilizando una operación **in()**.
2. Actualizar los campos necesarios en la tupla.
3. Insertar de nuevo la tupla en el espacio mediante una operación **out()**.

Una ventaja de este proceso es que se elimina el problema de consistencia de las tuplas y en que la sincronización de los procesos se realiza de manera automática, aunque de manera no determinista. La atomicidad de las operaciones básicas in() y out() permite hacer esta última aseveración.

La desventaja consiste en que una pequeña modificación es costosa en términos del número de operaciones necesarias.

2.7 Estado interno

Los siguientes supuestos definen el estado interno de un servidor que implementa un servicio de espacio e tuplas:

- Las variables de estado de un servicio de espacio de tuplas son las tuplas mismas.
- El estado interno de un servicio de espacio de tuplas es una función de sus variables de estado, cuyo número no es necesariamente constante en el tiempo.
- Un servicio de espacio de tuplas es también un servicio con estado, es decir:

  > su estado interno actual es una función de la secuencia de operaciones realizadas anteriormente sobre el espacio.

- Un servicio de espacio de tuplas es implementado por un algoritmo determinístico, es decir:

la semántica de cada respuesta que genera es una función de los argumentos que recibe y del estado interno actual del servicio.

- El estado interno inicial de un espacio se caracteriza por ser el conjunto vacío de variables de estado.

2.8 Discusión

Este capítulo trató del modelo de espacio de tuplas y sus características principales. Entre los principales puntos tratados se encuentran:

- El concepto de espacio de tuplas se clasifica como un sistema de memoria compartida distribuida por que ofrece la abstracción de una sola memoria global en un ambiente conformado por distintas computadoras conectadas en red.
- Un espacio de tuplas es accedido por medio de seis operaciones atómicas fundamentales: out(), rd(), rdp(), in(), inp() y eval().
- El orden de las operaciones globales es completamente no determinístico y queda en manos del programador de aplicaciones.

En general, el modelo de espacio de tuplas es una alternativa para la solución de los problemas de comunicación y sincronización entre procesos. Junto con los demás sistemas de memoria compartida, su objetivo principal es disminuir la complejidad en la solución de dichos problemas en comparación con su contraparte en un sistema basado en el modelo de paso de mensajes.

En [CG93], David Gelernter y Nicholas Carriero afirman que la relación entre el modelo de Linda y el de paso de mensajes es similar al existente entre un lenguaje de alto nivel y el lenguaje ensamblador. Ellos afirman que existen ciertos problemas que pueden resolverse más fácilmente utilizando Linda y, además, con un rendimiento comparable al de una misma solución utilizando paso de mensajes.

Capítulo 3

Tolerancia a fallas

Este capítulo trata el tema de la tolerancia a fallas en los sistemas distribuidos. El principio del mismo presenta algunas definiciones básicas, las cuales servirán de marco contextual para los siguientes apartados. Las siguientes secciones abarcan las técnicas principales de tolerancia a fallas junto con los problemas básicos de su implantación en sistemas distribuidos. El capítulo, por último, presenta los conceptos de transacciones y grupos; así como sus ventajas y desventajas para la construcción de sistemas.

3.1 Introducción

Uno de los principales objetivos de las computadoras, desde su creación, ha sido sustituir al hombre en tareas arduas, complicadas y monótonas. En este sentido, las computadoras son ahora las encargadas de funciones críticas en los negocios, la salud, sistemas de transporte, etc. La responsabilidad actual delegada a las computadoras es tal que una desviación no planeada en el funcionamiento normal de un sistema de "hardware" y "software" puede significar la pérdida de dinero o de vidas. La caída, por ejemplo, de la red de larga distancia de AT&T por nueve horas en 1990 significó un pérdida de $60 a $75 millones de dólares, lo cual significa más de $100,000 dólares por minuto [SV97].

3.2 Conceptos básicos

El estudio de este problema, como cualquier otro, requiere de la identificación de los actores e interacciones principales. En este sentido, el primero

paso consiste en establecer y definir los conceptos básicos de la tolerancia a fallas.

3.2.1 Servicio y servidor

En términos generales, el entendimiento de un sistema computacional se facilita cuando se utilizan los conceptos de *servicio* y *servidor*. A continuación se presenta su definición de acuerdo con [Cri91]:

Servicio ("service"). Un *servicio* define un una colección de operaciones cuya ejecución puede ser iniciada ("triggered") por orden de los usuarios del mismo (clientes) o el paso del tiempo. La ejecución de una operación puede traer como consecuencia un cambio en el estado interno del servicio o la generación de un resultado ("output").

Servidor ("server"). Las operaciones definidas teóricamente por un *servicio* son implementadas en la práctica por un *servidor*. Un *servidor* no expone a sus clientes el estado interno del *servicio* ni los detalles pertenecientes a la implementación de sus operaciones. El estado interno de un servicio está normalmente representado por una abstracción de datos ("data abstraction") en el servidor y es el resultado de la secuencia de operaciones ejecutadas anteriormente (peticiones), o "inputs". El comportamiento externo de un servidor o estado externo del servicio es también una abstracción del mismo estado interno del servicio. En este sentido, algunos de los cambios en el estado interno de un servicio pueden reflejarse también en el comportamiento externo de un servidor (i.e., en un resultado).

Un servidor, en computación, utiliza normalmente otros servidores de de igual o más bajo nivel, lo cual establece una relación de dependencia. Un servidor **u** *depende* entonces de otro servidor **r** si el desempeño correcto de **u** está en función del desempeño correcto de r [Cri91]. En este sentido, el servidor **u** se encuentra en un nivel de abstracción más alto que el de **r**; mientras que **r** se denomina como de más *bajo nivel* en comparación con **u** (ver ejemplo en Fig. 3.1).

3.2.2 Fallo, error y falla

En el diseño de un sistema tolerante a fallas existen tres términos fundamentales: falla ("fault"), error ("error") y fallo ("failure"). Entre estos existe

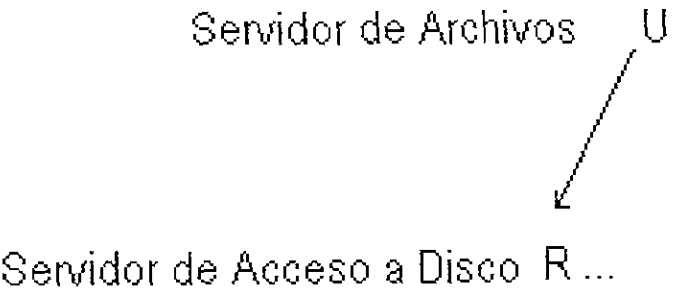

Figura 3.1: Ejemplo de la relación entre dos servidor ubicados en distintos niveles.

una relación causa-efecto, donde las fallas son el origen de errores y los errores son el origen de fallos. A continuación se presenta su definición, las cuales concuerdan con las encontradas en [Joh89]:

Falla ("fault") significa un defecto físico o imperfección de un componente de "hardware" o "software". Un ciclo ("loop") perteneciente a un programa x y del cual es imposible salir constituye un ejemplo de una falla de programación.

Error es la manifestación de un *falla* e involucra la desviación de un *estado* correcto ("correctness") o definido con exactitud ("accuracy"). Específicamente, un error es la ocurrencia de un valor incorrecto en alguna unidad de información dentro de un sistema, o servidor. Siguiendo con el ejemplo anterior, considérese que la condición para permanecer en el ciclo depende del valor *verdadero* ("true") de una variable booleana B. Una vez iniciado el ciclo y debido a la falla de programación en el algoritmo, el valor *verdadero* de B es imposible cambiarlo. En este ejemplo, un error se produce cuando el programa x debe salir del ciclo pero no le es posible cambiar el valor booleano de B a falso ("false"). En este caso, existe un error en el valor de la variable B. Si el algoritmo es parte de un servidor, el error forma parte también del estado interno del servicio que implementa.

Fallo ("failure" o "malfunction") conlleva el resultado de la **no** ejecución de una función o acción esperada por un servidor, ya sea un cambio en el estado interno de un servicio o una respuesta ("output"). Un fallo lo constituye también la ejecución de una operación de manera cualitativa o cuantitativamente anormal. Un error no significa el fallo de un servidor. El programa *x*, por ejemplo, puede estar funcionando correctamente aún cuando no salga del ciclo, siempre y cuando no tenga necesidad de salir del mismo. En el momento en que sea necesario su salida del ciclo y no pueda hacerlo, el programa *x* experimentará un fallo y se producirá un error en *B*.

Un error puede formar parte del estado interno de un servicio, lo cual permite que sea detectado y evaluado –es decir, forma parte de una abstracción de datos concreta en el servidor. Un fallo, por el contrario, no forma parte del estado interno de un servidor.

Los fallos son normalmente observables mediante el uso mecanismos de detección de ciertos eventos. Sin embargo, la ocurrencia de un fallo es deducida normalmente después de encontrarse un error en el estado interno de un servicio.

3.2.3 Enfoques principales

Son dos los enfoques principales que existen para mantener o garantizar el funcionamiento correcto de un servidor, o sistema, cuando existen fallas en sus componentes de Hardware y software o los servidores de más bajo nivel que utiliza:

Evitar las Fallas ("Fault-avoidance"). En esta estrategia se busca construir un servidor, o sistema, a partir de elementos con una mínima probabilidad de funcionar incorrectamente. Desde otro punto de vista, busca construir un servidor con una baja posibilidad de experimentar un fallo. En este categoría se ubican todos los esfuerzos por eliminar las fallas en la implementación de un componente o servicio. Las revisiones de diseño, las pruebas y los controles de calidad son algunos ejemplos de acciones comprendidas en este esfuerzo. Esta estrategia no representa un reto en el diseño del sistema y se halla sustancialmente limitada por factores económicos y tecnológicos [Bab].

Tolerancia a Fallas ("Fault-Tolerance"). La tolerancia a fallas consiste en la habilidad de un servidor para continuar funcionado de acuerdo a sus

especificaciones ("specifications") a pesar de la existencia de fallas en los servidores de más bajo nivel que utiliza. De esta forma, un servidor no puede ser nunca tolerante a sus propias fallas[1]. La tolerancia a fallas requiere de métodos como el enmascaramiento transparente de fallos o la reinicialización de una computación a partir de un estado consistente anterior. En ambos casos, el objetivo final es evitar que el servidor o sistema tolerante a fallas experimente un fallo.

Los métodos de *evitar las fallas* y el de *tolerancia a las mismas* son obviamente complementarios.

Por otra parte, el esquema de tolerancia a fallas basado en técnicas de "software" es el utilizado comúnmente en la práctica. Dichas técnicas, a su vez, pueden dividirse en dos tipos de enfoque, de acuerdo al tipo de fallas considerado [Gar98]:

Software tolerante a fallas ("fault-tolerant software"). Estas técnicas buscan tolerar fallas tanto en el nivel de hardware, como en el de los procesos o aplicaciones completas (software). Los problemas relacionados con este tipo de fallas son los abordados en este capítulo.

Tolerancia a fallas en el software ("software fault-tolerance"). Estas técnicas, en cambio, buscan tolerar únicamente fallas ("bugs") en la implementación de un servicio ("diseño"). Un estudio a profundidad de este tema es posible encontrarlo en [eaa]

3.3 Métodos de evaluación cuantitativa

Dos de las formas mediante las cuales se mide comúnmente la habilidad de un sistema[2] para tolerar fallas es utilizando los conceptos de *confiabilidad* ("reliability") y *disponibilidad* ("availability").

3.3.1 Confiabilidad

La *confiabilidad* $R(t)$ de un sistema es una función del tiempo. La *confiabilidad* está definida como la probabilidad condicional de que un sistema fun-

[1] Sin embargo, las especificaciones de un servidor pueden incluir un determinado tipo de fallos admisibles.

[2] Sistema se define comúnmente como una colección de elementos de hardware y software organizados de una forma específica para cumplir con un objetivo en común. Si dicho objetivo común constituye la implementación de un servicio, entonces servidor y sistema pueden considerarse sinónimos

cione correctamente en un **intervalo** de tiempo $[t_0,t]$ donde pueden ocurrir fallos potenciales y t_0 representa un instante en el cual el sistema comienza a operar sin fallas [Joh89]. El término probabilidad condicional indica que el sistema debe encontrarse operando correctamente al principio del intervalo. El requerimiento de un sistema, por ejemplo, puede requerir de una *confiabilidad* de 0.9999 para un tiempo de ejecución de 10 horas. En otras palabras, la probabilidad de producirse un fallo en dicho lapso de tiempo es a lo máximo de 10^{-4}.

Un sistema tolerante a fallas no necesariamente cuenta con una gran confiabilidad. Un sistema, por ejemplo, puede ser capaz de tolerar todos los fallos posibles y ofrecer una confiabilidad muy baja si la probabilidad de que se produzcan estos fallos es muy alta. Si el número de fallos que puede tolerar un sistema rebasa un cierto límite ("resiliency") y éstos cuentan una probabilidad muy alta de producirse, la propiedad de tolerancia a fallas de un sistema puede desaparecer. Así mismo, un sistema puede estar construído a partir de componentes con una probabilidad muy baja de sufrir un fallo, pero no contar con la propiedad de ser tolerante a fallas. En este último caso, el sistema cuenta con una alta confiabilidad, pero no cuenta con la posibilidad de continuar con un funcionamiento correcto si alguno de sus componentes sufre un fallo. En este contexto, un componente o sistema funciona correctamente si trabaja de acuerdo a sus especificaciones.

3.3.2 Disponibilidad

La *disponibilidad* $A(t)$ es también una función del tiempo. La *disponibilidad* se define como la probabilidad de que un sistema realice sus funciones y se mantenga operando correctamente en un **instante** de tiempo t [Joh89]. De esta forma, la disponibilidad depende de la frecuencia con que un sistema sufra fallos y del tiempo de recuperación necesario para alcanzar un funcionamiento correcto de nuevo. En términos generales, la disponibilidad es una propiedad que busca garantizar un funcionamiento de acuerdo a las especificaciones lo más frecuentemente posible.

Una de las formas más comunes de elevar la *disponibilidad* de un sistema es el uso de elementos físicamente redundantes.

En el contexto de la tolerancia a fallas, la *confiabilidad* puede considerarse como la expectativa de los usuarios; mientras que la *disponibilidad* es la garantía proporcionada por los diseñadores [Gar98]. La confiabilidad es

una propiedad importante en aplicaciones críticas, donde un solo fallo puede causar daños de consideración. La disponibilidad, por otra parte, es una característica de diseño básica para aplicaciones donde el tiempo de duración de un fallo significa pérdidas cuantitativas considerables.

Existen otros parámetros mediante los cuales se puede medir la capacidad de un sistema para tolerar fallas, por ejemplo: la seguridad ("safety"), el rendimiento ("performance"), la dependencia ("dependability"), el mantenimiento ("maintainability") y la evaluación ("testability").

3.3.3 Medidas principales

El propósito de los métodos de evaluación cuantitativa es calificar numéricamente los atributos, como confiabilidad y disponibilidad. de un sistema con el fin de contar con una base para su comparación con otros sistemas [Joh89].

Algunas de las medidas de evaluación cuantitativas principales son: tasa de experimentación de fallos ("failure rate"), tiempo promedio de aparición de un fallo "mean time to failure" ("MTTF"), tiempo promedio de recuperación "mean time to repair" ("MTTR") y tiempo promedio entre fallos "mean time between failure" ("MTBF").

"Failure Rate" y confiabilidad

La tasa de experimentación de fallos es el número esperado de fallos que sufre un dispositivo o sistema durante un lapso de tiempo definido. De esta forma, un sistema que sufre, en promedio, un fallo cada mil horas posee una tasa de experimentación de fallos de $1/1000\ fallos/hora$.

La relación existente entre la *confiabilidad* de un sistema $R(t)$ y el tiempo correspondiente a su *período de vida útil* obedece normalmente a la denominada **ley exponencial para la aparición de fallos** ("exponential failure law"). Esta ley establece que si la tasa de experimentación de fallos es igual a λ, la confiabilidad del sistema varía de manera exponencial en función del tiempo ($R(t) = e^{-\lambda t}$) [Joh89].

De esta forma, en el momento de iniciarse el período de vida útil del sistema, o $t = 0$, la confiabilidad del mismo $R(t = 0)$ es igual a 1 o del 100%. Sin embargo, la confiabilidad del sistema para un lapso de tiempo infinito, $t = \infty$, es igual a 0 o del 0%.

"MTTF"

El tiempo promedio de aparición de un fallo ("MTTF") representa un lapso de tiempo estimado durante el cual un sistema opera normalmente o sin sufrir fallos y que termina en el momento de presentarse el *primer* fallo del sistema.

Si t_i representa el lapso de tiempo en que cada sistema, de un conjunto de N sistemas idénticos, experimenta su primer fallo, el tiempo promedio de aparición de un fallo es igual a

$$MTTF = \frac{\sum_{i=1}^{N} t_i}{N} \tag{3.1}$$

De acuerdo con [Joh89], el tiempo promedio de aparición de un fallo se puede obtener a partir del la confiabilidad $R(t)$ del mismo sistema y conforme a la siguiente ecuación

$$MTTF = \int_0^\infty R(t)dt \tag{3.2}$$

con el único supuesto de que la función para la confiabilidad $R(t)$ cumple con $R(\infty) = 0$.

Si $R(t)$ cumple con la **ley exponencial para la aparición de fallos**, la función anterior se convierte en

$$MTTF = \int_0^\infty e^{-\lambda t}dt = \frac{1}{\lambda} \tag{3.3}$$

En conclusión, el tiempo *promedio de aparición de un fallo* (MTTF) para un sistema que sigue la **ley exponencial para la aparición de fallos** es el inverso de su **tasa de aparición de fallos** λ.

"MTTR"

El tiempo promedio de recuperación de un sistema es el tiempo estimado necesario para volverlo nuevamente operacional, o de reparación. El cálculo de este parámetro es difícil de determinar ya que depende de un número considerable de factores, como el número de elementos tomados en cuenta y

de los tipos de fallas abordadas, etc. La estimación del "MTTR" se realiza generalmente de manera experimental e involucra, por ejemplo, el tiempo promedio empleado en la reparación de un sistema después de experimentar individualmente todos los N elementos de un conjunto de fallas distintas. De esta forma, si a cada una de las N fallas le sigue un tiempo de recuperación t_i, el "MTTR" del sistema es igual a:

$$MTTR = \frac{\sum_{i=1}^{N} t_i}{N} \tag{3.4}$$

Si las N fallas pueden ser reparadas por cada miembro de un grupo de M individuos, cada uno de éstos promedia un "MTTR" distinto. Otra forma de calcular entonces el "MTTR" del sistema es obteniendo el promedio de los "$MTTR_i$" correspondientes a cada miembro i:

$$MTTR = \frac{\sum_{i=1}^{M} MTTR_i}{M} \tag{3.5}$$

La **velocidad de reparación o la tasa de experimentación de reparaciones** μ ("repair rate") de un sistema representa el número promedio de reparaciones que éste es capaz de realizar en un período de tiempo. Esta última medida sirve también para especificar el "MTTR" de un sistema y éstos se relacionan de la siguiente forma:

$$MTTR = \frac{1}{\mu} \tag{3.6}$$

En este sentido, el "MTTR" se mide en unidades de tiempo por acción de reparación; mientras que en la práctica, las de μ son reparaciones por hora.

"MTBF"

El **tiempo promedio entre fallos** ("MTBF") de un sistema representa el intervalo de tiempo estimado que transcurre *entre* la aparición de dos fallos consecutivos en el mismo.

El "MTBF" se calcula también mediante un promedio de lapsos de tiempo. Cada uno de estos lapsos incluye un intervalo de tiempo en modo operacional y otro de recuperación necesario para traer de vuelta al sistema a

un nuevo modo operacional.

Considérese un lapso de tiempo T durante el cual N sistemas idénticos son puestos en funcionamiento y sufren potencialmente de fallos, los cuales son seguidos inmediatamente de acciones de recuperación del sistema. Si el número de fallos que experimenta individualmente cada sistema, durante T, se define como n_i, el promedio de experimentación de fallos del sistema es igual a

$$n_{prom} = \sum_{i=1}^{N} \frac{n_i}{N} \tag{3.7}$$

De esta forma, el **tiempo promedio entre fallos** del sistema se define entonces como la razón entre el lapso de tiempo T y el promedio de experimentación de fallos, n_{prom}:

$$MTBF = \frac{T}{n_{prom}} \tag{3.8}$$

Si las acciones de recuperación de un sistema lo llevan al mismo modo de operación que tenía antes de sufrir el fallo, el **tiempo promedio entre fallos** "MTBF" mantiene la siguiente relación con los parámetros "MTTF" y "MTTR" del mismo sistema:

$$MTBF = MTTF + MTTR \tag{3.9}$$

En muchas ocasiones, los términos "MTBF" y "MTTF" son utilizados indistintamente; sin embargo, su diferencia es también en ocasiones de importancia conceptual y matemática.

3.4 Clasificación de los fallos

Los **fallos** pueden clasificarse de acuerdo al elemento que los sufre y a la forma en que se manifiestan. Flaviu Cristian presenta en [Cri91] una clasificación desde el punto de vista de las fallas potenciales de un servidor. Esta taxonomía asume una especificación donde se definen la acción[3] de un servidor a consecuencia de cualquier petición ("input") y un intervalo de tiempo

[3]Cambio en el estado interno de un servicio o respuesta.

real dentro del cual dicha acción debe ocurrir[4]. Hadzilacos y Toueg [HT94] proponen otra taxonomía donde se distinguen los fallos de un procesador, los de un canal de comunicación y los dependientes del tiempo en un sistema síncrono.

La Tabla 3.1 presenta una taxonomía básica de fallos con base en su ubicación y comportamiento (ver figura 3.2). Los fallos incluidos en esta tabla se aplican tanto a sistemas síncronos, como asíncronos[5].

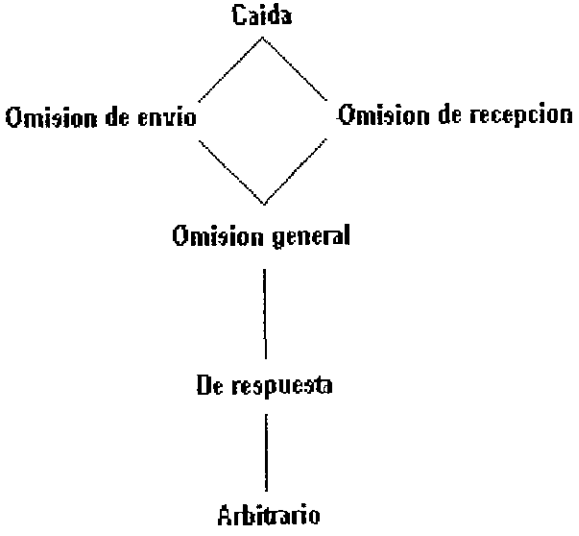

Figura 3.2: Diagrama general de clasificación de fallos.

La caída ("crash") y el reinicio de un servidor dan lugar una subclasificación de este tipo de fallo. Esta nueva subdivisión se realiza de acuerdo al nuevo estado del servicio, implementado por el servidor. La tabla 3.3 presenta esta nueva taxonomía. Nótese como un servidor puede implementar también un servicio de comunicación.

Cristian clasifica las caídas de servidores sin estado ("stateless servers")[6]

[4]Es decir, en un sistema síncrono

[5]Un sistema asíncrono no asume ninguna suposición acerca de la velocidad de ejecución o el retraso en el envío de un mensaje [Mul93]

[6]Un servidor sin estado es un proceso, o aplicación, cuyo estado actual es independiente

Fallo	Ubicación	Descripción
Omisión ("Omission")	Servidor	El servidor no genera la respuesta ("output") o el cambio de estado correspondiente a una petición ("input").
Omisión de envío ("Send-omission")	Procesador	El mensaje generado por un procesador no alcanza el canal de comunicación o lo hace de manera incompleta.
Omisión ("Omission")	Canal de Comunicación.	El canal de comunicación experimenta un fallo al omitir la transmisión de un mensaje colocado en el mismo.
Omisión de Recepción ("Receive-Omission")	Procesador	El procesador experimenta un fallo al no recibir ("receive") un mensaje o recibirlo de manera incompleta.
Omisión General ("General Omission")	Procesador	El procesador es susceptible de sufrir fallos tanto de omisión de recepción, como de envío.
Fallo de valor ("value failure")	Servidor	El servidor genera una respuesta ("output") incorrecta a una petición ("input").
	Canal de Comunicación	El canal de comunicación corrompe los mensajes que transmite.
Fallo de transición ("state transition failure")	Servidor	El servidor produce un error en el estado interno del servicio en respuesta a una petición ("input").
Fallo de respuesta ("response failure")	Servidor	Los fallos de respuesta de un servidor abarcan los del tipo de valor y transición.
Caída ("Crash")	Servidor	El servidor, después de un primer fallo de omisión, deja de producir respuestas ("outputs") o cambios de estado a peticiones ("inputs") subsecuentes. La reanudación del servicio es sólo posible mediante su reinicialización.

Tabla 3.1: Taxonomía básica de fallos.

y las del tipo *con pausa* y *total* como subclases del fallo de omisión de un servidor.

Otra clasificación distingue entre fallos *transitorios* ("transient failures"), *intermitentes* ("intermitent failures") y *permanentes* ("permanent

de sus estados anteriores [BM95]. Es decir, su estado actual no guarda relación con las interacciones llevadas a cabo con sus clientes en el pasado.

Fallo	Ubicación	Descripción
	Procesador	El procesador experimenta un fallo que le hace dejar de funcionar prematuramente y no realiza ninguna operación en adelante. Momentos antes del fallo, el procesador funciona correctamente.
	Canal de Comunicación	El canal de comunicación deja transportar mensajes. Sin embargo, en el instante anterior al momento en que deja de transmitir, el canal se comporta correctamente.
Fallo Arbitrario ("Byzantine failure")	Servidor o Procesador	Un servidor (procesador) que experimenta un fallo arbitrario continua operando, genera respuestas incorrectas y, posiblemente, trabaja maliciosamente en conjunto con otros servidores (procesadores) con fallos. En general, exhibe un comportamiento arbitrario. Los defectos en el software ("bugs"), por ejemplo, ocasionan comúnmente fallos arbitrarios [TR85].
Fallo Arbitrario ("Byzantine failure")	Canal de Comunicación	El canal de comunicación continúa funcionando de manera completamente arbitraria. Un canal de comunicación que genera mensajes por si mismo ("spurious messages") experimenta un fallo arbitrario.

Tabla 3.2: Taxonomía básica de fallos, continuación.

failures") [TR85]. Los *transitorios* ocurren una vez y desaparecen en una segunda ejecución, situación común en sistemas concurrentes. Los *intermitentes* aparecen y desaparecen de manera arbitraria; mientras que las *permanentes* existen hasta el momento en que el elemento es reparado de su falla.

Un sistema síncrono distribuido se caracteriza por un límite superior en el intervalo de tiempo real durante el cual deben realizarse la transferencia de un mensaje, la calendarización ("scheduling") de un proceso y la revisión de un mensaje. Este tipo de sistemas cuenta con un número de fallos particulares. La tabla 3.4 define los fallos relacionados con un sistema síncrono.

Fallo	Ubicación	Descripción del nuevo estado
Caída con Amnesia ("Amnesia-crash")	Servidor	El servicio reinicia en un estado predeterminado ("default") e independiente de las peticiones ("inputs") anteriores a la caída.
Caída con Amnesia Parcial ("Partial-amnesia-crash")	Servidor	Una parte del estado es igual al anterior de la caída y el resto se define de igual forma que en el caso de una caída con amnesia.
Caída con Pausa ("Pause-crash")	Servidor	El estado del servicio es igual al anterior a su caída.
Caída Total ("Halting-crash")	Servidor	El servidor nunca reinicia.

Tabla 3.3: Subclasificación del fallo tipo *caída*.

La taxonomía presentada en este trabajo está basada en las clasificaciones, anteriormente mencionadas, de Cristian, Hadzilacos y Toueg.

El hecho de que un fallo del tipo caída pueda detectarse, o no, da lugar a dos clases de sistema:

Sistemas de Fallo Silencioso ("Fail-Silent Systems"). Un sistema en esta categoría funciona de acuerdo a sus especificaciones o deja completamente de operar. Los fallos de caída y omisión en estos sistemas son imposibles de distinguir [KP93].

Sistemas de Fallo de Terminación ("Fail-Stop Systems"). Estos sistemas se caracterizan por la forma en que experimentan un fallo del tipo caída:

> en el momento que pierden su capacidad para continuar funcionando correctamente y de acuerdo a una especificación, estos sistemas cambian primero a un estado donde otros sistemas pueden detectar su caída ("crash") y sólo entonces cesan de ejecutar operaciones [Sch84].

Este propiedad puede implementarse, por ejemplo, incluyendo la transmisión de una señal periódica en las especificaciones de un sistema.

En los sistemas asíncronos es imposible distinguir entre un computación que se realiza de manera muy lenta y la caída de un proceso. De igual forma, la distinción entre un fallo de omisión y uno de caída es también difícil.

Fallo dependiente del tiempo	Ubicación	Descripción
Fallo de Reloj	Procesador	El reloj de un proceso excede el límite de desviación respecto al tiempo real definido en su especificación.
Fallo de Rendimiento ("Perfomance")	Canal de Comunicación	La transmisión de un mensaje ocurre en un lapso de tiempo real mayor al especificado en un sistema síncrono.
Fallo de Rendimiento ("Performance")	Procesador	El tiempo de ejecución entre dos pasos de una computación excede el intervalo de tiempo real especificado en un sistema síncrono.
Fallo de Respuesta Temprana ("Early timing failure")	Servidor	La respuesta o transición llevada a cabo por un servidor es correcta, pero ocurre en un instante anterior al intervalo de tiempo real especificado en un sistema síncrono.
Fallo de Respuesta Tardía ("Late timing failure")	Servidor	La respuesta o transición llevada a cabo por un servidor es correcta, pero ocurre en un instante posterior al intervalo de tiempo real especificado en un sistema síncrono.
Fallo de Rendimiento ("Performance")	Servidor	La respuesta o transición llevada a cabo por un servidor es correcta, pero ocurre en un instante fuera del intervalo de tiempo real especificado en un sistema síncrono. Incluye los fallos de respuesta tardía y temprana.

Tabla 3.4: Tipo de fallos relacionados con un sistema síncrono.

El reconocimiento de estas diferencias es importante para lograr una característica a tolerancia a fallas y obliga a que se introduzcan medidas para transformar un sistema del tipo "Fail-Silent" en un del tipo "Fail-Stop".

En los sistemas síncronos, por otra parte, los modelos "Fail-Silent" y "Fail-Stop" son equivalentes.

Los fallos posibles de un servidor tienen un impacto considerable en la arquitectura de un sistema. La mayoría de los diseños consideran servidores del tipo "fail-stop". En el otro extremo, la consideración pesimista de que un sistema pueda exhibir fallos del tipo arbitrario ("byzantine failures") es únicamente necesaria en algunos casos -sistemas de control de vuelo, por ejemplo. El conjunto de fallos incluido en la especificación de un sistema

excluye normalmente a los fallos del tipo arbitrario. En primer lugar, el tratamiento fallos arbitrarios lleva a sistemas más lentos y con una mayor redundancia. Además, la probabilidad de que un proceso encargado de la detección de una falla trabaje maliciosamente es muy baja en la práctica [BJ].

3.5 Semántica de fallos

En la tolerancia a fallas, la programación necesita considerar siempre una semántica de operación normal y otra para el caso de la aparición de fallos [Lam78a]. Un requisito indispensable para cumplir con este objetivo es contar con el número y definición concreta del tipo de fallos considerados en el diseño del sistema. Este conjunto queda normalmente definido, entre el conjunto de fallos potenciales, mediante un análisis estocástico y de los requerimientos [Cri91].

Un servidor s posee una semántica de fallos ("failure semantics") F si su especificación establece que los fallos que puede sufrir, y que pueden detectar sus usuarios, se encuentran en una clase F [Cri91]. En este contexto, la clase F consiste simplemente en un conjunto determinado de fallos. La semántica de fallos, en otras palabras, determina las formas en las cuales puede detectarse que un servidor deja de funcionar correctamente.

La especificación de un servidor s puede permitirle exhibir fallos en la unión de 2 o mas clases. Un servidor s cuenta con una semántica de fallas más *débil* que otro servidor w si el número de clases consideradas para s es mayor que el considerado para w. El servidor w, a su vez, posee una semántica de fallos más *fuerte* en comparación con s. Una *semántica arbitraria de fallos* ("arbitrary failure semantics") contempla una clase formada por todos los tipos de fallos.

Los diseñadores son los responsables de crear el sistema con la semántica de fallos definida en los requerimientos. La consideración de una semántica de fallos permite la *detección* de los mismos y constituye la base para su posterior tratamiento –*enmascaramiento*, por ejemplo.

La semántica de fallos de un sistema incluye el conjunto de fallos que éste mismo puede experimentar, pero no hace referencia a los de sus componentes o subsistemas de más bajo nivel que emplea.

3.6 Técnicas principales de tolerancia a fallas

Dos son las técnicas principales utilizadas para implementar una filosofía de tolerancia a fallas en un sistema (ver figura 3.3):

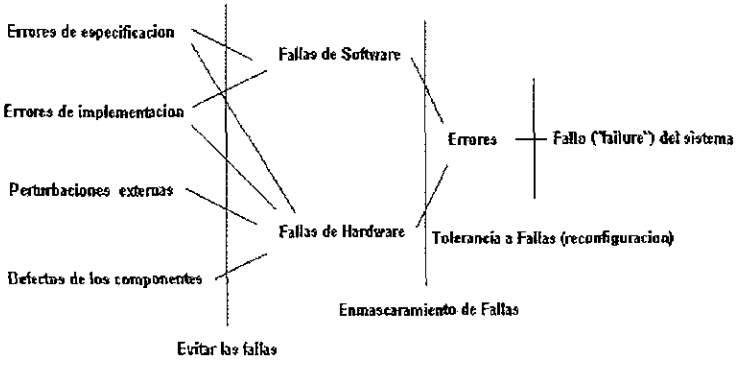

Figura 3.3: Técnicas principales para el manejo de fallas.

1. La reconfiguración del sistema.
2. El enmascaramiento de fallas.

Ambos enfoques son explicados a continuación.

3.6.1 Reconfiguración

La reconfiguración como medida de tolerancia a fallas es el proceso de eliminar a los elementos con fallas de un sistema y devolver a éste último a un estado o condición operacional –o sin errores.

El diseño de una política de este tipo involucra cuatro pasos secuenciales fundamentales:

1. Detección de las fallas/errores ("Fault/Error detection").
2. Ubicación de las fallas/errores ("Fault/Error location").
3. Confinamiento de las fallas/errores ("Fault/Error containment").
4. Recuperación de las fallas/errores ("Fault/Error recovery").

La razón de que las fallas sean el origen de los errores es el motivo de que en la literatura se utilicen indistintamente ambos términos. En este trabajo, las referencias a estas fases se harán en adelante con respecto a los errores.

Detección de errores

Detección de errores es el proceso de reconocer que un error se ha producido y sus resultados determinan estrictamente la política de tolerancia a fallas a seguir.

Un mecanismo de detección de errores ideal consiste en uno capaz de detectar todos los errores posibles (*completo*), que es confiable (*correct*) e independiente del sistema y, en su caso, revisa el resultado de todas las transiciones en el estado interno y externo de un servicio. En la práctica, la búsqueda de un error depende en gran medida del tipo de fallo que se busca prevenir y de las características del sistema, lo cual da lugar a diferentes tipos de mecanismos de revisión ("acceptance checks"):

- Revisión de réplicas ("Replication checks"). El uso de elementos redundantes permite la comparación de resultados y el empleo de votaciones para detectar la experimentación de fallos arbitrarios en los primeros.

- Revisión de tiempo ("Timing checks"). El uso de limites de tiempo ("time-outs") permite la detección de errores en sistemas distribuidos síncronos (fallos de rendimiento) y la *suposición* de su existencia en los del tipo asíncrono (caídas).

- Revisión de información ("Structural and coding checks"). El empleo de información redundante, como en los códigos de detección de errores, es un ejemplo de esta clase de revisión.

- Revisión de estado ("Reasonableness checks"). Los mecanismos de revisión en tiempo de ejecución ("run-time checks") que utiliza un componente o servidor para comprobar el estado interno del servicio, por ejemplo.

- Revisión de elementos ("Diagnostics checks"). La invocación de procedimientos de prueba, con salidas bien definidas para ciertas entradas, permiten corroborar la presencia de errores en un componente, o servidor, de más bajo nivel.

Ubicación de errores

La ubicación de errores es el proceso de determinar donde se encuentra la falla que produce un error, así como el alcance de este último.

Si el proceso de detección de errores no se realiza de manera continua o adecuada, un error es posible que se propague a otras partes de un sistema. Entonces, una vez detectado un error es necesario definir exactamente sus límites dentro del sistema.

Los límites de un error pueden determinarse *dinámicamente* siguiendo el flujo de información desde la fuente del mismo a otros componentes del sistema. Otra forma es definir sus límites de manera *estadística* y establecer areas con una gran probabilidad de contener el error, una vez detectado el mismo.

En la práctica, este proceso no se realiza explícitamente sino de forma indirecta utilizando alarmas de tiempo o pruebas específicas y en base a las características del sistema. De manera concreta, los posibles resultados de este proceso son sustituidos por la política de *confinamiento de errores* establecida en las especificaciones del sistema tolerante a fallas.

Confinamiento de errores

La etapa de confinamiento de errores es el proceso de aislar una falla y prevenir la propagación del error causado por la misma a través del sistema. En este sentido, *confinamiento* no implica una restauración del fallo, ni tampoco la reanudación del servicio o las funciones del sistema.

El objetivo principal de esta fase es la obtención del último estado correcto perteneciente al sistema. Dos son las técnicas principales que existen para conseguir un estado libre de errores: *backward recovery* y *forward recovery*. A continuación se proporciona su definición, de acuerdo con [BV]:

1. *Backward Recovery.* Considérese un conjunto de procesos $P_1, \ldots, P_n$, los cuales realizan en conjunto una computación A en el instante que P_s sufre un fallo. Como requisito para su participación en A, cada proceso $P_i = 1, 2, ..., n$ debe guardar su estado actual en *almacenamiento estable* ("checkpointing") antes de cooperar con los demás procesos. Una vez que se presenta el fallo en P_s, el resto de los procesos debe

decidir si completar (A) o recobrar de almacenamiento estable el último estado de P_s ("rollback"). En el último caso, la computación A puede volverse a ejecutar desde el inicio.

2. *Forward Recovery.* Esta política cuenta con dos principios básicos:

   - La adopción de uno más procesos –P_1, ..., P_n– con copia del estado interno de un proceso crítico P. Si el conjunto de copias C(P_1, ..., P_n) es el encargado del estado de P, C(P_1, ..., P_n) puede utilizarse para recobrar el estado de P cuando éste último sufra un fallo.
   - Si el fallo en P no es detectado por C(P_1, ..., P_n), las acciones de recuperación en P pueden extenderse a P_1, ..., P_n. El costo, obviamente, de la recuperación es más alto si el fallo se extiende a otros procesos.

La recuperación de un estado sin errores es importante porque permite al sistema contar con el último estado correcto de una computación y, a partir de éste, continuar con el proceso de *recuperación* de un error.

Recuperación de errores

El proceso de *recuperación de errores* consiste en mantener, o alcanzar de nuevo, un status operacional a pesar de la presencia de fallas y después de la ocurrencia de un error.

Hasta esta fase, el error ha sido detectado, ubicadas sus fronteras y confinado; sin embargo, la fuente del mismo permanece en el sistema. De esta forma, el objetivo de esta etapa es tomar acciones para eliminar o sustituir al componente que alberga a la falla y reiniciar total o parcialmente las funciones del sistema.

En ocasiones, la *detección* y *ubicación* de un error pueden identificar plenamente al componente dueño de la falla; sin embargo, este no es siempre el caso. De esta forma, la *recuperación de errores* consisten en identificar primeramente al componente dueño de la falla y, a continuación, en la reparación del sistema.

La reparación de un sistema se refiere concretamente a las acciones correctivas sobre un componente defectuoso[7]. De manera concreta, dichas

[7] Con una o mas fallas.

acciones pueden significar

1. La reparación de la falla en el componente,
2. La sustitución del mismo,
3. El uso del componente de una manera indistinta tomando en cuenta una disminución en sus capacidades de funcionamiento.
4. Su eliminación por completo del sistema.

En particular, la sustitución de un componente depende del tipo de funciones del mismo. Es decir, si sus funciones son las de un servidor con estado, las abstracciones de datos del nuevo componente pueden necesitar potencialmente de una actualización. En el caso de un servidor sin estado, el uso de un mismo componente sin fallas es suficiente.

Además, la reparación de un sistema debe realizarse de manera automática.

Por último, el empleo de una técnica de *reconfiguración* como medida de tolerancia a fallas debe mantener siempre a un sistema disponible a sus usuarios. De esta forma, el procedimiento completo de *reconfiguración* significa solamente una pérdida en el rendimiento ("performance") de las operaciones en curso y de un retraso en la atención a las nuevas.

3.6.2 Enmascaramiento de fallas

El *enmascaramiento de fallas* ("fault masking") es otra forma de implementar una técnica de tolerancia a fallas.

Un proceso encargado de llevar a cabo esta técnica intenta evitar que una falla de software o hardware introduzca un error en el estado interno de un servicio, o sistema. Uno de los ejemplos más representativo de esta técnica es el empleo de votaciones. Considérese un conjunto A de tres módulos digitales encargados de producir un valor booleano a partir de una misma función y los mismos elementos del dominio, o valores de entrada. Cualesquiera dos votos que concuerden en el mismo valor determinan la salida del conjunto y permiten enmascarar la salida del módulo en desacuerdo con la mayoría. Si el estado interno de un servicio depende del resultado alcanzado por el conjunto A, el algoritmo de votación evita que la falla en uno de los

módulos pueda introducir errores en el mismo.

En los sistemas distribuidos, dos son los enfoques fundamentales para enmascarar fallas:

1. Enmascaramiento jerárquico de fallos.
2. Enmascaramiento de fallos utilizando grupos.

Ambos se refieren al enmascaramiento de fallos y no de fallas. La razón de esto es que en los sistemas distribuidos, por lo general, la detección de un fallo en uno de los nodos corresponde unívocamente a la existencia de un falla en el mismo. En otras palabras, la probabilidad de encontrar un error que involucre varios nodos es muy baja.

Enmascaramiento jerárquico

Los servidores se ubican normalmente dentro de una relación **jerárquica**[8], donde cada servidor pertenece a un determinado nivel de abstracción. De esta forma. la existencia de un tipo de fallo, en cierto nivel, puede ocasionar la aparición de otro fallo en un nivel más alto.

Sean B y C dos niveles de abstracción. Defínase también un servidor **s** en B, el cual depende de un servicio implementado por otro servidor **t** en C. El servidor **s** **enmascara** los fallos de **t** si puede continuar proporcionando su servicio a pesar de la presencia de fallos en **t**. El servidor **s** puede intentar una nueva petición o comunicarse con otro servidor que ofrezca el mismo servicio que **t**, por ejemplo. A este tipo de *enmascaramiento* se le denomina **enmascaramiento jerárquico.**

En la práctica, la mayoría de los servidores se diseñan con una semántica de fallos *fuerte* que permita detectarlos de manera más sencilla. En el otro extremo. la detección de un fallo en un servidor con una *semántica arbitraria de fallos* obliga a sus clientes a contar con un medio de comparación de resultados. lo cual implica una mayor complejidad y tiempo de procesamiento.

Además de la programación de las acciones de enmascaramiento, un servidor debe programarse con un procedimiento para recobrar su último estado consistente – estable– en caso de sufrir un fallo. Como condición,

[8]Ver sección 3.2.1 en página 25.

éste estado debe quedar listo antes de la aparición de la excepción ("exception") correspondiente en el nivel superior de abstracción[9].

El recobro del último estado consistente no significa la desaparición del fallo, sino que sirve únicamente como garantía de que el servidor no dejará a los niveles inferiores con un estado inconsistente. Desde el punto de vista de un cliente, la programación de sus acciones de construcción de un estado estable y de enmascaramiento se facilitan cuando éste puede estar seguro de que los servidores que emplea y que sufren un fallo no dejan con un estado inconsistente al resto de los servidores en los niveles inferiores de abstracción [Cri91].

El enmascaramiento jerárquico puede utilizar tanto una política de *backward recovery*, como de *forward recovery*, para construir un estado estable.

Enmascaramiento de fallos utilizando grupos

Otra forma de enmascarar un fallo consiste en implementar un servicio utilizando un grupo **redundante** y físicamente independiente de servidores. La redundancia en hardware y software permite contar con un servicio ininterrumpido, aún cuando alguno de los servidores experimente un fallo.

Un grupo **G** de servidores *enmascara* el fallo de uno de sus miembros **m** cuando el resto del grupo, en conjunto, continúa respondiendo correctamente a las peticiones ("inputs") de sus clientes.

De acuerdo a la forma en que los servidores de un grupo se coordinan, éstos últimos pueden clasificarse como fuertemente sincronizados ("closely synchronized group") o débilmente sincronizados ("loosely coupled synchronized group").

En un grupo *fuertemente sincronizado*, las llamadas de cada cliente son atendidas una sola vez por cada servidor y el servicio continúa mientras exista el mínimo número necesario de servidores, dependiendo de sus semánticas de fallos. Este tipo de sincronización es empleada comúnmente cuando los requerimientos establecen respuestas en tiempo real ("real-time response") o los servidores poseen una semántica arbitraria de fallos.

[9] En este caso, se supone que el servidor cuenta con una semántica de fallas del tipo *terminación* ("fail-stop"). Ver sección 3.4 en página 37

Un grupo débilmente sincronizado, por otra parte, trabaja básicamente con un solo servidor, denominado primario, y utiliza un conjunto de uno o mas servidores como respaldo ("backup servers" o "stand-by servers"). El servidor primario se encarga de atender las peticiones de los clientes y de generar las respuestas correspondientes. Durante la ejecución normal de un servicio, los servidores de respaldo reciben periódicamente una copia, o actualización, del último estado consistente del servidor primario ("check-pointing") y una bitácora ("log") con las operaciones en proceso de éste último. En el momento de presentarse un fallo en el servidor primario, uno de los servidores de respaldo ejecuta las operaciones de la última bitácora recibida y asume el lugar del servidor primario.

Un grupo débilmente acoplado cuenta con la ventaja de utilizar un menor número de recursos al atender una petición ("overhead"), lo cual puede traducirse en un menor tiempo de respuesta[10]. Un mayor tiempo de recuperación ("recovery") representa su mayor desventaja y vuelve a éste esquema no apropiado para aplicaciones con requerimientos de tiempo real. Además, no permite detectar fallas bizantinas por proporcionar únicamente una salida. De esta forma, un grupo débilmente acoplado es ideal para un servicio con una semántica de fallos que incluya fallas de terminación ("fail-stop") y de rendimiento ("performance").

A diferencia del enmascaramiento jerárquico, en este caso no es necesario el manejo de excepciones entre un cliente y un servidor para enmascarar un fallo. En el manejo de grupos, existe comúnmente un mecanismo encargado del enmascaramiento de los fallos y de la administración[11] de los servidores, el cual es transparente para los usuarios del servicio.

El grupo que implementa un servicio puede también contar con una semántica de fallos. Un grupo **G** posee una semántica de fallos F si su especificación establece que los fallos que puede sufrir, y que pueden detectar sus usuarios, se encuentran en una clase F[12].

Un grupo capaz de enmascarar un número de k fallos concurrentes entre sus miembros se denomina k-Tolerante a fallos[13] ("k-fault tolerant"). Un grupo puede diseñarse para tolerar desde el fallo de uno solo de sus miembros

[10] Sin embargo, esta aseveración depende del tipo de aplicación.

[11] Es decir, encargado de la adición y eliminación de servidores, la comunicación, etc.

[12] Ver sección 3.5 en página 39.

[13] El número k, en [Bab], recibe el nombre de "resiliency".

("single-fault tolerant") hasta varios ("multiple-fault tolerant"). El número k depende en gran medida de la semántica de fallos de cada elemento redundante. Si los miembros de un grupo pueden experimentar fallas bizantinas o fallos de respuesta, el grupo requiere de $2k+1$ miembros[14], como mínimo, para enmascarar un solo fallo [Gra]. En otro caso, si cada miembro pueden experimentar fallos del tipo *omisión* o *caída*, el número de miembros (n) necesita ser solamente más grande que k ($n>k$) [Bab].

Muchos sistemas, en la práctica, utilizan una técnica de enmascaramiento formada por una combinación del modelo jerárquico y de grupos.

Comparación entre el modelo jerárquico y de grupos

La diferencia principal entre ambos modelos son los supuestos que se hacen acerca de los sistemas de más bajo nivel que residen debajo de un sistema con fallos.

El modelo jerárquico enmascara eficazmente los fallos de un servidor s, siempre y cuando los niveles inferiores de abstracción por debajo de s no sufran ninguno fallo o guarden algún error. De esta forma, la recuperación de un estado consistente se vuelve muy importante en este modelo.

El enmascaramiento utilizando grupos, en cambio, es capaz de enmascarar el fallo de un servidor, aún cuando esta situación se deba a la presencia de errores o fallos en otros servidores en los niveles inferiores de abstracción.

3.6.3 Técnica híbrida

Las técnicas híbridas buscan combinar las ventajas del enmascaramiento de fallas y la reconfiguración de un sistema.

En este esquema, el enmascaramiento es utilizado para prevenir la fabricación de respuestas erróneas; mientras que la reconfiguración es empleada para remover los elementos con fallas y sustituirlos por algún tipo de reemplazo.

Por otra parte, la implementación de una política híbrida es generalmente más costosa que la de una sola de las originales.

[14]Este problema es conocido, en inglés, como " Byzantine Agreement Problem".

3.7 Sistemas distribuidos y tolerancia a fallas

El principio básico de la tolerancia a fallas se llama *redundancia* [SV97]. Los primeros trabajos en este tema se enfocaban a la redundancia de componentes físicos; sin embargo, hoy en día existen nuevas y variadas formas [Joh89]:

1. **Redundancia de Hardware** signfica la adición de componentes físicos ("hardware"). Su propósito es, por lo general, la tolerancia o la detección de fallas.

2. **Redundancia de Software** implica la adición de programas de "software" al conjunto mínimo y necesario para realizar una función. Su propósito es detectar y, de ser posible, tolerar fallas.

3. **Redundancia de Información** significa la suma de información adicional a la necesaria para llevar a cabo una función. Los códigos de detección de errores, por ejemplo, utilizan una forma de redundancia de información.

4. **Redundancia de Tiempo** implica la adición de tiempo en la realización de una función. Esta mayor cantidad de tiempo puede ser utilizada en la ejecución de mecanismos de detección de errores o de tolerancia a fallas en sistemas con fallos transitorios, por ejemplo.

La redundancia en hardware y software se denominan también **redundancia de recursos o física.** En este sentido, la redundancia de recursos se divide también en **redundancia pasiva y redundancia activa.** Un ejemplo de redundancia pasiva es la utilizada en un grupo débilmente acoplado; mientras que uno de redundancia activa es el propio de un grupo fuertemente acoplado.

El manejo de excepciones es considerado también una clase de **redundancia pasiva**[KP93]. En este sentido, la detección de una excepción provoca que el control sea transferido a un manejador especial encargado de la terminación del programa respectivo o la sustitución de éste último por uno sin errores.

Un caso particular de *redundancia pasiva de tiempo* es el de una recuperación del tipo "rollback", donde una computación vuelve a comenzar desde un estado sin errores almacenado con antelación ("checkpointing").

La relación entre los sistemas distribuidos y la tolerancia a fallas se da en ambos sentidos [Sch93]. En primer lugar, todos los métodos de tolerancia a fallas utilizan alguna forma de redundancia mediante el uso de elementos que realizan una misma función y cuyos orígenes posibles de fallos son totalmente independientes entre si. De esta forma, el problema de un arquitectura tolerante a fallas consiste principalmente en manejar dicha redundancia de componentes independientes.

Una de las características de los sistemas distribuidos es el empleo de protocolos para la comunicación y sincronización entre elementos físicamente independientes entre si. En este sentido, muchos de los problemas de esta clase de sistemas encuentran aplicación en los de esquemas de tolerancia a fallas.

En segundo lugar, la probabilidad de presentarse **al menos** un fallo en un sistema distribuido aumenta significativamente con el número de sus elementos constituyentes. La necesidad, entonces, de contar con una política de tolerancia a fallas es gran importancia en los sistemas distribuidos. La confiabilidad de los protocolos distribuidos depende en gran medida de la adopción de dicha medida.

La conclusión de esta correspondencia es que ambos temas de estudio se complementan y se benefician uno del otro.

La dificultad de construir entonces un sistema distribuido tolerante a fallas abarca diferentes problemas, los cuales se suman a los encontrados tradicionalmente en los sistemas distribuidos. A continuación se presentan algunos de los más importantes:

- Consenso ("Agreement"). En términos generales, el problema del consenso involucra una decisión concensada entre un número de procesos, susceptibles a experimentar un fallo, para llevar a cabo o no una acción determinada [Had].

  M.J. Fischer, N.A. Lynch y M.S. Paterson han demostrado que no es posible, en un sistema asíncrono, resolver el problema del consenso entre procesos, aún cuando solamente uno experimente un fallo del tipo *caída* ("crash"). Este resultado ha propiciado la definición de un conjunto mínimo de propiedades mediante las cuales puede resolverse el problema en un sistema asíncrono y susceptible a fallos [Ray].

El concepto de detectores no confiables de fallos ("unreliable failure detectors"), propuesto por Chandra y Toueg [CT96], es una de las contribuciones teóricas más importantes en este sentido.

- Detección de Fallos ("Failure Detection"). Los sistemas distribuidos asíncronos reales no pueden establecer un límite de tiempo (δ) al retardo ("delay") en el inicio de un proceso("scheduling"), *ni* tampoco al tiempo de transferencia y procesamiento de un mensaje [Cri91]. Esta afirmación hace muy difícil distinguir entre un fallo en el canal de comunicación o de un proceso y un retardo en la transmisión de un mensaje o la ejecución de un programa.

  Chandra y Toueg proponen, en [CT96], incluir en el sistema la noción de detectores de fallos ("failure detector"). Estos detectores, por ejemplo, pueden construirse haciendo suposiciones acerca de los tiempo de respuesta de un proceso. Los detectores se ubican en cada nodo y dan pistas ("hints") de los sitios que probablemente experimenten fallos; sin embargo, no son confiables ("unreliable") por que pueden hacer suposiciones incorrectas acerca de participantes correctos o con fallos.

  Chandra y Toueg han definido ocho clases de detectores no confiables ("unreliable failure detectors") con los cuales se puede resolver, en teoría, el problema del consenso. La clasificación se basa en las siguientes propiedades:

  "Strong (Weak) Completeness" : Todos los participantes con fallos se vuelven, eventualmente, sospechosos ("suspected") de haber sufrido un fallo por todos (o algunos de) los participantes correctos, o sin fallos.

  "(Eventual) Weak Accuracy" : (Eventualmente) Existe algún participante, el cual nunca ha sido considerado sospechoso de sufrir un fallo.

- Recepción[15] Confiable de Mensajes ("Reliable Message Delivery"). Los participantes en un sistema distribuido necesitan comúnmente intercambiar mensajes entre si. Estos envíos de mensajes deben realizarse cumpliendo dos propiedades [Pan94]:

[15]Recepción, en este caso, significa que un proceso examina el contenido de un mensaje y actúa acorde con el mismo ("deliver").

- El mensaje enviado por un nodo i es siempre recibido ("received") correctamente por otro nodo j – es decir, íntegro.
- Los mensajes enviados por un nodo i son examinados ("delivered") por un nodo j en el mismo orden de envío utilizado por i.

- Orden. En un sistema distribuido es algunas vez imposible definir cuando un evento ocurre después de otro. Sin embargo, el orden de los eventos ocurridos en diferentes nodos, o procesos, de un sistema distribuido puede ser muy importante. Por otra parte, todos los relojes de un sistema nunca están perfectamente sincronizados y tampoco pueden sincronizarse de manera perfecta [Lam78b]. Lamport, ante este problema, define la relación "sucedió antes" ("Happened before" o "causal ordering", en inglés), " $\rightarrow$", de la siguiente manera:

  Las acciones de enviar y recibir un mensaje se asumen como un evento dentro de un proceso.

  Definición. La relación "$\rightarrow$" en conjunto de eventos de un sistema es la relación más pequeña ("smallest relation") que satisface las tres siguientes condiciones:

  1. Si a y b son eventos de un mismo proceso, donde a sucede antes que b, entonces $a \rightarrow b$.
  2. Si a es el envío de un mensaje por parte de un proceso y b es la recepción del mismo mensaje por otro proceso, entonces $a \rightarrow b$.
  3. Si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$.

  Dos eventos a y b son concurrentes si $a \not\rightarrow b$ y $b \not\rightarrow a$.

  En [Lam78b], Lamport propone también un mecanismo mediante el cual se puede manejar numéricamente la relación "sucedió antes". Este mecanismo recibe el nombre de relojes lógicos ("logical clocks"). Los relojes lógicos pueden modificarse, empleando identificadores de procesos, para definir un orden total ("total order") en un sistema distribuido – es decir, un orden entre cualquier par de eventos del sistema.

  Un orden total, por ejemplo, puede ser muy útil en un sistema distribuido para sincronizar ("synchronization") los accesos conflictivos

sobre un conjunto de datos compartidos. Dos accesos son conflictivos si se refieren a una misma locación de memoria (o dato) y al menos uno de ellos es una operación del tipo read-modify-write [ea98c]. El orden total puede servir también para sincronizar la comunicación con dos o mas conjuntos de procesos [eac].

- Particiones de la Red ("Network Partitions"). Una partición de la red separa a un grupo de nodos en uno o mas subconjuntos. Existen dos clases, de acuerdo con [Fin]:

  1. Permanentes ("Permanent Partitions"). Las particiones permanentes ocurren cuando uno o mas nodos (procesos) no pueden comunicarse con el resto del grupo ("cluster") por razones físicas.
  2. Transitorias ("Transient Partitions"). Las particiones transitorias ocurren cuando la comunicación toma más tiempo del esperado – es decir, los mensajes no son recibidos a tiempo.

  Algunos sistemas distribuidos consideran la solución eventual del problema de una partición. En este caso, los diferentes subconjuntos deben encargarse de poder llegar a un estado consistente cuando se elimine la partición.

  Los servicios para el manejo de réplicas, por ejemplo, utilizan dos tipos de enfoque en cuanto a la presencia de particiones: protocolos optimistas ("optimistic approach") y pesimistas ("pessimistic approach"). Los primeros consideran que los conflictos con las actualizaciones son poco probables y, en el caso de presentarse, la mayoría son fáciles de resolver. Los protocolos pesimistas se preocupan mas por la consistencia de los datos que por su disponibilidad. Estos últimos utilizan normalmente un mecanismo de votación ("quorum") y proveen de una menor disponibilidad [Par93].

- Consistencia ("Consistency"). Cuando un conjunto de procesos trabaja en cooperación, es necesario asegurarse que los procesos operacionales – sin fallos – mantengan una vista consistente del estado general del grupo. Esta vista consistente puede incluir información sobre el mismo grupo o del estado de cada nodo – es decir, el estado de la computación en conjunto.

Considérese, por ejemplo, un grupo redundante de nodos donde cada uno guarda la copia de un dato w. Cada nodo puede experimentar fallos y, en respuesta a las peticiones de sus procesos, es responsable de actualizar todas las copias de w. La coordinación de esta actualización con las de otros nodos es importante para mantener la *consistencia* de las copias.

Si se considera el ejemplo anterior, la información de cuáles son los nodos activos, u operacionales[16], se vuelve también de vital importancia para cada nodo y debe conservarse siempre consistente. Esta información le permite conocer cuales nodos siguen los mismos pasos dentro de una computación en conjunto. Un modelo que permite implementar esta vista consistente del estado general del grupo es el de "view synchrony" [Bir93].

En gran parte, el problema de la consistencia en ambientes distribuidos es responsabilidad del *control de concurrencia* del sistema, el cual tiene que interactuar con los mecanismos de detección de errores, consenso, orden, los encargadados de la atomicidad en las operaciones, etc.

- Transparencia ("Transparency"). Existen diferentes tipos y definiciones de transparencia en un sistema distribuido [ea95a]: transparencia de acceso, locación, concurrencia, replicación, fallos, migración, rendimiento y escalabilidad.

  La transparencia, por otra parte, es importante para los usuarios y los programadores de aplicaciones porque facilita su interacción con el sistema. El encargado, en los sistemas distribuidos, de materializar esta transparencia son los procesos denominados "Front-Ends" (FE). Un Front-End sirve de intermediario entre los clientes y los servidores.

  Los Front-Ends, por ejemplo, son los encargados de hacer transparente el funcionamiento de un mecanismo de tolerancia a fallas[17]. Un FE puede encontrarse en diferentes casos de fallos y actuar de manera distinta en cada uno [BM95]:

[16] Sin fallos

[17] Un Front-End puede también ocuparse del *balanceo de carga* ("load balancing") y la *transparencia de locación, replicación o concurrencia.*

- El fallo de un servidor ocasiona que un FE no pueda terminar las peticiones de un cliente al servidor, lo cual presenta dos posibilidades:
  1. Si el servidor con fallos era el único disponible, el FE informa al cliente de la no disponibilidad del servicio.
  2. Si existen otros servidores que proveen el mismo servicio, el FE contacta alguno de los mismos y puede repetir la peticiónes inconclusas del cliente.
- Fallo en el FE. Un FE puede guardar periódicamente su estado actual en almacenamiento estable [18]. En el momento de recuperarse, el FE puede recurrir al último estado almacenado, contactar a los clientes y reiniciar sus operaciones.
- Cuando un cliente experimenta un fallo, el FE deja de recibir peticiones ("requests") del mismo. Con el fin de poder afrontar esta situación, el FE puede aprovechar el próximo almacenamiento en disco de su estado para guardar también los mensajes desde y para el cliente con fallos.

3.8 Transacciones y grupos

Dos son los tipos de abstracciones tradicionalmente utilizadas para construir un sistema distribuido tolerante a fallas: las transacciones y los grupos. Las transacciones provienen de la comunidad dedicada a las bases de datos; mientras que los grupos pertenecen a la de los sistemas distribuidos. Ambos modelos utilizan alguna forma de redundancia[19], o varias.

En la actualidad, por otra parte, existe la tendencia de combinar ambos esquemas. ISIS [Bir93] es un herramienta para construir software distribuido tolerante a fallas donde se utiliza un mecanismo de transacciones sobre una abstracción de grupos. Sus creadores afirman que la ausencia de una combinación semejante implica una mayor complejidad en los protocolos y un mayor soporte de infraestructura [BJ].

Por otra parte, la combinación de ambos enfoques representa un importante paso en la extensión de la comunicación de grupos y, por ende, en el diseño e implementación de aplicaciones confiables [SR96].

[18] Es decir, un tipo de almacenamiento tolerante a fallos.

[19] Ver sección 3.7 en página 49.

3.8.1 Transacciones en sistemas distribuidos

Un servicio de datos puede construirse con base en un sistema distribuido formado por un número de servidores –o computadoras– independientes.

En un ambiente distribuido, las operaciones de una transacción pueden involucrar elementos de datos ubicados en dos o más computadoras independientes, las cuales son susceptibles de sufrir todo tipo de fallos. En este contexto y para la tolerancia a fallas, el problema consiste entonces en contar con un algoritmo de recuperación distribuido. De esta forma, el empleo de transacciones distribuidas sigue requiriendo del supuesto de que cualquier nodo del sistema que haya sufrido un fallo debe ser recuperado lo más pronto posible. Sin un algoritmo de recuperación, la puesta en marcha de un nodo es capaz de dejar en un estado inconsistente a todo el sistema.

Los sistemas distribuidos tolerantes a fallas tradicionales se han apoyado en mecanismos de control de concurrencia y de recuperación de fallos para implementar un servicio persistente basado en transacciones atómicas ("atomic transaction semantics") [Cri91].

El ejemplo más representativo de una plataforma transaccional es el conjunto de bibliotecas denominado Arjuna [ea91].

Limitación de las transacciones

Entre las limitaciones más significativas de las transacciones es que no son apropiadas para el manejo de réplicas. Por ejemplo, si una transacción se utiliza en el proceso de modificación de un conjunto de réplicas y una de las actualizaciones no es posible llevarla a cabo, la transacción será abortada. En esta situación y si las réplicas son utilizadas como medida a tolerancia a fallas, el empleo de transacciones es más bien una desventaja al no garantizar la supervivencia de los procesos ("liveness") en la presencia de fallos [Gar98].

Otra desventaja importante de las transacciones se encuentra en su implementación. Cada servicio de transacciones distribuidas resuelve el problema de la *atomicidad respecto a fallos* de una manera *ad hoc* y sin posibilidad de modificación. Con el fin de aprovechar las semántica de las operaciones y disminuir el costo de un algoritmo de recuperación, la noción de atomicidad o transacción no deben ser parte íntegral e indispensable del lenguaje de programación [BV]. La solución ideal es permitir una mayor flexibilidad en

la utilización, o no, de una transacción de acuerdo a cada situación.

Por último, la solución de la *atomicidad respecto a fallos* se presta potencialmente a una relación beneficiosa con otros dos problemas: el de consenso y el de los servicios multicast totalmente ordenados. En particular, una solución a la *atomicidad respecto a fallos*, donde se toma la decisión de comprometer o abortar una transacción, puede construirse encima de de una abstracción que resuelva el problema del consenso de si debe llevarse a cabo la misma, o no, de acuerdo a cierta invariante [Had]. Por otra parte, las abstracciones encargadas de una comunicación multicast totalmente ordenada pueden utilizarse para resolver el problema de los interbloqueos en un protocolo de compromiso de dos fases, ver por ejemplo [Gar98].

3.8.2 Grupos

El ejemplo más característico de una técnica de *enmascaramiento* de fallas es el uso de un *grupo redundante* formado por servidores físicamente independientes[20].

Su principal ventaja se resume en un hecho de probabilidad: sea G un grupo redundante con N servidores físicamente independientes y donde cada uno cuenta con la misma probabilidad F de sufrir un fallo durante la ejecución de una computación C. Dentro del grupo G, mientras que existe potencialmente una alta probabilidad de que **al menos** un elemento sufra un fallo $(1-[1-(F)]^N)$, la probabilidad de que **todos** los elementos experimenten un fallo al mismo tiempo es muy baja (F^N).

El desarrollo de la abstracción de grupos se debe en gran parte a una necesidad dentro de la comunidad de los sistemas distribuidos [Bir93]. En este contexto, los problemas en el uso de un grupo son similares a los de estos sistemas[21].

Dentro de la tolerancia a fallas, el énfasis principal en la abstracción de grupos ha sido proveer de una mayor *supervivencia* ("liveness") en la prestación de un servicio. En [Pow96], David Powell presenta una colección de artículos con los proyectos más representativos en esta dirección.

[20] Ver sección 3.6.2 en página 46.
[21] Ver sección 3.7 en 50.

Los grupos, en particular, constituyen un medio conveniente para el manejo de réplicas. En este sentido, el direccionamiento de un solo grupo es más adecuado que el direccionamiento explícito de un número de réplicas. De esta forma, la comunicación propia empleada en esta abstracción sirve potencialmente de base para la implementación de diversas técnicas de replicación.

El aspecto más importante en el manejo simultáneo de múltiples réplicas como medida de tolerancia a fallas es el criterio de ***"secuencialidad"***[22]. Este criterio exige que el efecto resultante de los accesos de un número de clientes sobre un conjunto de elementos de datos replicados sea el mismo a que si estos se realizaran de manera secuencial y sobre un solo elemento [GS97]. La importancia de este criterio radica en que su cumplimiento preserva la semántica de cualquier programa que no tome en cuenta datos replicados.[23].

Un aspecto que es importante preservar en el manejo de un grupo es la notación en la invocación de las operaciones definidas en un servicio. Con un grupo redundante de servidores, un cliente p_i debe continuar realizando una sola invocación, $op(arg)$, y recibir una sola respuesta, $ok(res)$, en caso de ser necesario.

El cumplimiento del criterio de "secuencialidad" se traduce en el cumplimiento de dos propiedades:

Defínase como x a un servidor no replicado y a un conjunto de réplicas del mismo como $x^1, \ldots, x^n$.

Orden Sean $op(arg)$ y $op'(arg)$ dos invocaciones a un servidor redundante x y realizadas por dos clientes p_i y p_j, respectivamente. Cualesquiera dos réplicas x^k y x^l en el grupo atienden ("deliver"), cada una, las invocaciones $op(arg)$ y $op'(arg)$ en el mismo orden.

Atomicidad Este propiedad significa que si una réplica x^i del grupo atiende la invocación $op^i(arg)$ de un cliente p_i, entonces toda réplica sin fallos

[22] "One-copy equivalence" o "linearizability".

[23] La división entre la semántica de un programa y el control en el acceso concurrente a datos replicados, o sin replicar, se realiza normalmente por razones de rendimiento o de una política de consistencia más relajada. Sin embargo, este es un problema antiguo para el cual existe un patrón llamado "Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-Structured Concurrent I/O" [ea96a]

en el grupo atiende también la petición $op^i(arg)$.

Por otra parte, tres son los esquemas principales para el manejo de réplicas que pueden utilizarse en la abstracción de grupos:

1. **Primario-Respaldo**
2. **Replicación Activa.**
3. **Votación.**

Primario-Respaldo

Este esquema corresponde al de un grupo *débilmente acoplado* mencionado en la sección 3.6.2, página 46. El esquema primario-respaldo se describe a profundidad en [ea93b].

Sus actores principales son:

- **Réplica Primaria.**
- **Réplica(s) de Respaldo.**

La función de una réplica primaria, $prim(x)$, es primordialmente recibir las invocaciones de los clientes, procesarlas y producir el cambio de estado y respuesta correspondiente, en su caso. Las réplicas de respaldo nunca interactúan directamente con los clientes y lo hacen únicamente con la réplica primaria. En operación normal, o sin fallos en ninguno de los actores, la comunicación entre las partes se realiza siguiendo los pasos descritos a continuación [GS97] (ver figura 3.4):

1. Un proceso p_i invoca una operación $op(arg)$ en la réplica primaria, $prim(x)$, acompañada de un identificador de invocación, $invID$.
2. La réplica primaria, $prim(x)$, lleva a cabo la operación $op(arg)$ y genera una respuesta, res. Una vez realizada la *actualización* $(state - update)$ de su estado interno , $prim(x)$ envía un mensaje a las réplicas de respaldo de la forma $\{invID, res, state - update\}$. Al momento de recibir este último mensaje, las réplicas de respaldo actualizan su propio estado interno y devuelven un mensaje de confirmación, ack, a la réplica primaria.

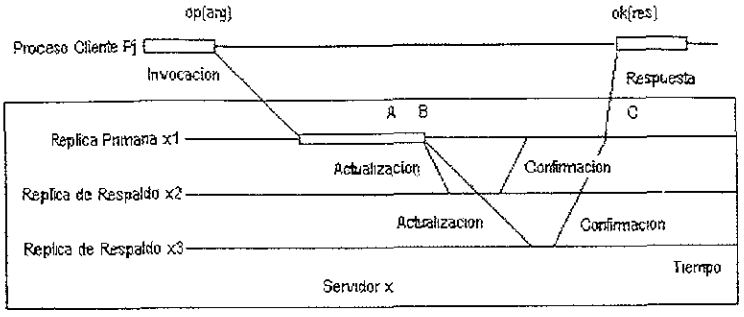

Figura 3.4: Esquema primario-respaldo.

3. Una vez recibidas todos las confirmaciones de todas las réplicas de respaldo *sin fallas*, $prim(x)$ devuelve el resultado, res, al proceso p_i.

Una ventaja importante de este esquema es el aseguramiento del criterio de "secuencialidad". En primer lugar, la réplica primaria define un *orden* definitivo en la atención de todas las invocaciones. El empleo de mensajes de confirmación, *acks*, sirve potencialmente de base para un protocolo de compromiso que garantice la *atomicidad* en la actualización de los estados internos de las réplicas de respaldo.

En operación normal, el fallo de una réplica *primaria* puede ocurrir en dos puntos de la atención de una invocación, $op(arg)$:

1. Antes de enviar los mensajes de actualización correspondientes a cada réplica de respaldo (punto A en la figura 3.4.).

2. En el instánte, o después, de enviar los mensajes de actualización correspondientes a cada réplica de respaldo y antes de devolver una respuesta, res, al proceso cliente p_i (punto B en la figura 3.4). La propiedad de *atomicidad* permite garantizar que todas o ninguna de las réplicas de respaldo actualice su estado interno.

En ambos casos, todos los clientes de la réplica primaria, $prim(x)$, dejan de recibir respuesta alguna.

Después de detectar o suponer el fallo en una réplica primaria, el paso más importante a seguir por todos sus clientes es seleccionar una misma

réplica primaria entre el conjunto de respaldo. Una vez con la identidad de la nueva réplica primaria, cada cliente debe volver a invocar la operación anterior, $op(arg)$.

En el primer caso, la nueva réplica primaria considera simplemente la invocación $op(arg)$ como nueva.

El segundo caso presenta, a su vez, dos posibilidades:

2.1 Todas las réplicas de respaldo actualizan su estado interno. En este caso, la nueva réplica primaria recibe una invocación, $op(arg)$, con el mismo $invID$ del último mensaje de actualización atendido. De esta forma, la nueva invocación no es necesario volver a realizarla y se regresa el resultado correspondiente, res.

2.2 Ninguna réplica de respaldo actualiza su estado interno. Esta situación es tratada exactamente como en el primer caso.

Replicación activa

Este esquema corresponde al de un grupo *fuertemente acoplado* mencionado en la sección 3.6.2, página 46. Una descripción a fondo de este enfoque es posible encontrarla en [Sch90].

Los actores o réplicas en este esquema no se dividen en ninguna clase y guardan todos un mismo nivel jerárquico desde el punto de vista de los clientes.

En este caso, el manejo de las réplicas se realiza de la siguiente manera [GS97] (ver figura 3.5):

Considérese un proceso p_i que invoca una operación $op(arg)$ sobre un grupo formado por un conjunto de servidores idénticos $x^1, \ldots, x^n$.

1. La invocación $op(arg)$ es realizada por el cliente p_i en todas y cada una de las réplicas $x^1, \ldots, x^n$.

2. Cada réplica x procesa la invocación $op(arg)$, actualiza su estado interno y produce la respuesta correspondiente res, en su caso.

Capítulo 4

Diseño de FT-JavaSpaces

El propósito de este capítulo es definir los requerimientos que describen el nuevo servicio de espacio de tuplas tolerante a fallas propuesto, llamado FT-JavaSpaces.

A continuación, se definen las hipótesis básicas de FT-JavaSpaces y los tipos de fallos que tolera.

4.1 Requerimientos

Los requerimientos de FT-JavaSpaces son:

- FT-JavaSpaces exporta las cinco operaciones básicas de Linda: *out()*, *in()*, *inp()*, *rd()* y *rdp()*.

  La operación *eval()* no está implementada en FT-JavaSpaces por considerarse que aumenta un trabajo computacional adicional e innecesario al sistema, el cual puede llevarse a cabo perfectamente por sus clientes.

  Por una parte, el objetivo de los procesadores encargadados de las funciones de FT-JavaSpaces es implementar un servicio de espacio de tuplas tolerante a fallas y no el de realizar un trabajo por encargo de los clientes como sucede en el caso de la operación *eval()*. En segundo lugar, el trabajo ordenado por una operación *eval()* puede describirse también en una *tupla pasiva* y ser ejecutado por cualquiera de un número de clientes de FT-JavaSpaces encargados de prestar su servicio de procesador.

- FT-JavaSpaces debe cumplir también con la semántica de operación *normal* (SON) del *modelo espacio de tuplas y definida originalmente* en Linda [CG86a].

- El requerimiento de disponibilidad de FT-JavaSpaces puede definirse como el de una mayor probabilidad de que preste su servicio en un instante de tiempo en comparación con la de una sola implementación de un espacio de tuplas no tolerante a ninguno de los fallos que enmascara FT-JavaSpaces.

- En cuanto al requerimiento de confiabilidad de FT-JavaSpaces, éste puede definirse como el de una mayor probabilidad de que una computación que utiliza FT-JavaSpaces termine satisfactoriamente su objetivo en contraste a si utilizara una implementación de un espacio de tuplas no tolerante a ninguno de los fallos que enmascara FT-JavaSpaces.

- FT-JavaSpaces ofrece una vista transparente de sus mecanismos propios. En este sentido, la acción de los mecanismos de tolerancia a fallas de FT-JavaSpaces es completamente invisible a los usuarios del espacio de tuplas.

4.2 Ambiente de implementación

Los supuestos acerca del ambiente de implementación y las condiciones de operación de FT-JavaSpaces son:

- Número de recursos. En cuanto a su cantidad, el número de fallos simultáneos que debe tolerar FT-JavaSpaces es solamente de un fallo. FT-JavaSpaces es clasificado entonces como un sistema *single fault-tolerant*.

- Los programas que implementan a FT-JavaSpaces están libres de fallas de programación ("bugs").

- Todos los programas son correctamente compilados y ejecutados por procesadores del tipo "Fail-Stop". Esta suposición deja fuera la posibilidad de que el procesador experimente fallos arbitrarios.

- Los procesadores no cuentan con ningún tipo de *almacenamiento estable* para almacenar el último estado consistente de una computación ("checkpointing").

- La invocación de una petición de servicio por parte de un cliente a FT-JavaSpaces es considerada una clase de RPC ("Remote Procedure Call").
- Los procesos clientes utilizan un *protocolo de comunicación confiable* ("reliable") y punto a punto ("peer to peer") para el intercambio de datos con FT-JavaSpaces. En este sentido, el protocolo es *confiable* por que enmascara los siguientes tipos de fallos en el canal de comunicación:

  1. Fallos de valor. Es decir, los mensajes no se corrompen durante su transmisión.
  2. Fallos de omisión. Es decir, los mensajes no se pierden en su transmisión a través del canal de comunicación.

- El canal de comunicación sigue un modelo "*FIFO*" ("first in, first out") en el orden de transmisión de los datos.
- La red de comunicación entre FT-JavaSpaces y sus clientes posee un ancho de banda y tiempos de transmisión y recepción de mensajes variables.
- El protocolo utilizado en la comunicación entre los procesos clientes y la implementación de un espacio de tuplas es implementado por un sistema del tipo "Fail-Stop".

4.3 Fallos a tolerar

El hardware que compone la implementación de un espacio de tuplas son procesadores, memorias volátiles y canales de comunicación. Todos estos elementos son susceptibles de sufrir un fallo. En contraste, el software de FT-JavaSpaces no sufre un desgaste y su función es tolerar fallos en el hardware.

Los fallos que tolera FT-JavaSpaces en una implementación de un espacio de tuplas son:

1. Omisión del Servidor.
2. Caída del Procesador.
3. Caída del Canal de Comunicación.

Omisión del servidor

En respuesta a una operación, el servidor encargado de implementar las funciones de un espacio de tuplas puede no generar el cambio de *estado interno* o resultado correspondiente. En otras palabras, el servidor es susceptible de sufrir un fallo del tipo *omisión*. Las causas que llevan al servidor a experimentar este fallo son:

Considérese una tupla t correspondiente al resultado ("output") de una operación **op-x(t)** realizada por un cliente C. **op-x(t)** equivale a una de las siguientes operaciones **rd(t)**, **rdp(t)**, **in(t)** o **inp(t)**.

1. Corrupción transitoria de una tupla en el espacio ("soft error"). La tupla t no es devuelta a C si el servidor encuentra un error en t. Sin embargo, una subsecuente e inmediata operación **op-x(t)** no se percata del mismo error.
2. Corrupción permanentes de una tupla en el espacio ("persistent error"). La tupla t no es devuelta a C si el servidor encuentra un error en t. En este caso, cualquier operación **op-x(t)** subsecuente sigue detectando el mismo error.
3. El servidor no realiza un cambio en el estado interno del servicio si detecta un error en una tupla t_x utilizada como argumento de una operación **out**(t).

Fallos del tipo caída

Este modelo considera que los únicos elementos capaces de sufrir un fallo del tipo caída son el procesador y el canal de comunicación. Además, en ambos casos la caída de uno de los mismos es detectable.

En particular, las causas que llevan a la caída de un procesador son:

1. La detección de un estado interno *inconsistente* en el servicio de espacio de tuplas –la detección de un error.
2. La falta de recursos necesarios para continuar con el servicio de espacio de tuplas –la detección de una falla.
3. Fallas de naturaleza física como el corte de la energía eléctrica, un incendio, etc.

4.4 Estrategia de tolerancia a fallas

FT-JavaSpaces emplea específicamente un enfoque de *enmascaramiento* con redundancia simple (2 réplicas) para tolerar los tres tipos de *fallos* abordados en las especificaciones:

1. Omisión del servidor,
2. Caída del procesador y
3. Caída del canal de comunicación.

El diseño de FT-JavaSpaces se fundamenta en la utilización de dos réplicas independientes de un espacio de tuplas.

En particular, el esquema para el manejo del grupo que utiliza FT-JavaSpaces es el de *primario-respaldo* (ver sección 3.8.2). De esta forma, una de las implementaciones es denominada *primaria* y la otra de *respaldo*.

4.5 Arquitectura funcional

En FT-JavaSpaces, el manejo de los espacios de tuplas primario y de respaldo es realizado por uno o varios *Front-Ends* ubicados también, cada uno, en máquinas física y eléctricamente independientes. FT-JavaSpaces puede considerarse entonces como un sistema con tres capas conceptuales (ver figura 4.1):

1. Aplicación
2. Manejo del grupo
3. Grupo redundante de espacios de tuplas (primario y de respaldo)

El servidor que implementa el servicio que ofrece FT-JavaSpaces se compone por la capa encargada del manejo del grupo (capa No. 2) junto con el par de implementaciones de los servicios de los espacios de tuplas primario y de respaldo.

Una aplicación utiliza el servicio de FT-JavaSpaces por medio de un representante (*proxy*) del mismo, el cual reside en la misma máquina de la aplicación. El objetivo de este representante es que un programa cuente con

la funcionalidad de un servicio sin necesidad de implementarlo. En un ambiente distribuido, la ventaja del uso de representantes es que pueden existir en diferentes máquinas a las del servidor y actuar en favor de un servicio único.

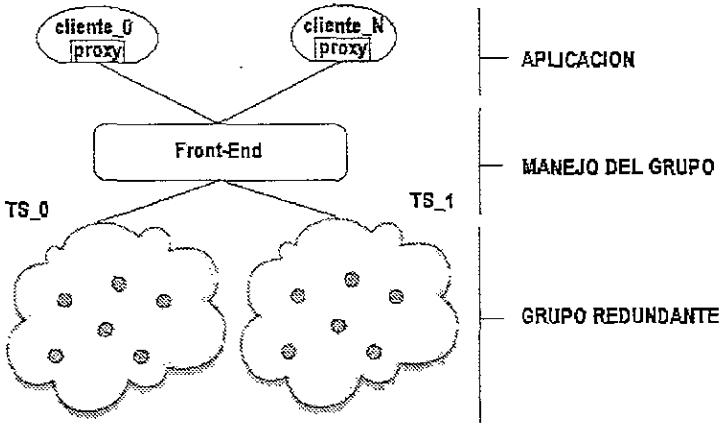

Figura 4.1: Manejador del grupo (o *Front-End*).

El conjunto de *Front-Ends* es el encargado de manejar el acceso concurrente de los clientes al sistema, implementar la política de tolerancia a fallas y de la recuperación del par de implementaciones de un espacio de tuplas. En general, funcionan como intermediarios entre los clientes y el grupo.

En el diseño de FT-JavaSpaces, los fallos que pueda experimentar un *Front-End* no son tampoco considerados. De esta forma, las secciones siguientes consideran a la operación de un *Front-End* como libre de cualquier tipo de fallo. La razón principal de la ausencia de una política de tolerancia a fallas en los *Front-Ends* es que su implementación es un problema que merece un estudio por si mismo y la adopción de un tipo adicional de redundancia física o de tiempo muy particular. Este problema en los *Front-Ends* es el mismo de la tolerancia a fallas en procesos, el cual toma normalmente como base el empleo de transacciones.

4.5.1 Configuración de Front-Ends

Los Front-Ends en FT-JavaSpaces pueden ser uno o varios:

- Un Front-End: en este caso, un sólo *Front-End* es el encargado del grupo redundante de espacios de tuplas (figura 4.2). Esta versión facilita básicamente el control de concurrencia y la implementación del algoritmo de recuperación. Por otra parte, el *Front-End* se convierte potencialmente en un cuello de botella importante.

- Varios Front-Ends: dos o más *Front-Ends* interactúan directamente con el grupo redundante de espacios de tuplas (figura 4.3). La principal ventaja de esta configuración es que elimina los cuellos de botella; sin embargo, la implementación del algoritmo de recuperación y el control de concurrencia se vuelven mas complicados.

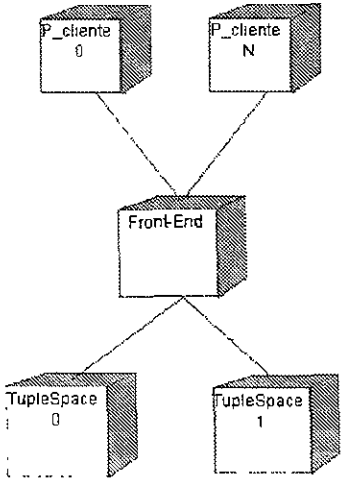

Figura 4.2: Configuración centralizada.

4.5.2 Capas funcionales de un Front-End

El diseño de cada Front-End es una adaptación de un **patrón** denominado *backup pattern*[1] y definido en [ea96a]. Una ventaja de este patrón de diseño es que ofrece diferentes opciones para la realización de cierta función y permite al sistema alternar entre las mismas de manera dinámica y transparente.

[1] En particular, el *backup pattern* es un patrón de diseño para sistemas de *software* orientado a objetos.

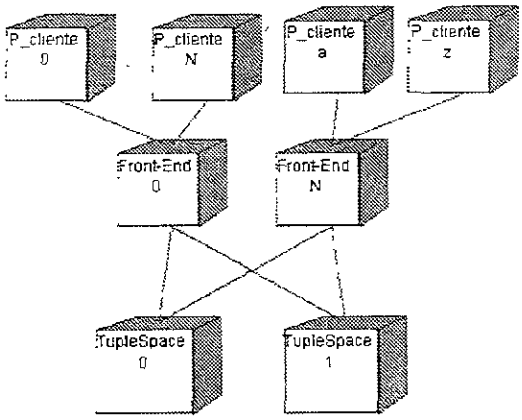

Figura 4.3: Configuración distribuida.

Un *Front-End* puede dividirse en cuatro capas funcionales (ver figura 4.4). A continuación, se definen éstas últimas junto con el rol que desempeñan:

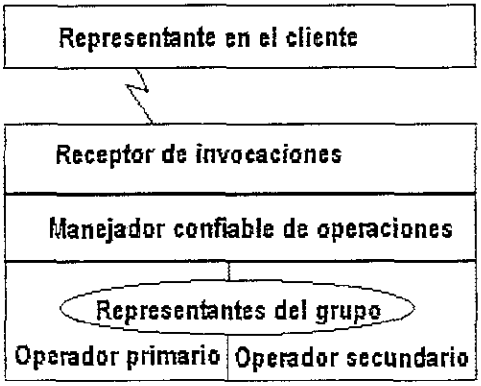

Figura 4.4: Capas funcionales de un *Front-End*

1. Representante en el cliente (*Proxy*).- Esta capa del *Front-End* en el cliente sirve como referencia al servicio que `FT-JavaSpaces` ofrece. Su rol es básicamente:

- Enviar peticiones de servicio al Receptor de Invocaciones.

2. Receptor de Invocaciones (*RDI*).- Dentro de un *Front-End*, el receptor de invocaciones es el encargado de la creación, manejo y destrucción de los hilos de control (*threads*) correspondientes a cada petición.

   De esta forma, los roles básicos de un receptor de invocaciones son:

   - Recibir las peticiones de servicio de los clientes.
   - Delegar cada petición a un *manejador confiable de operaciones*.

3. Manejador Confiable de Operaciones (*MCO*).- Es el encargado principal de implementar el esquema *primario-respaldo* de manera transparente. Con este fin, crea y maneja dos operadores, denominados *primario* y *secundario*, encargados de cumplir con una petición de servicio dependiendo del estado del grupo. Es decir, si tanto el espacio de tuplas primario y de respaldo se encuentran funcionando o sólo uno de ellos. Las acciones de un *MCO* se realizan en un hilo de control independiente.

   La roles principales que lleva a cabo son:

   - Atender (*deliver*) la petición asignada por el receptor de invocaciones (*RDI*).
   - Determinar la utilización correspondiente de un operador de acuerdo a la existencia, o no, de fallos y su ubicación (i.e., confinamiento de fallas/errores).

   Existen cinco clases de *manejadores confiables de operaciones* correspondientes a cada operación del servicio de `FT-JavaSpaces`.

4. Representantes del Grupo.- Un represente del grupo es una abstracción con funciones específicas de cómo y cuántos espacios de tuplas centralizados deben ser utilizados en la atención de una petición.

   El representante de un grupo pertenece potencialmente a una de dos clases: *operador primario* u *operador secundario*. En concordancia con el esquema *primario-respaldo* utilizado por `FT-JavaSpaces`, ambos operadores consideran a un espacio de tuplas como *primario* y al restante como de *respaldo*. De igual forma, los canales de comunicación entre el *Front-End* y los espacios *primario* y de *respaldo* se

denominan canal *primario* y canal de *respaldo*, respectivamente.

Los roles específicos de un representante de grupo son:

- Interactuar directamente con los espacios de tuplas (primario y de respaldo).
- Detectar fallos o errores en el grupo y canales de comunicación (i.e., detección de fallas/errores).

4.1 Operador primario (*OP*).- Este operador intenta últimamente realizar la acción de una petición en ambos o en uno solo de los espacios de tuplas del grupo, dependiendo de la operación: en ambos espacios para las operaciones out(), in() e inp() y únicamente en el primario en el caso de las operaciones rd() y rdp(). Además, es el encargado de asegurar el criterio de *secuencialidad*[2] del esquema *primario-respaldo*.

4.2 Operador secundario (*OS*).- Este operador entra en acción cuando el primario no puede llevar a cabo su función. En presencia de una falla en el espacio *primario*, un operador secundario intenta realizar directamente la acción de una petición en el espacio de *respaldo*.

4.6 Modos de servicio

De acuerdo con la existencia de los tipos de fallos que tolera, `FT-JavaSpaces` puede encontrarse en tres modos de operación:

1. Modo completo. No existen fallos en el sistema.
2. Modo reducido. Existe un fallo en el sistema.
3. Modo con fallo. `FT-JavaSpaces` sufre un fallo del tipo caída debido a la presencia de mas de uno de los fallos que tolera en su ambiente de implementación.

4.7 Operaciones en modo completo de operación

`FT-JavaSpaces` exporta cinco de las operaciones básicas de Linda: **out()**, **in()**, **rd()**, **inp()** y **rdp()**.

[2]Es decir, orden y atomicidad.

Las operaciones que exporta FT-JavaSpaces pueden dividirse en dos grupos de acuerdo a si involucran tanto el espacio primario como el de respaldo o solamente el primero:

1. Operaciones del tipo *lectura*: rd() y rdp(). Las características principales de estas operaciones es que *no* modifican el estado interno y que se realizan únicamente en el espacio *primario*.

2. Operaciones del tipo *escritura*: out(), in() e inp(). En cambio, los miembros de este conjunto *si* actualizan el estado interno y se llevan a cabo en ambos espacios de tuplas.

De esta forma, la actualización del estado interno del espacio de *respaldo* es exclusivamente necesaria para las operaciones del tipo *escritura*.

4.7.1 Operaciones del tipo lectura

Este par de operaciones toma en cuenta únicamente al espacio *primario* y su ejecución se traduce también en una operación en el mismo. La tabla 4.1 resume las operaciones respectivas en el espacio *primario* junto con su operación correspondiente de FT-JavaSpaces.

Op. de FT-JavaSpaces	Op. en E. *primario*
rd(t)	rd(t)
rdp(t)	rdp(t)

Tabla 4.1: Operaciones del tipo lectura.

4.7.2 Operaciones del tipo escritura

Estas operaciones involucran tanto al espacio *primario* como el de *respaldo* y se traducen en la ejecución de dos operaciones en ambos espacios de manera consecutiva. La operación respectiva al espacio *primario* se realiza primero y la del de *respaldo* a continuación.

Considérese una tupla t utilizada como argumento de una operación out(t), in(t) o inp(t). La tabla 4.2 resume las operaciones respectivas en cada espacio de tuplas (primario y de respaldo) y correspondientes a una operación del servicio de FT-JavaSpaces.

Op. de FT-JavaSpaces	Op. en E. *primario*	Op. en E. de *Respaldo*
out(t)	out(t)	out(t)
in(t)	in(t)	in(t)
inp(t)	inp(t)	**in(t)**

Tabla 4.2: Operaciones del tipo escritura.

4.8 Evolución de las réplicas

Sea $op'(arg)$ una operación de FT-JavaSpaces y $op(arg)$ una operación del servicio del espacio primario o de respaldo.

La invocación de una operación $op'(arg)$ del tipo *escritura* obliga a FT-JavaSpaces a realizar la siguiente secuencia de pasos:

1. FT-JavaSpaces invoca la operación $op(arg)$ respectiva en el espacio *primario*, $prim(x)$.
2. $prim(x)$ atiende la operación $op(arg)$ y, en su caso, genera una respuesta, res, almacenada temporalmente por FT-JavaSpaces en el *Front-End*. A continuación, FT-JavaSpaces realiza la *actualización* $(state - update)$ del estado interno correspondiente al espacio de *respaldo* utilizando a res como patrón, o argumento.
3. Una vez realizada la actualización $(state - update)$ en el espacio de *respaldo*, FT-JavaSpaces devuelve el resultado res al cliente.

En contraste con el esquema *primario-respaldo*, en este caso no es necesario que el espacio *primario* envié un mensaje de la forma $\{invID, res, state-update\}$ al espacio de *respaldo* para que realice la actualización de estado interno. Toda la información respecto al cliente $(invID)$, el resultado (res) y la actualización necesaria en el servidor de *respaldo* $(state - update)$ es guardada y utilizada por el *Front-End* de manera **transparente** para los clientes.

Por otra parte, la invocación de una operación $op'(arg)$ del tipo *lectura* obliga a FT-JavaSpaces a realizar la siguiente secuencia de pasos:

1. FT-JavaSpaces invoca la operación $op(arg)$ en el espacio *primario*, $prim(x)$.

2. $prim(x)$ atiende la operación $op(arg)$ y genera una respuesta, res.

3. FT-JavaSpaces devuelve el resultado res al cliente.

Con cero fallos en el grupo o canales de comunicación, un operador primario (OP) es el encargado de utilizar el número de espacios de tuplas requeridos de acuerdo al tipo de operación (Modo completo).

4.9 Semántica de operaciones concurrentes

En FT-JavaSpaces, las operaciones de diferentes procesos clientes pueden realizarse de manera secuencial o concurrente.

Operaciones Secuenciales Dos operaciones O_x y O_y emitidas por los procesos P_x y P_y, respectivamente, se dice que son secuenciales, denotado $O_x \rightarrow O_y$, si O_x es recibida y atendida por el espacio de tuplas antes de recibir la operación O_y.

Operaciones Concurrentes Dos operaciones O_x y O_y emitidas por los procesos P_x y P_y, respectivamente, se dice que son concurrentes, denotado $O_x||O_y$, si $O_x \nrightarrow O_y$ y $O_y \nrightarrow O_x$ se cumplen[3] [Lam78b]. En otras palabras, O_x no antecede secuencialmente a O_y ni viceversa. En este sentido, cualesquiera dos operaciones O_x y O_y concurrentes, $O_x||O_y$, deben originarse necesariamente en dos procesadores diferentes P_x y P_y. Cualquier operación O_z cumple con la propiedad $O_z \nrightarrow O_z$; es decir, en ningún sistema, una operación puede suceder antes que ella misma. La relación de concurrencia, $||$, guarda la propiedad de conmutatividad, lo cual significa que $O_x||O_y$ y $O_y||O_x$ son equivalentes.

Por otra parte, FT-JavaSpaces hereda tres importantes cualidades de los sistemas de *multiprocesadores* con una sola memoria, o *memoria compartida*:

1. Cualquier par de operaciones emitidas por un mismo procesador P_z sobre la memoria guardarán siempre un orden secuencial. Por esta razón, las operaciones provenientes de un mismo procesador se denominan **totalmente ordenadas**. En este contexto, las operaciones de P_z se dice que guardan también un **orden parcial** con respecto a las operaciones de otros procesadores.

[3] Es decir, si $O_x \rightarrow O_y$ y $O_y \rightarrow O_x$ no se cumplen.

2. La comunicación y sincronización se implementan usualmente mediante la compartición controlada de datos en memoria [Bri98]. En este sentido, cada emisión de una operación por parte de un procesador no está en coordinación con ninguna otra realizada por cualquier otro procesador. Por esta razón, cualquier par de operaciones provenientes de dos procesadores *distintos* se *clasifican como* **globalmente desordenadas**.

3. Los accesos a memoria se realizan normalmente de manera asíncrona. Debido a esta situación, la comunicación entre procesos se realiza comúnmente gracias a un mecanismo de sincronización [Bri98]. Es decir, la comunicación se implementa gracias a un acceso mutuamente exclusivo ("mutually exclusive access") a ciertas partes de la memoria, llamadas por lo general "mailboxes". Esta situación evita que un *mensaje sea borrado antes de tiempo y permite que sea recibido en el* momento exacto.

4.9.1 Modelo de consistencia secuencial

Una forma de clasificar los diferentes sistemas de memoria compartida es mediante su modelo de consistencia. Un modelo de consistencia define el orden legal de las referencias hechas a la memoria por un procesador, desde el punto de vista de otros procesadores [ea98b].

En un sistema de memoria compartida, la única manera en que dos procesadores pueden afectar la ejecución de uno y otro es mediante la actualización de datos en memoria. De esta manera, el orden en la realización de los eventos en memoria se vuelve de gran importancia. En particular, la correctez ("correctness") en la ejecución de las operaciones depende del comportamiento correcto y esperado del modelo de consistencia utilizado [Bri98].

Linda, al igual que FT-JavaSpaces, maneja un modelo de consistencia del tipo *secuencial* [ea98a]. De acuerdo con Lamport [ea98d], este modelo se define como:

Consistencia Secuencial Un sistema es secuencialmente consistente si el resultado de cualquier ejecución es el mismo que si las operaciones de todos los procesadores se hubieran ejecutado en algún orden secuencial, y las operaciones de cada procesador aparecen en esta secuencia en el orden especificado en su programa.

De esta forma, FT-JavaSpaces presta un orden de atención equivalente al secuencial aún cuando reciba peticiones de servicio de manera concurrente[4]. Particularmente, FT-JavaSpaces adopta el orden secuencial de atención que imponga el espacio primario.

El hecho de que FT-JavaSpaces maneje un modelo de consistencia secuencial y que cualquier par de operaciones provenientes de dos procesadores distintos se clasifiquen como **globalmente desordenadas** signfica que todas las operaciones sobre un espacio de tuplas están **totalmente ordenadas** desde el punto de vista del espacio [5].

Operaciones secuenciales

Si es posible *garantizar* que cualquier par de operaciones de un número de procesadores guarda un orden secuencial, dicha secuencia de operaciones es equivalente a la de un sólo procesador qne realiza operaciones sobre un espacio. En este sentido, esta clase de secuencia de operaciones no causa ninguna ambigüedad posible en la semántica (significado) resultante de su ejecución.

Operaciones concurrentes

Desde el punto de vista de los procesadores, el orden secuencial adoptado finalmente por FT-JavaSpaces en la atención de un número de operaciones concurrentes es capaz de afectar la semántica resultante de las mismas.

Una característica importante de FT-JavaSpaces es que el efecto de un número de operaciones concurrentes sobre una misma tupla es el mismo de la ejecución de las mismas en un determinado orden secuencial y dictado por el espacio primario, en concordancia con el esquema primario-respaldo. Por ejemplo, si dos procesadores realizan una operación **in(t)** de manera concurrente sobre una tupla única **t**, uno logrará el retiro mientras que el otro entrará en un estado de bloqueo.

Otro tipo de acceso concurrente se presenta cuando un número de operaciones concurrentes se realiza sobre un mismo número de tuplas distintas. En este caso, la semántica resultante de las operaciones no se ve afectada

[4] Aquí se incluyen tanto accesos concurrentes a diferentes tuplas, como a una misma

[5] Incluyendo tanto las operaciones secuenciales, como concurrentes, de dos o mas procesadores.

por el orden de su atención en el espacio.

En conclusión, el problema más grave de concurrencia en FT-JavaSpaces ocurre cuando un número de operaciones concurrentes hacen referencia a una misma tupla y el resultado de una operación influye en el de las otras. En este último caso, dichas operaciones se denominan como *conflictivas*.

4.9.2 Semántica resultante de operaciones conflictivas

En FT-JavaSpaces y en Linda en general, la relación entre el resultado semántico y el orden interno de ejecución de un conjunto de operaciones *conflictivas* se puede dividir en dos clases [LS99]:

Dependientes (D) El resultado semántico de las operaciones *conflictivas* depende totalmente del orden interno adoptado por el espacio de tuplas.

Independientes (I) El resultado semántico de las operaciones *conflictivas* es independiente totalmente del orden interno adoptado por el espacio de tuplas.

La tabla 4.3 resume la relación entre el resultado semántico y el orden interno de atención para un par de operaciones *conflictivas*. Esta tabla es simétrica debido a la propiedad de conmutatividad en las operaciones concurrentes, ||.

Los supuestos para la realización de esta tabla son los siguientes:

- Las operaciones **rd()** e **in()**, así como sus versiones **rdp()** e **inp()**, utilizan siempre un mismo patrón de coincidencia ("template"), llamado P.
- La operación **out()** inserta siempre una tupla que coincide con el patrón P.
- Una tupla es solamente la que coincide en todo momento con el patrón P.

El caso de un conjunto de operaciones *conflictivas* con tres o más elementos es deducible a partir del caso de dos operaciones. Si la relación *resultado-orden* de cualquier *combinación* con dos elementos del conjunto es dependiente, entonces lo es también la relación entre el resultado semántico y el orden de ejecución del conjunto entero.

$O_x \| O_y$	out(P)	rd(P)	in(P)	rdp(P)	inp(P)
out(P)	I	I	I	D	D
rd(P)		I	D	I	D
in(P)			D	D	D
rdp(P)				I	D
inp(P)					D

Tabla 4.3: Depedencia entre el resultado semántico y el orden de ejecución de dos operaciones *conflictivas*.

4.9.3 Resultado semántico dependiente del orden

La tabla 4.3 señala nueve casos en donde el resultado semántico de un par de operaciones *conflictivas depende* del orden de su ejecución.

Considérese un par de operaciones *conflictivas*, $P_x[op_a(t)]||P_y[op_b(t)]$, en esta categoría. Internamente, el espacio con la tupla t tiene dos opciones para el ordenamiento de ambas operaciones $op_a(t)$ y $op_b(t)$:

- Considerar $op_a(t)||op_b(t)$ como $op_a(t) \rightarrow op_b(t)$ o
- Considerar $op_a(t)||op_b(t)$ como $op_b(t) \rightarrow op_a(t)$.

En ambos casos anteriores, el *resultado semántico* de la ejecución del par de operaciones es distinto. Sin embargo, ambos resultados son correctos desde el punto de vista tanto de los procesos que las emitieron, P_x y P_y, como del espacio de tuplas donde se encontraba la tupla t.

Para el espacio de tuplas en donde se halla la tupla t es perfectamente válido que se lleve a cabo primeramente la operación de P_x o de P_y. La razón es su definición de un orden *secuencial* en la atención de operaciones concurrentes (ver sección 4.9.1 en página 80).

Las razón de la validez de ambos resultados desde el punto de vista de los procesos, P_x y P_y, cuenta con dos argumentos principales:

1. Si cualquier par de operaciones provenientes de P_x y P_y se encuentran *globalmente desordenadas* entonces ambos procesadores no anticipan ningún orden específico para la atención de sus operaciones.

2. Si la *única forma* en que P_x y P_y pueden afectar la ejecución de uno y otro es mediante la *actualización* de t entonces el orden *secuencial*[6] en la ejecución de sus operaciones no afecta la sincronización y comunicación **entre** ellos.

Sean, por ejemplo, dos operaciones *conflictivas* $P_x[out(t)]||P_y[inp(t)]$. Entonces, ambas operaciones pueden ser ordenadas internamente por el espacio dueño de t de manera equivalente a una de las siguientes formas:

P_x**(out(t))** $\rightarrow$ P_y**(inp(t))**.- P_x inserta inicialmente la tupla t en el espacio de tuplas. A continuación, P_y retira inmediatamente la tupla t del espacio.

P_y**(inp(t))** $\rightarrow$ P_x**(out(t))**.- Inicialmente, P_y no encuentra ninguna tupla t en el espacio de tuplas y, por ende, devuelve inmediatamente un valor booleano $false$ al proceso P_y. Enseguida, P_x inserta la tupla tupla t en el espacio.

En conclusión, P_x(out(t)) $\rightarrow$ P_y(inp(t)) $\neq$ P_y(inp(t)) $\rightarrow$ P_x(out(t)).

4.9.4 Resultado semántico independiente del orden

La tabla 4.3 señala seis casos en donde el resultado semántico de un par de operaciones *conflictivas* no depende del orden de su ejecución.

Tómese en cuenta el ejemplo anterior; es decir, dos operaciones *conflictivas* $P_x[op_a(t)]||P_y[op_b(t)]$. De igual forma, el espacio con la tupla t tiene dos opciones para el ordenamiento de las operaciones $op_a(t)$ y $op_b(t)$.

En esta categoría, sin embargo, el resultado semántico de la ejecución del par de operaciones es el mismo en ambos ordenamientos. El resultado sigue siendo además correcto desde el punto de vista de los procesos y del espacio por las mismas razones descritas en la sección anterior.

Sean, por ejemplo, dos operaciones *conflictivas* $P_x[out(t)]||P_y[in(t)]$. De esta forma, ambas operaciones pueden ser ordenadas internamente por el espacio dueño de t de manera equivalente a una de las siguientes formas:

P_x**(out(t))** $\rightarrow$ P_y**(in(t))**.- P_x inserta inicialmente la tupla t en el espacio de tuplas. A continuación, P_y retira la tupla t del espacio.

[6]Ver sección 4.9.1 en página 80.

P_y(**in(t)**) $\rightarrow$ P_x(**out(t)**).- Inicialmente, P_y no puede retirar ninguna tupla t del espacio de tuplas, lo cual causa que el proceso P_y entre en un estado de bloqueo en espera de la misma. Enseguida, P_x inserta la tupla tupla t en el espacio y permite a P_y retirarla finalmente.

En conclusión, $P_x(\text{out(t)}) \rightarrow P_y(\text{in(t)}) = P_y(\text{in(t)}) \rightarrow P_x(\text{out(t)})$.

El hecho de que FT-JavaSpaces adopte un orden secuencial en la atención de peticiones de servicio y que la comunicación y coordinación entre procesos se realicen a través de la modificación de tuplas, o variables, compartidas son dos fenómenos muy importantes que deben tenerse muy en cuenta por una aplicación basada en un espacio de tuplas.

El modelo de consistencia secuencial de FT-JavaSpaces define últimamente que las operaciones concurrentes en un espacio serán atendidas de manera como si fueran creadas por una sola entidad. Sin embargo, la decisión del orden final adoptado es una fuente importante de no determinismo característica de los espacios de tuplas.

El análisis realizado de la semántica de las operaciones concurrentes y secuenciales sobre un espacio de tuplas es de importancia porque establece las reglas básicas que deben cumplirse en cualquier de sus implementaciones.

4.10 Evolución del estado interno en el espacio de respaldo

Como se mencionó anteriormente, FT-JavaSpaces maneja un modelo de consistencia secuencial donde el orden de atención de las invocaciones lo dicta el espacio de tuplas primario.

De acuerdo con la propiedad de *orden* del criterio de secuencialidad del esquema *primario-respaldo*, las actualizaciones en una réplica de *respaldo* necesitan ser atendidas en el mismo orden utilizado en la réplica *primaria*. Sin embargo, las siguientes características de las operaciones del tipo *escritura* permiten relajar convenientemente este requerimiento en FT-JavaSpaces:

1. La actualización ($state - update$) del estado interno correspondiente al espacio de *respaldo* se realiza **después** de la correspondiente en el espacio primario, $prim(x)$. Es decir, cada actualización sigue a la atención de una operación $op(arg)$ por parte de $prim(x)$.

2. Una operación **out(t)** ingresa una tupla t en el espacio *primario* e inserta *enseguida* una copia de la misma en el espacio de *respaldo*.

3. La actualización en el espacio de respaldo se traduce únicamente en una operación del tipo in() u out(). En el caso de una operación in(*res*) en el espacio de respaldo, dicha operación busca de antemano una tupla que concuerda específicamente con el patrón *res* y que existe o está por ingresar al espacio de respaldo gracias a la característica de las operaciones tipo **out(t)** de FT-JavaSpaces (punto No. 2). En el caso de no existir todavía la tupla en el espacio de respaldo, la operación *in(res)* espera simplemente su ingreso.

De esta forma, la actualización que realiza un operador primario en el espacio de *respaldo* tiene una relación concreta con el número de tuplas en el mismo y no con la modificación de las mismas. En este sentido, el operador primario necesita solamente ingresar (out()) o retirar (in()) tuplas específicas en el espacio de *respaldo* para llevar el estado interno de éste último a uno consistente.

De manera particular y de acuerdo con las características anteriores, cualquier operación in(*res*) en el espacio de respaldo **retira** una tupla concordante con el patrón *res* hasta el momento particular de su inserción en el espacio de respaldo.

En conclusión, cualquier orden secuencial que adopte el espacio de respaldo en la atención de las peticiones respeta la secuencia de ingreso y retiro de una tupla llevada a cabo en el espacio primario.

Por otra parte, los retiros correspondientes en ambos espacios a una operación **in(t)** o **inp(t)** de FT-JavaSpaces se realizan sobre el mismo par de réplicas de una tupla. Esto debido a que el argumento, o patrón, utilizado en la operación *in(res)* en el espacio de respaldo corresponde exactamente a la tupla devuelta por la operación *in(t)* o *inp(t)* realizada en el espacio primario.

La razón por la cual un operador primario no utiliza una operación inp(t) para retirar una tupla del espacio de *respaldo* tiene que que ver con el manejo multihilos de FT-JavaSpaces.

FT-JavaSpaces no impone ninguna política específica de calendarización para los hilos de control correspondientes a cada petición. De esta forma,

las invocaciones en los espacios primario y de respaldo en los que se transforma una petición del servicio de FT-JavaSpaces no guardan orden alguno con las demás invocaciones realizadas sobre el mismo espacio. En conclusión, el orden de las invocaciones en cada uno de los espacios primario y de respaldo depende de dos factores además del orden secuencial de atención que adopten individualmente los mismos: la política de calendarización de los procesadores encargados de los Front-Ends y el orden de transmisión de mensajes utilizado por la red.

Considérese el caso de dos operaciones P_x[inp(t)] y P_y[out(t)] CONFLICTIVAS[7] del servicio de FT-JavaSpaces y ejecutadas por dos clientes distintos, P_x y P_y. La política de calendarización de procesos ("scheduling") en los Front-Ends y el orden secuencial e individual adoptado en su atención por parte de los espacios primario y secundario pueden no garantizar que el par de espacios de tuplas (primario y de respaldo) modifiquen su estado interno en el mismo orden.

Supóngase ahora que ambos espacios contienen a la tupla t y que ordenan individualmente de la siguiente forma la atención de las operaciones inp(t) y out(t), ambas del servicio de FT-JavaSpaces:

- Espacio *primario*. Considera a P_x[inp(t)]||P_y[out(t)] como out(t)→inp(t).
- Espacio de *respaldo*. Considera a P_x[inp(t)]||P_y[out(t)] como inp(t)→out(t).

En el espacio *primario*, la operación inp(t) logra retirar la tupla t insertada anteriormente por out(t). Sin embargo, el orden adoptado por el espacio de *respaldo* provoca que inp(t) no encuentre inmediatamente la tupla t y se devuelva a P_x un valor equivalente a la ausencia de dicha tupla en el espacio. Esta situación deja un estado inconsistente en el espacio de respaldo debido a que out(t) insertará la tupla t que ya no debería estar almacenada.

4.11 Manejo de fallos

Las capas funcionales de un *Front-End* colaboran de la siguiente manera:

[7] Es decir, utilizan un mismo patrón como argumento y se ejecutan de manera concurrente.

1. El *representante en el cliente* (proxy) recibe una petición y la envía al *receptor de invocaciones.*

2. El *receptor de invocaciones* (RDI) delega la invocación a un *manejador confiable de operaciones* específico.

3. El *manejador confiable de operaciones* (MCO) utiliza un *operador primario* para interactuar inicialmente con el espacio *primario* y, en su caso, actualizar posteriormente el estado interno del de *respaldo*[8] (ver figura 4.5).

   Si el *operador primario* detecta un fallo en el espacio *primario*, el *manejador confiable de operaciones* utiliza un *operador secundario* para interactuar de manera exclusiva con el espacio de *respaldo* (ver figura 4.7). En este momento, el espacio de *respaldo* toma el lugar del *primario.* La forma en que el operador primario (OP) anuncia la presencia de un fallo en el espacio/canal *primario* es mediante el lanzamiento de una *excepción* (Modo reducido).

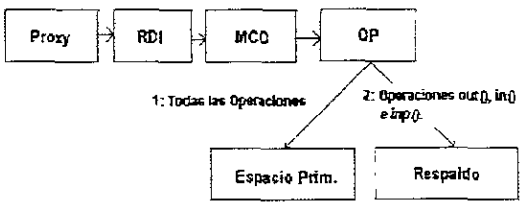

Figura 4.5: Colaboración entre las capas funcionales de un *Front-End.*

Los operadores primario y secundario son capaces de detectar una caída en los espacios y canales de comunicación primario y de respaldo gracias a que los procesadores en el ambiente y el protocolo de comunicación utilizado entre un Front-End y los espacios son ambos considerados del tipo Fail-Stop.

La detección de fallos del tipo omisión en los servidores que implementan los espacios de tuplas primario y de respaldo se asume con la terminación de un lapso de tiempo de espera ("time-out") para el recibimiento de un

[8]El estado interno del espacio de *respaldo* es necesario actualizarlo solamente en el caso de una operación out(), in() o inp().

mensaje ("acknowledge") especial enviado por los mismos y que confirme la atención en marcha de una petición. El cumplimiento de un lapso de espera sin respuesta provoca que el Front-End asuma un fallo de omisión en el servidor.

4.11.1 Fallo en el espacio de respaldo

En el caso de presentarse un fallo en el espacio o canal de *respaldo*, el operador primario lleva solamente a cabo su función en el espacio *primario* (Modo reducido). Es decir, el Front-End considera únicamente y a partir de ahora al espacio primario como el único en funcionamiento. El criterio de *secuencialidad* toma en cuenta solamente las *réplicas de respaldo* funcionales[9]. Por esta razón, el espacio *primario* es el único involucrado en cualquier petición subsecuente (ver figura 4.6).

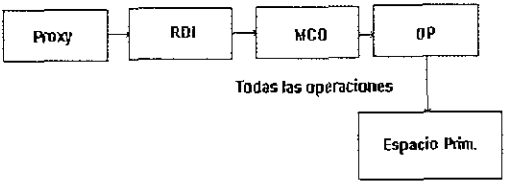

Figura 4.6: Colaboración entre las capas de un *Front-End* con una falla en el espacio secundario.

4.11.2 Fallo en el espacio primario

Todas las peticiones de servicio realizadas a FT-JavaSpaces utilizan siempre e inicialmente un operador primario. En el caso de un fallo en el espacio o canal primario, el manejador confiable de operaciones intenta utilizar primeramente un operador primario y crea después un operador secundario para interactuar única y finalmente con el espacio de respaldo (ver figura 4.7).

Un *manejador confiable de operaciones* continúa empleando un *operador secundario* hasta el momento de completarse el algoritmo de recuperación del espacio *primario*, el cual se explicará más adelante. A partir de este

[9]Es decir, sin fallos.

momento, comienza a utilizar de nuevo un *operador primario*.

Por otra parte, siempre que existan en funcionamiento ambos espacios, una misma maáquina será siempre la designada para actuar como espacio primario. Esto quiere decir que una vez que se recupera un espacio primario, éste vuelve a asumir el rol de espacio primario.

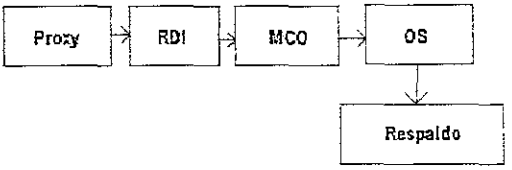

Figura 4.7: Colaboración con la utilización del operador secundario.

En el momento de que el espacio primario experimenta un fallo es necesario garantizar que todos los `operadores primarios` en proceso de *retirar* una tupla del espacio de respaldo terminen antes de que se creen nuevos `operadores primarios` correspondientes a nuevas peticiones de servicio y de que los `operadores secundarios` recién creados entren en acción. La única excepción es la operación out() donde sus operadores primario y de respaldo no requieren revisar ninguna condición y los cuales necesitan actualizar inmediatamente el estado interno de los espacios que involucran. En otras palabras, los operadores primarios existentes que lograron retirar exitosamente una tupla del espacio primario antes de que éste último experimentara un fallo deben hacer lo mismo con el espacio de respaldo **antes** de que cualquier operador secundario subsecuente modifique directamente el estado interno del espacio de respaldo mediante el retiro de una tupla. De otra forma, un operador secundario podría retirar anticipadamente una tupla del espacio de respaldo requerida específicamente por un operador primario y dejar potencialmente éste último en un un estado de bloqueo.

En consecuencia. una vez que un manejador confiable de operaciones (MCO), con excepción al de las operaciones out(), detecta un fallo en el espacio primario, la siguiente secuencia de pasos debe llevarse a cabo:

1. Avisar del fallo en el espacio primario para que los nuevos `operadores primarios` esperen a que los `operadores primarios` actuales en pro-

ceso de actualizar el espacio de respaldo terminen.

2. Crear un operador secundario.

3. Esperar también a que los operadores primarios actuales en proceso de actualizar el espacio de respaldo finalicen.

Una vez que terminan su trabajo todos los operadores primarios que no se dieron cuenta del fallo en espacio primario y se encontraban en proceso de retirar una tupla del de respaldo es necesario entonces que todos los operadores primarios y secundarios en espera reanuden su función. A partir de este momento, el espacio de respaldo se convierte en el primario y el único en funcionamiento.

4.11.3 Semántica de fallos de FT-JavaSpaces

En FT-JavaSpaces, los fallos que pueden presentarse en su ambiente de implementación son los mismos que tolera. Es decir, omisión del servidor y caída del procesador o canal de comunicación.

Fuera de los fallos y el número de los mismos que tolera FT-JavaSpaces en su ambiente de implementación, existen situaciones que llevan a FT-JavaSpaces a experimentar en si un fallo del tipo caída.

En particular, FT-JavaSpaces experimenta una caída cuando el número de fallos que puede tolerar es mayor a uno de manera simultánea.

FT-JavaSpaces es un sistema del tipo *fail-stop*; es decir, los clientes del mismo son capaces de detectar su caída mediante la recepción de una *excepción*.

Nótese como la semántica de fallos de FT-JavaSpaces es más fuerte que la de una implementación de un espacio de tuplas no tolerante a ningún tipo de fallo.

Caída de FT-JavaSpaces

Si el espacio/canal de *respaldo* experimenta un fallo, un operador *secundario* anuncia una *excepción* a su *MCO* respectivo y FT-JavaSpaces experimenta un fallo del tipo caída (Modo con fallo).

De esta forma, FT-JavaSpaces puede experimentar un solo tipo de fallo:

Caída de servidor Si en un mismo instante de tiempo, los dos espacios de tuplas (primario y de respaldo)

- experimentan un fallo,
- se encuentran en proceso de reinicialización o
- incorporándose a la red,

entonces el *servidor*[10] de FT-JavaSpaces experimenta un fallo del tipo caída.

4.12 Estrategia de recuperación

Una vez corregida la falla en el canal de comunicación o en el espacio primario o de respaldo, el espacio de tuplas que experimentó el fallo debe primeramente ser reinicializado. A continuación y sin interrumpir el servicio, FT-JavaSpaces *transfiere* la última versión actualizada y consistente del estado interno del espacio en operación al espacio recién restaurado.

El algoritmo de recuperación de un espacio de tuplas sigue la siguiente secuencia de *pasos*:

1. FT-JavaSpaces continúa *recibiendo* nuevas peticiones de servicio, pero *suspende* su atención (*delivery*) hasta la terminación del traspaso de estado interno.

2. Las operaciones de los tipos **in()** y **rd()** atendidas actualmente por FT-JavaSpaces son *interrumpidas* y consideradas como recién recibidas[11]. La *interrupción* de una operación en proceso de atención no le permite modificar el estado interno de ninguno de los espacios ni tampoco devolver resultado alguno.

3. FT-JavaSpaces queda en un estado de espera hasta que no exista ninguna operación **out()**, **inp()** o **rdp()** en proceso de atención.

4. Una vez atendidas todas las operaciones de los tipos **out()**, **inp()** y **rdp()**, FT-JavaSpaces traspasa el estado interno del espacio de tuplas funcional a un nuevo espacio o al recién restaurado.

[10] ver sección 4.5 en página 70.

[11] Es decir, son tratadas como en el paso 1.

5. FT-JavaSpaces reanuda todas las operaciones *suspendidas* en los pasos número uno y dos.

Este algoritmo funciona tanto en la versión centralizada como en la distribuida.

Con un solo Front-End en la implementación de FT-JavaSpaces, el receptor de invocaciones (RDI) es la responsable de comenzar el algoritmo de recuperación de uno de los dos espacio de tuplas que haya sufrido un fallo y que se encuentre listo para incorporarse de vuelta. En una configuración distribuida, el receptor de invocaciones (RDI) monitorea constantemente si está por iniciarse la recuperación de un espacio para realizar su parte en el algoritmo de recuperación.

4.13 Soporte externo

El soporte externo que recibe FT-JavaSpaces depende del tipo de causa que haya provocado un fallo en su ambiente de operación.

FT-JavaSpaces es capaz de iniciar automáticamente operaciones o la recuperación de uno de sus espacios de tuplas cuando es necesario únicamente encender o reiniciar las computadores que guardan los espacios de tuplas primario y de respaldo.

4.14 Discusión

El objetivo de este capítulo consistió en presentar el diseño de la propuesta de este trabajo, denominada FT-JavaSpaces, para un espacio de tuplas de alta confiabilidad y disponibilidad.

El principio abarca los supuestos principales acerca del ambiente de implementación de FT-JavaSpaces, el conjunto de fallos posibles y la semántica de fallos del mismo FT-JavaSpaces. En este sentido, FT-JavaSpaces es tolerante a tres tipos de fallos en una implementación única de un espacio de tuplas: caída del procesador y caída del canal de comunicación y omisión del servidor. Los fallos que tolera FT-JavaSpaces son considerados los más susceptibles de aparecer en una implementación de un espacio de tuplas.

En la práctica, FT-JavaSpaces utiliza como estrategia principal de tolerancia a fallas un par de servidores idénticos que implementan un servicio de espacio de tuplas. El servicio de FT-JavaSpaces se mantiene mientras exista al menos una implementación en funcionamiento y el mecanismo de manejo del grupo es completamente transparente a los usuarios. De esta forma, la implementación de FT-JavaSpaces es indistinguible de cualquier otra, lo cual es uno de los objetivos deseables en cualquier sistema tolerante a fallas.

En el capítulo se presentan también el manejo que hace del sistema FT-JavaSpaces sin la presencia de los fallos que puede tolerar y en el caso de la aparición de los mismos. Sin la aparición de dichos fallos, FT-JavaSpaces garantiza en todo momento una evolución consistente de las réplicas. En el caso de aparecer un fallo del tipo que tolera, FT-JavaSpaces permite enmascararlo y seguir prestando el servicio.

Además, FT-JavaSpaces cuenta con un algoritmo para la recuperación de una de sus implementaciones de un espacio de tuplas y ejecutarlo de manera transparente para sus clientes.

En conclusión, FT-JavaSpaces es un desarrollo que cuenta con el mínimo número de características deseables para aumentar la confiabilidad y disponibilidad de una implementación de un espacio de tuplas si se consideran a los tres tipos de fallos que tolera.

```
//RETIRO DE LA TUPLA CORRESPONDIENTE DEL ESPACIO PRIMARIO
result = espacio_primario.take(tmpl);

//RETIRO DE LA TUPLA CORRESPONDIENTE DEL ESPACIO DE RESPALDO
result = espacio_de_respaldo.take(result);

//DEVOLUCION DE RESULTADO;
return result;
}
```

Pseudocódigo 5.3.2 (OP de out())

```
public Lease write_(Entry tmpl) throws
   RemoteExceptionPrimary {

//INSERCION DE LA TUPLA CORRESPONDIENTE EN EL ESPACIO PRIMARIO
espacio_primario.write(tmpl);

//INSERCION DE LA TUPLA CORRESPONDIENTE EN EL ESPACIO DE RESPALDO
espacio_de_respaldo.write(tmpl);

}
return;
```

Por el contrario, la invocación de inp() en FT-JavaSpaces utiliza las operaciones equivalentes a inp() e in() de *JavaSpaces* en los espacios primario y de respaldo, respectivamente (ver pseudocódigo 5.3.3).

Pseudocódigo 5.3.3 (OP de inp())

```
public Entry takeif_(Entry tmpl) throws RemoteExceptionPrimary,
 InterruptedException {

//RETIRO DE LA TUPLA CORRESPONDIENTE DEL ESPACIO PRIMARIO
result = (Entry)espacio_primario.takeIfExists(tmpl);

//INICIO DEL ALGORITMO EN EL ESPACIO DE RESPALDO
  //SI EXISTE LA TUPLA CORRESPONDIENTE EN EL ESPACIO PRIMARIO
  if(result != null){
   try  {
   // RETIRO DE LA TUPLA CORRESPONDIENTE DEL ESPACIO DE RESPALDO
    result = (Entry)espacio_de_respaldo.take(result);
```

```
   }fi
  }

//DEVOLUCION DE RESULTADO
 return result;
}
```

Operador secundario de las Ops. del tipo escritura

Los operadores secundarios de las operaciones in(), inp() y out() de FT-JavaSpaces realizan simplemente sus invocaciones respectivas en el nuevo y único espacio *primario*, o implementación de *JavaSpaces* (ver pseudocódigo 5.3.4).

Pseudocódigo 5.3.4 (OS de inp(), in() y out())

```
public OP_out,inp,in(Entry e) throws RemoteExceptionFatal,
 InterruptedException{

 //TAKE, TAKEIF o WRITE EN EL NUEVO Y UNICO ESPACIO PRIMARIO
  backupresult=(Entry/Lease)espacio_de_respaldo.take/takeif/write(e);

 //DEVOLUCION DE RESULTADO
 return this.backupresult;

}//fin metodo
```

Representantes de las Ops. del tipo lectura

Los representantes primario y secundario de las operaciones rd() y rdp() de FT-JavaSpaces se incluyen en esta categoría.

Operador primario de las Ops. del tipo lectura

El operador primario de estas operaciones realiza una única invocación en la implementación primaria de *JavaSpaces*, o espacio *primario* (ver pseudocódigo 5.3.5).

Pseudocódigo 5.3.5 (OP de rd() y rdp())

```
public Entry read/readIfExists(Entry tmpl) throws
 RemoteExceptionPrimary, InterruptedException {

        //READ O READIFEXISTS EN EL ESPACIO PRIMARIO
        result = (Entry)espacio_primario.read/readIfExists(tmpl);
```

Capítulo 5

Implementación

Este capítulo presenta la implementación de FT-JavaSpaces y describe la estructura y funcionamiento de sus partes desde el punto de vista de las herramientas de programación utilizadas.

5.1 Introducción

FT-JavaSpaces no implementa en si un espacio de tuplas sino que toma dos implementaciones del mismo para sus espacios primario y de respaldo y crea los mecanismos de tolerancia a fallas para manejarlos.

En particular, los mecanismos implementados por FT-JavaSpaces radican en la capa del manejo de los espacios primario y de respaldo. De esta forma, el manejo de la política de tolerancia a fallas de FT-JavaSpaces se desarrolla específicamente en sus Front-Ends.

5.2 JavaSpaces

En FT-JavaSpaces, el nombre específico del servicio que ofrecen los espacios de tuplas es *JavaSpaces* [ea99]. *JavaSpaces* es una versión orientada a objetos de Linda y parte del lenguaje de programación JAVA. JAVA es un lenguaje multiplataforma y *JavaSpaces* es en particular un servicio de distribuido, en el sentido que puede ser empleado por otras máquinas por medio de una conexión en red.

JavaSpaces comparte con Linda la características de almacenar conjuntos de datos para su posterior procesamiento y un manejo de los mismos

basado en su contenido. Por otra parte, las diferencias significativas entre ambos son:

- El conjunto de tipos de datos disponibles en *JavaSpaces* es mayor. A diferencia de Linda, *JavaSpaces* permite contar diferentes tipos de tuplas. De esta forma, dos tuplas distintas con el mismo número y tipo de campos pueden representar dos abstracciones distintas (e.g., un vector o un punto)

- Una tupla en *JavaSpaces* es manejada tal como un objeto con sus atributos y métodos asociados. Es decir, las tuplas tienen potencialmente también un comportamiento.

- En *JavaSpaces*, la herencia del paradigma orientado a objetos permite que un patrón concuerde con una tupla perteneciente a una subclase.

- Los campos de una tupla en *JavaSpaces* son cualquier objeto del lenguaje de programación JAVA.

- *JavaSpaces* no implementa la operación eval(). Según sus especificaciones, eval() no es incluida por considerar que la misma necesita de la solución de problemas (e.g., seguridad) y ofrecer garantías de servicio (e.g., "fairness") adicionales en la realización de computaciones arbitrarias a petición de los clientes.

`FT-JavaSpaces` conserva estas diferencias con Linda.

5.2.1 Tuplas

JavaSpaces almacena concretamente *entradas* o *entries*. Una *entrada*, o *entry*, es una colección de referencias a objetos de una clase definida y que implementa la interfaz[1] `net.jini.core.Entry`.

Por ejemplo, la forma de crear una *entrada* del tipo "Mensaje" y con un campo del tipo cadena de caracteres ("String") - llamado *contenido* - es de la forma siguiente:

```
public class Mensaje implements Entry {
    public String contenido;
```

[1]La palabra clava *interfaz* sirve para especificar un conjunto de métodos que pueden ser implementados por una o más clases.

```
    public Mensaje () {
    }
}
```

La forma de instanciar un *entrada* del tipo Mensaje y con el contenido "Te espero a las 7." se muestra a continuación:

```
Mensaje msg = new Mensaje();
msg.contenido = ''Te espero a las 7.'';
```

5.2.2 Operaciones

JavaSpaces implementa cinco de las operaciones básicas de Linda, con excepción de eval(). La tabla 5.1 resume la correspondencia entre las operaciones que ofrece *JavaSpaces* y sus versiones equivalentes en Linda.

Linda	JavaSpaces
out(N, P_2, ..., P_j)	write(Entry entry, ...)
in(N, P_2, ..., P_j)	take(Entry template, ...)
rd(N, P_2, ..., P_j)	read(Entry template, ...)
inp(N, P_2, ..., P_j)	takeIfExists(Entry template, ...)
rdp(N, P_2, ..., P_j)	readIfExists(Entry template, ...)
eval(N, P_2, ..., P_j)	No incluida

Tabla 5.1: Correspondencia entre operaciones de Linda y *JavaSpaces*.

5.2.3 Soporte de tolerancia a fallas

Todas las operaciones que exporta *JavaSpaces* son invocadas en un objeto que implementa la interfaz JavaSpace:

```
public interface JavaSpace {

 Lease write(Entry e, ...)
  throws RemoteException;

 Entry read(Entry tmpl, ...)
  throws UnusableEntryException, RemoteException,
         InterruptedException;
```

```
Entry readIfExists(Entry tmpl, ...)
 throws UnusableEntryException, RemoteException,
        InterruptedException;

Entry take(Entry tmpl, ...)
 throws UnusableEntryException, RemoteException,
        InterruptedException;

Entry takeIfExists(Entry tmpl, ...)
 throws UnusableEntryException, RemoteException,
        InterruptedException;
}
```

La interfaz *JavaSpace* es implementada por un objeto representante ("proxy") del servidor en el cliente mismo. Este representante se comunica con el servidor por medio de un servicio de llamadas a procedimientos remotos (RPC) particular de JAVA y denominado RMI (i.e., Remote Method Invocation).

La definición de los métodos correspondientes a cada operación señalan una serie de excepciones posibles en la invocación de las mismas. A continuación, se presenta la descripción de las dos excepciones referentes a fallos en el servicio de *JavaSpaces* y una con relación al hilo de control de quien la invoca:

RemoteException Los fallos que ocasionan el lanzamiento de esta excepción son:

1. Una caída del canal de comunicación o del servidor. En particular, ocasionada por la falta de recursos computacionales o fallas de naturaleza física.
2. Un fallo de omisión en el servidor. En este caso, la causa particular del mismo es la detección de un error durante la realización de los procedimientos de "marshalling" o "unmarshalling"[2] de los objetos transmitidos en la invocación de un método remoto.

[2]Los procedimientos de "marshalling" y "unmarshalling" se encargan de preparar los objetos para su envío y recepción en el proceso de la invocación de un método remoto, respectivamente.

UnusableEntryException *JavaSpaces* utiliza un procedimiento denominado "serialization" que convierte a un objeto en un arreglo lineal de bytes para su almacenamiento en memoria y transmisión a través de la red. El procedimiento inverso es llamado "deserialization". La excepción *UnusableEntryException* es lanzada si este último procedimiento encuentra un error en alguno de dichos arreglos lineales; situación que se traduce en un fallo de omisión en el servidor. De manera específica, notifica la presencia de un error en el estado interno del servicio.

InterruptedException Este excepción es lanzada cuando el hilo de control que ejecuta un método es interrumpido de alguna forma; en particular, con la invocación de su método `Thread.interrupt()`. En la situación de su lanzamiento y cuando se trata de una operación take() o takeIfExists, ninguna *entrada* es retirada de forma garantizada del espacio.

Nótese como el tipo de fallos correspondientes a las excepciones son los mismo abordados en las especificaciones (ver tabla 5.2). El mismo fenómeno sucede con la suposición acerca de la interrupción de una operación en el algoritmo de recuperación de un espacio de tuplas y la excepción *InterruptedException*.

Fallos Abordados	Excepciones
Omisión del Servidor	UnusableEntry Exception
Omisión del Servidor	RemoteException
Caída del Procesador	RemoteException
Caída del Canal de Com.	RemoteException

Tabla 5.2: Correspondencia entre excepciones y tipos de fallos abordados.

5.3 Implementación de la política de tolerancia a fallas

En concordancia con el diseño de FT-JavaSpaces, los paquetes principales (ver figura 5.1) que componen a un Front-End, encargado de la política de tolerancia a fallas, se dividen en cuatro grupos:

- Representante en el cliente (Proxy). Estos paquetes contienen la interfaz necesaria para acceder al servicio de `FT-JavaSpaces` y se ubican

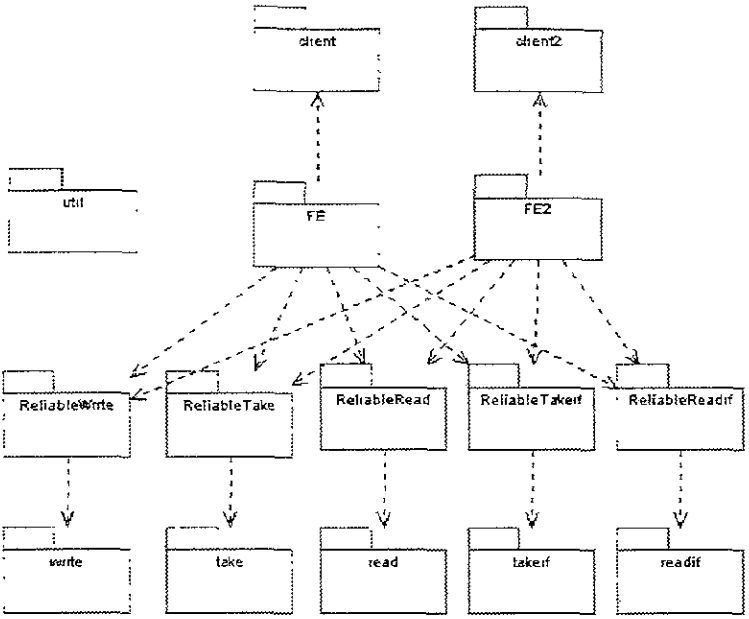

Figura 5.1: Paquetes de FT-JavaSpaces.

en el cliente.

- ftlinda.client - Interfaz para la versión centralizada.
- ftlinda.client2 - Interfaz para la versión distribuida.

- Receptor de invocaciones (RDI). Estos paquetes contienen las clases del receptor de invocaciones.
  - ftlinda.fe - Clases para el RDI de la versión centralizada.
  - ftlinda.fe2 - Clases para el RDI de la versión distribuida.

- Manejadores confiables de operaciones (MCO). Los paquetes en este grupo contienen las clases respectivas a cada uno de los cinco manejadores confiables de operaciones.
  - ftlinda.reliableWrite - Clases para el MCO de la operación out().
  - ftlinda.reliableTake - Clases para el MCO de la operación in().
  - ftlinda.reliableRead - Clases para el MCO de la operación rd().

- ftlinda.reliableTakeif - Clases para el MCO de la operación inp().
- ftlinda.reliableReadif - Clases para el MCO de la operación rdp().

- Representantes del grupo (OP y OS). Estos paquetes guardan las clases respectivas a los operadores primario (OP) y secundario (OS) necesarios para cada operación.

  - ftlinda.write - Clases del OP y OS de la operación out().
  - ftlinda.take - Clases del OP Y OS de la operación in().
  - ftlinda.takeif - Clases del OP y OS de la operación inp().
  - ftlinda.read - Clases del OP Y OS de la operación rd().
  - ftlinda.readif - Clases del OP Y OS de la operación rdp().

Además, FT-JavaSpaces cuenta con otros paquetes adicionales en su estructura:

1. ftlinda.stateTransfer - Este paquete contiene las clases necesarias para el traspaso de estado interno de una implementación de *JavaSpaces* a otra.

2. ftlinda.util - Las clases en este paquete sirven para establecer comunicación con las implementaciones de *JavaSpaces*, el manejo de los hilos de control, el monitoreo del traspaso de estado interno, la definición de *entradas* con funciones especiales, utilerias, etc.

3. ftlinda.pegasoRecuperation - Contiene todas las clases necesarias para la recuperación del espacio de tuplas primario en la versión distribuida de FT-JavaSpaces. Este paquete es específico para el espacio primario y reside en la misma máquina del mismo.

4. ftlinda.debianRecuperation - Contiene las clases equivalentes del paquete *ftlinda.debianRecuperation* para el espacio de respaldo.

5. ftlinda.inicializacion - Las clases en este paquete se encargan de ingresar *entradas* especiales en el par de espacios de tuplas antes de que FT-JavaSpaces comience a ofrecer su servicio. Dichas *entradas* son utilizadas en el algoritmo de traspaso de estado interno en su versión distribuida.

Por último, la figura 5.2 muestra el diagrama general de clases que forman la estructura de FT-JavaSpaces.

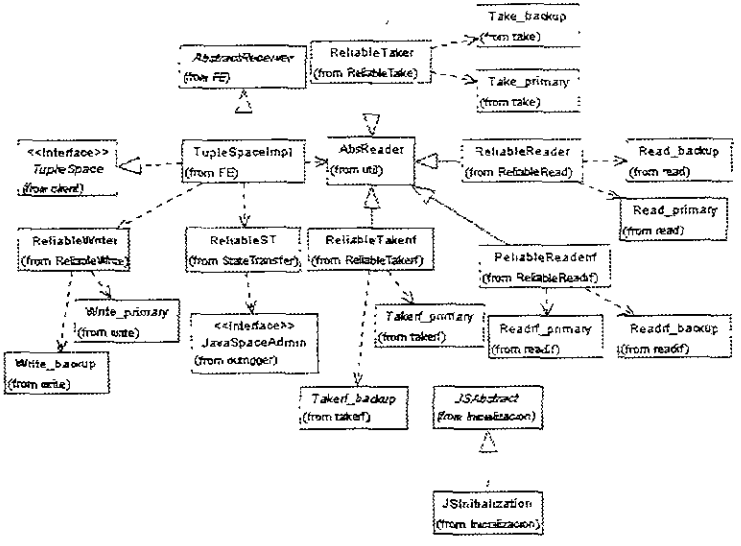

Figura 5.2: Diagrama general de clases de FT-JavaSpaces.

5.3.1 Representantes del grupo

Los representantes del grupo se dividen en primario (OP) y secundario (OS). Además, existen representantes especiales para las operaciones del tipo *escritura* y *lectura*.

Representantes de Ops. del tipo escritura

Esta categoría abarca los representantes primario y secundario para las operaciones in(), inp() y out() de FT-JavaSpaces.

Operador primario de las Ops. del tipo escritura

Los operadores primarios de las operaciones in() y out() de FT-JavaSpaces se caracterizan por utilizar un mismo tipo de invocación en ambas implementaciones de *JavaSpaces*. El algoritmo de los *OP*s para las operaciones in() y out() se muestra en los pseudocódigos 5.3.1 y 5.3.2, respectivamente.

Pseudocódigo 5.3.1 (OP de in())

```
public Entry take_(Entry tmpl) throws RemoteExceptionPrimary,
                                      InterruptedException {
```

```
//RETIRO DE LA TUPLA CORRESPONDIENTE DEL ESPACIO PRIMARIO
result = espacio_primario.take(tmpl);

//RETIRO DE LA TUPLA CORRESPONDIENTE DEL ESPACIO DE RESPALDO
result = espacio_de_respaldo.take(result);

//DEVOLUCION DE RESULTADO;
return result;
}
```

Pseudocódigo 5.3.2 (OP de out())

```
public Lease write_(Entry tmpl) throws
   RemoteExceptionPrimary {

//INSERCION DE LA TUPLA CORRESPONDIENTE EN EL ESPACIO PRIMARIO
espacio_primario.write(tmpl);

//INSERCION DE LA TUPLA CORRESPONDIENTE EN EL ESPACIO DE RESPALDO
espacio_de_respaldo.write(tmpl);

}
return;
```

Por el contrario, la invocación de inp() en FT-JavaSpaces utiliza las operaciones equivalentes a inp() e in() de *JavaSpaces* en los espacios primario y de respaldo, respectivamente (ver pseudocódigo 5.3.3).

Pseudocódigo 5.3.3 (OP de inp())

```
public Entry takeif_(Entry tmpl) throws RemoteExceptionPrimary,
 InterruptedException {

//RETIRO DE LA TUPLA CORRESPONDIENTE DEL ESPACIO PRIMARIO
result = (Entry)espacio_primario.takeIfExists(tmpl);

//INICIO DEL ALGORITMO EN EL ESPACIO DE RESPALDO
  //SI EXISTE LA TUPLA CORRESPONDIENTE EN EL ESPACIO PRIMARIO
  if(result != null){
   try  {
   // RETIRO DE LA TUPLA CORRESPONDIENTE DEL ESPACIO DE RESPALDO
    result = (Entry)espacio_de_respaldo.take(result);
```

```
   }fi
  }

//DEVOLUCION DE RESULTADO
 return result;
}
```

Operador secundario de las Ops. del tipo escritura

Los operadores secundarios de las operaciones in(), inp() y out() de FT-JavaSpaces realizan simplemente sus invocaciones respectivas en el nuevo y único espacio *primario*, o implementación de *JavaSpaces* (ver pseudocódigo 5.3.4).

Pseudocódigo 5.3.4 (OS de inp(), in() y out())

```
public OP_out,inp,in(Entry e) throws RemoteExceptionFatal,
 InterruptedException{

 //TAKE, TAKEIF o WRITE EN EL NUEVO Y UNICO ESPACIO PRIMARIO
  backupresult=(Entry/Lease)espacio_de_respaldo.take/takeif/write(e);

 //DEVOLUCION DE RESULTADO
 return this.backupresult;

}//fin metodo
```

Representantes de las Ops. del tipo lectura

Los representantes primario y secundario de las operaciones rd() y rdp() de FT-JavaSpaces se incluyen en esta categoría.

Operador primario de las Ops. del tipo lectura

El operador primario de estas operaciones realiza una única invocación en la implementación primaria de *JavaSpaces*, o espacio *primario* (ver pseudocódigo 5.3.5).

Pseudocódigo 5.3.5 (OP de rd() y rdp())

```
public Entry read/readIfExists(Entry tmpl) throws
 RemoteExceptionPrimary, InterruptedException {

        //READ O READIFEXISTS EN EL ESPACIO PRIMARIO
        result = (Entry)espacio_primario.read/readIfExists(tmpl);
```

```
      //DEVOLUCION DE RESULTADO
       return result;

}//fin metodo take_
```

Operador secundario de las Ops. del tipo lectura

Los operadores secundarios de rd() y rdp() actúan sobre la nueva implementación *primaria* de *JavaSpaces* y recién designada como tal (ver pseudocódigo 5.3.6).

Pseudocódigo 5.3.6 (OS de rd() y rdp())

```
public Entry read/readIfExists(Entry tmpl) throws
 RemoteExceptionFatal, InterruptedException {

//READ O READIFEXISTS EN EL NUEVO Y UNICO ESPACIO PRIMARIO
 backupresult = (Entry)espacio_de_respaldo.read/readIfExists(tmpl);

//DEVOLUCION DE RESULTADO
 return backupresult;

}//fin metodo take_
```

5.3.2 Manejador confiable de operaciones

Los manejadores confiables de operaciones son cinco en FT-JavaSpaces y corresponden a cada una de sus operaciones. Una manejador confiable de operaciones (MCO) *implementa* la *interfaz* `Runnable` para trabajar en un hilo de control independiente. Además, maneja dos objetos `X_primary` y `X_backup` equivalentes los operadores primario y secundario, respectivamente[3]. El pseudocódigo 5.3.7 muestra el algoritmo genérico para el MCO de una operación.

Pseudocódigo 5.3.7 (Manejador confiable de operación)

```
public class ReliableWriter implements java.lang.Runnable {

//INICIO DE HILO DE CONTROL
 public void run(){
```

[3]Donde X es igual a write, take, takeIfExists, read y readIfExists

```
  try {
// CREACION DE UN OPERADOR PRIMARIO
     Operador_primario primary = new Operador_primario();
     primary.write/read/readIfExists/take/takeIfExists(template);

    //SI EL ESPACIO PRIMARIO SUFRE UN FALLO SE CREA UNA EXCEPCION
    //QUE OBLIGA A INTERACTUAR UNICAMENTE CON EL ESPACIO DE
    //RESPALDO
    } catch (RemoteExceptionPrimary e0) {
      try {
    // CREACION DE UN OPERADOR SECUNDARIO
         Operador_secundario backup = new Operador_secundario();
         backup.write/read/readIfExists/take/takeIfExsts(template);

    //SI EL ESPACIO SECUNDARIO SUFRE UN FALLO SE CREA UNA EXCEPCION
    //QUE ANUNCIA LA TERMINACION DEL SERVICIO DE FT-JAVASPACES
      } catch (RemoteExceptionFatal e1) {
                System.out.println("*****************");
                System.out.println("FAILURE IN SERVICE");
                System.out.println("*****************");
                failure = true;
      //ENVIO DE LA EXCEPCION ESPECIFICA AL CLIENTE;
                throw new RemoteException();
      }
  }
 }//FIN DE HILO DE CONTROL
}
```

5.3.3 Receptor de invocaciones

El receptor de invocaciones (RDI) implementa todas las operaciones que ofrece el servicio de FT-JavaSpaces y, en general, la interfaz *TupleSpace* (ver pseudocódigo 5.3.8).

Pseudocódigo 5.3.8 (Receptor de Invocaciones (RDI))

```
public class TupleSpaceImpl extends AbstractReceiver implements
ftlinda.client.TupleSpace {
  ...
}
```

5.3.4 Representantes en el cliente

El elemento importante en los paquetes `ftlinda.client` y `ftlinda.client2` es un interfaz que sirve a los clientes para utilizar el servicio de FT-JavaSpaces. Dicha interfaz reside en la máquina del cliente y sirve de referencia a un *receptor de invocaciones* (RDI) local o distribuido.

ftlinda.client.TupleSpace

La interfaz TupleSpace contiene cinco métodos correspondientes a las operaciones de FT-JavaSpaces y uno para la recuperación de cada espacio de tuplas:

```
package ftlinda.client;

public abstract interface TupleSpace extends java.rmi.Remote {
    public Lease out(Entry tuple) throws RemoteException;
    public Entry in(Entry template) throws RemoteException;
    public Entry rd(Entry template) throws RemoteException;
    public Entry inp(Entry template) throws RemoteException;
    public Entry rdp(Entry template) throws RemoteException;
    public void STtoPegaso() throws RemoteException;
    public void STtoDebian() throws RemoteException;
}
```

Esta interfaz es la empleada en la versión centralizada de FT-JavaSpaces.

ftlinda.client2.TupleSpace

La interfaz `ftlinda.client2.TupleSpace` ofrece igualmente los cinco métodos principales de FT-JavaSpaces en su versión distribuida. A diferencia de `thesis.client.TupleSpace`, no requiere exportar los métodos de recuperación de un espacio debido a que su invocación se encuentra fuera del alcance de los clientes y es responsabilidad de un objeto especializado.

```
package ftlinda.client2;

public abstract interface TupleSpace extends java.rmi.Remote {

    public Lease out(Entry tuple) throws RemoteException;
    public Entry in(Entry template) throws RemoteException;
    public Entry rd(Entry template) throws RemoteException;
    public Entry inp(Entry template) throws RemoteException;
    public Entry rdp(Entry template) throws RemoteException;
}
```

5.3.5 ftlinda.stateTransfer

Este paquete guarda la clase ReliableST encargada del traspaso de estado interno del espacio *primario* en **modo reducido** de operación a un espacio nuevo o recuperado.

ReliableST

Esta clase cuenta igualmente con un método run() para iniciar el traspaso de estado interno en un hilo de control independiente.. La acción que realiza es prácticamente una iteración a través de todas las tuplas del espacio primario e insertar una copia de cada una en el espacio cuyo estado interno hay que actualizar (ver pseudocódigo 5.3.9).

Pseudocódigo 5.3.9 (Algoritmo de la clase ReliableST)

```
public class ReliableST implements java.lang.Runnable {

//INICIO DE HILO DE CONTROL
 public void run() {

//COMIENZA EL TRASPASO DE ESTADO
   try {
    JavaSpaceAdmin jsadmin = (JavaSpaceAdmin)
             ((Administrable)espacio_sin_fallo).getAdmin();

  //ITERACION A TRAVES DE TODAS LAS TUPLAS Y ESCRITURA EN EL ESPACIO
  //RECIEN INCORPORADO
    iterator = jsadmin.contents(null,null);
    int c = 0;
    do {
        //LECTURA EN EL ESPACIO ORIGEN
        e = iterator.next();
        if (e != null) {
         //ESCRITURA EN EL ESPACIO DESTINO
         espacio_a_actualizar.write(e);
         c++;
        }
    //MIENTRAS EXISTAN TUPLAS QUE TRASPASAR
    } while(e != null);
    iterator.close();
   } catch (Exception e) {
       System.out.println(e);
       //FALLO EN ALGUNO DE LOS ESPACIOS DURANTE EL TRASPASO
```

```
        this.failure = true;
    }
} //FIN DEL HILO DE CONTROL
```

5.3.6 ftlinda.util

`ftlinda.util` guarda clases con usos variados. De manera específica, las clases `GuardedST` y `GuardedWC` son de utilidad en el algoritmo de recuperación de un espacio. `Processes`, `BarrierSynch` y `StateTransferONOFF` son *entradas* del sistema utilizadas en la versión distribuida de `FT-JavaSpaces`.

GuardedST

La clase `GuardedST` es empleada básicamente por el algoritmo de recuperación, en sus dos versiones, para notificar su inicio y terminación al receptor de invocaciones.

GrdTran y GuardedT

Las clases GrdTran y GuardedT sirven en la transición de rol de espacio de respaldo a primario en las versiones centralizada y distribuida de FT-JavaSpaces, respectivamente. Ambas clases llevan una cuenta de los operadores primarios existentes en el momento de producirse un fallo en el espacio primario y detienen la interacción con el nuevo espacio primario de las invocaciones nuevas y de los operadores secundarios actuales hasta que no exista ningún operador primario en funcionamiento.

GuardedWC

En `FT-JavaSpaces`, una instancia de la clase `GuardedWC` lleva el control de las operaciones out(), rdp() e inp() en ejecución actualmente. El algoritmo de recuperación espera hasta el momento en que dicha instancia indique que no existe ninguna atención correspondiente a dichas operaciones y así poder comenzar el traspaso.

Processes

La clase `Processes` define una *entrada* que maneja el número de *Front-Ends* existentes en una implementación distribuida de `FT-JavaSpaces`. Una instancia de esta clase de *entrada* ingresa en el grupo redundante antes de comenzar a funcionar el servicio de `FT-JavaSpaces`.

BarrierSynch

`BarrierSynch` es una clase entrada que implementa un tipo de coordinación condicional que envuelve la decisión para iniciar o postergar el traspaso de estado interno en el algoritmo de recuperación. Una instancia de `BarrierSynch` ingresa también en el grupo redundante antes de comenzar el servicio de `FT-JavaSpaces` .

StateTransferONOFF

Una instancia de la *clase* `StateTransferONOFF` posee una atributo del tipo booleano y utilizado en el algoritmo de recuperación para anunciar a los *Front-Ends* el inicio y fin del algoritmo de recuperación. Dicha instancia ingresa al mismo tiempo que `Processes` y `BarrierSynch` en el par de espacios de tuplas.

5.4 Algoritmo de atención de invocaciones

El pseudocódigo 5.4.1 muestra el algoritmo general empleado por un receptor de invocaciones para la atención de una invocación.

Pseudocódigo 5.4.1 (Algoritmo de Atención de las Operaciones)

```
Sea Op_x una operaci\'on del servicio de FT-JavaSpaces

public {Lease|Entry} OP_x(Entry tuple) throws RemoteException {

//REVISION DE LA EJECUCION DEL ALGORITMO DE RECUPERACION
 if (ST.state() == true) {
    System.out.println("WAITING IN RECEIVER");
    ST.waituntilpossible();
 }

// CREACION DEL MANJEADOR CONFIABLE DE OPERACION RESPECTIVO
 ReliableOp_x a = new ReliableOp_x(tuple);
// NOMBRAMIENTO DEL ESPACIO ESPACIO PRIMARIO Y DE RESPALDO
 a.addHandler0(super.js0);
 a.addHandler1(super.js1);

//INICIO DE UN NUEVO HILO DE CONTROL
 Thread t = new Thread(this.W, a);
 t.start();
```

```
try {
   t.join();
} catch(InterruptedException ex) {
              ...
}

// FASE DE DETECCION DE FALLOS
 boolean b = a.failure();
 if(b == true) {throw new RemoteException();}

// OBTENCION DE RESULTADO
 ResultadoOp_x = a.result();

// DEVOLUCION DE RESULTADO
 return ResultadoOP_x;
}
```

La figura 5.3 muestra el diagrama de interacción correspondiente a la atención de una operación out(). La relación entre los actores en este diagrama puede extenderse a las demás operaciones y resumirse en la siguiente secuencia de pasos:

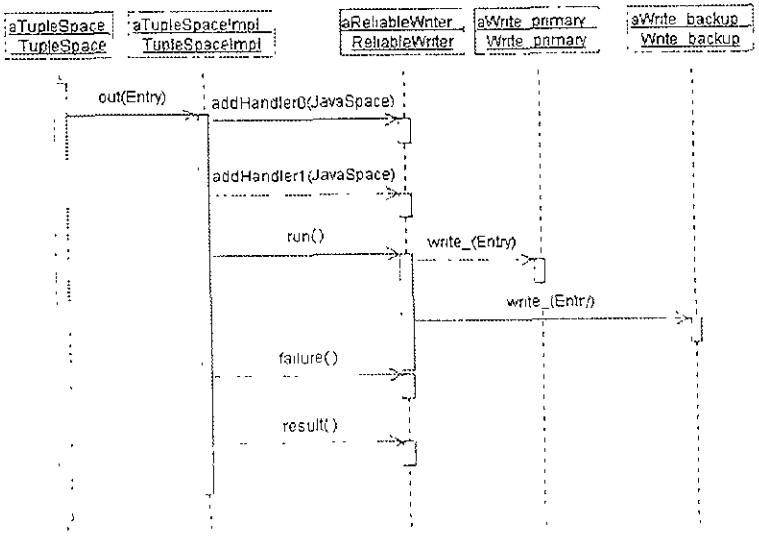

Figura 5.3: Diagrama de interacción para la operación out().

1. Un proxy (aTupleSpace) recibe una invocación y la delega al receptor de invocaciones (aTupleSpaceImpl).

2. A su vez. el receptor de invocaciones (aTupleSpaceImpl) delega la atención de la invocación a un manejador confiable de operaciones (aReliableWriter), el cual opera en un hilo de control independiente.

3. El manejador confiable de operaciones (aReliableWriter) utiliza inicialmente un operador primario (aWrite_primary) para interactuar con el grupo redundante o únicamente con el espacio primario, en caso de que el de respaldo haya experimentado un fallo.

4. En caso de presentarse un fallo en el espacio *primario*, el manejador confiable de operaciones (aReliableWriter) se apoya un operador secundario (aWrite_backup) y la interacción se realiza solamente con el espacio de respaldo.

5. Por último, el receptor de invocaciones pregunta si la invocación fue satisfactoriamente atendida (failure()) y el resultado de la misma (result()). En el caso de haber experimentado un fallo ambos espacios la invocación no es posible que sea atendida satisfactoriamente y el cliente recibe un excepción para indicar el fallo en el servicio de FT-JavaSpaces.

5.5 Manejo de un fallo en el espacio o canal de respaldo

En el caso de presentarse un fallo en el espacio o canal de respaldo, el `operador primario` (OP) lleva únicamente acabo su función en el espacio primario (ver figura 5.4, por ejemplo).

5.6 Manejo de un fallo en el espacio o canal primario

En el momento de que el espacio primario experimenta un fallo es necesario que todos los operadores primarios en proceso de retirar una tupla del espacio de respaldo terminen su función antes de que se creen nuevos operadores primarios y que los operadores secundarios recién creados entren en acción.

El manejo que se realiza de un fallo en el espacio o canal primario varía si se trata de una operación out() o del resto de las mismas.

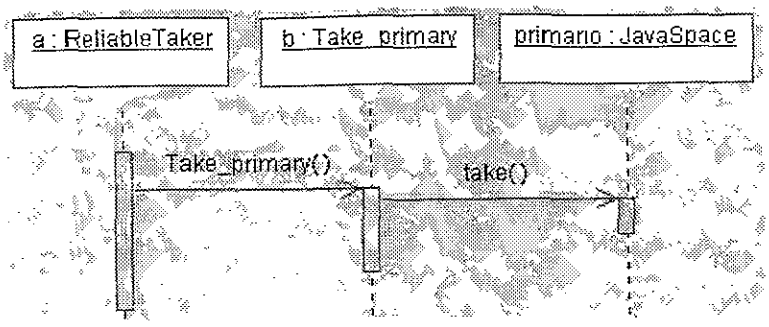

Figura 5.4: Fallo en el espacio de respaldo.

5.6.1 Caso 1: operación out()

Si se presenta un fallo en el espacio primario, una invocación del tipo out() se realiza de la siguiente forma (ver figura 5.5):

1. El manejador confiable (a:ReliableWriter) detecta un fallo en el espacio primario al utilizar un operador primario (b:Write_primary).

2. El manejador confiable crea un operador secundario (c:Write_backup), el cual modificará el estado interno del nuevo espacio primario.

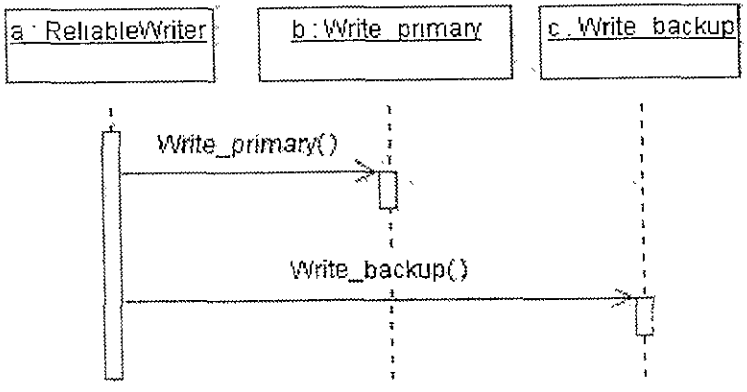

Figura 5.5: Algoritmo para la operación out() y fallo en el espacio primario.

5.6.2 Caso 2: operaciones in(), rd(), inp() o rdp()

Las invocaciones de una operación in(), rd(), inp() o rdp() se clasifican como **nuevas** o **en proceso** dependiendo del grado de avance que se tenga en su atención, por parte del servidor, en el momento de detectarse un fallo en el espacio primario.

Invocación nueva

En el caso de tratarse de una invocación nueva, los pasos que sigue la atención de una invocación del tipo in(), rd(), inp() o rdp() son (ver figura 5.6):

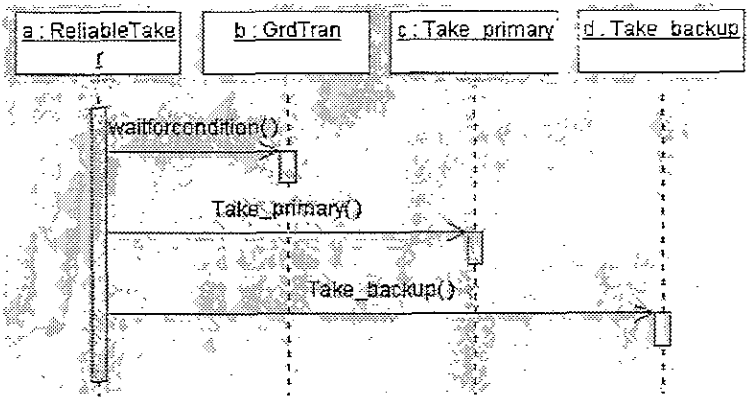

Figura 5.6: Algoritmo para invocaciones nuevas y fallo en el espacio primario.

1. El manejador confiable de la operación respectivo (a:ReliableTaker) revisa si existe un estado de transición donde un fallo en el espacio primario obliga al de respaldo a convertirse en primario (waitforcondition()).

2. Una vez que el espacio de respaldo asume el rol de primario. el manejador confiable de la operación in(), rd(), rdp() o inp() intenta crear un operador primario (c:Take_primary) por si el espacio que experimentó un fallo ha sido satisfactoriamente recuperado y ha asumido de nuevo el papel de espacio primario.

3. Si persiste el fallo en el espacio primario, el manejador confiable de la operación respectivo crea ahora un operador secundario (d:Take_back-

up). El operador secundario intervendrá finalmente con el nuevo espacio primario.

Invocación en proceso

En el caso de tratarse de una invocación en proceso y que al intentar interactuar con el espacio primario detecta un fallo, los pasos que sigue el manejador confiable de operación respectivo son (ver figura 5.7):

1. Avisar del fallo e indicar de una transición de rol de respaldo a primario a todos los demás manejadores confiables de operaciones (primaryNotOKnow()). En caso de que algún otro manejador haya informado anticipadamente de la transición, el manejador confiable pasa simplemente al siguiente paso del algoritmo.

2. El manejador confiable (a:ReliableTaker) espera que los operadores primarios existentes se terminen y el cambio de rol se lleve a cabo (waitforcondition()).

3. Finalmente, el manejador confiable crea un operador secundario (d:Take_backup) para interactuar con el nuevo y único espacio.

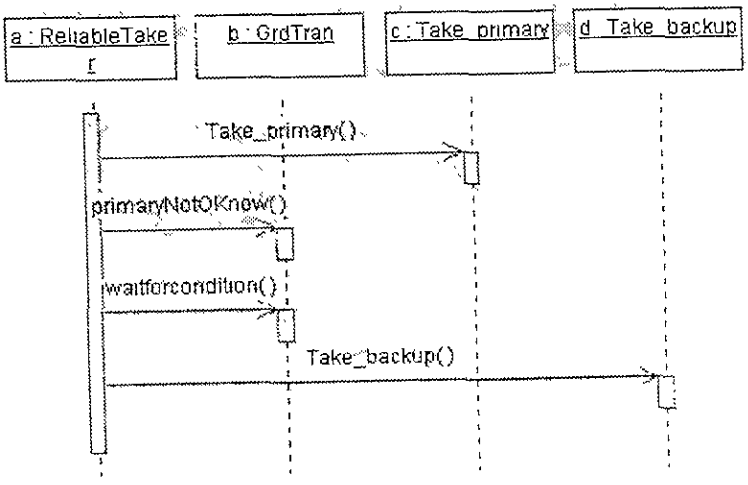

Figura 5.7: Algoritmo para invocaciones en proceso y fallo en el espacio primario.

Nota: Las clases GrdTran y GuardedT implementan ambas los métodos waitforcondition() y primaryNotOKnow(); sin embargo, GrdTran es utilizada en la versión centralizada de FT-JavaSpaces y GuardedT en la distribuida.

5.7 Algoritmo de recuperación centralizado

La figura 5.8 muestra la comunicación y coordinación entre los objetos involucrados en la recuperación de un espacio de tuplas. La secuencia de pasos para el traspaso de estado interno se resume a continuación:

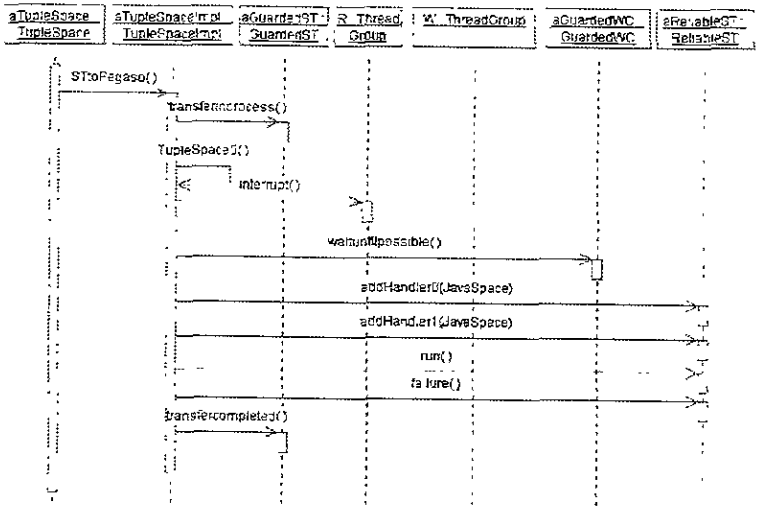

Figura 5.8: Diagrama de interacción para el algoritmo de recuperación centralizado.

1. El método para la transferencia de estado interno a un espacio nuevo o reinicializado es invocado (STtoPegaso()).

2. El receptor de invocaciones RDI_i (aTupleSpaceImpl) notifica a las nuevas peticiones del inicio de un procedimiento de recuperación (trasferinprocess()). De esta forma, todas las peticiones nuevas son suspendidas.

3. RDI_i actualiza su referencia al nuevo espacio en cuestión (TupleSpace0).

4. La atención de las invocaciones correspondientes a las operaciones rd() e in() es interrumpida (interrupt()).

5. RDI_i espera a que no exista ninguna operación inp(), rdp() o out() en proceso de atención.

6. A continuación, se crea una instancia de un objeto de la clase *ReliableST* y se le informa la identidad de los espacios fuente (addHandler0()) y destino (addhandler1()).

7. La instancia de *ReliableST* completa el traspaso y es interrogada si hubo algún error en el mismo.

8. Por último, RDI_i reanuda todas las operaciones suspendidas en el segundo paso.

5.8 Algoritmo de recuperación distribuido

El algoritmo de recuperación distribuido se divide en tres procedimientos. El primero es el de inicialización y le siguen los de recuperación y el de *monitoreo y participación.*

5.8.1 Inicialización

Las acciones en el procedimiento de incialización se señalan en la figura 5.9. En este sentido, un objeto de la clase *JSInitialization* crea una instancia de las *entradas BarrierSynch*, *StateTransferONOFF* y *Processes* con el fin insertar una copia de las mismas en cada uno de los espacios.

5.8.2 Monitoreo y participación en el Alg. de recuperación

Todos los *Front-Ends* en la versión distribuida de FT-JavaSpaces participan en el algoritmo de recuperación. Los pasos que sigue cada *Front-End* en dicho algoritmo se explican a continuación (ver figura 5.10):

1. Un hilo de control independiente monitorea constantemente si está por comenzar la recuperación de un espacio de tuplas (rd(tcondition)).

2. En el caso de detectarse el comienzo de un algoritmo de recuperación, todas las invocaciones nuevas son temporalmente suspendidas (transferinprocess()).

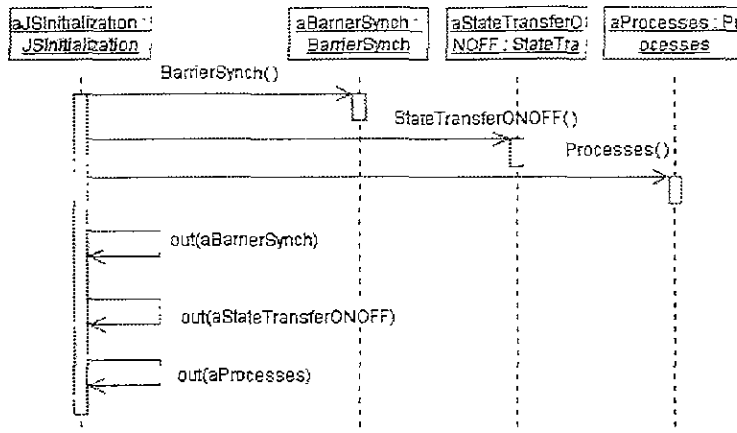

Figura 5.9: Diagrama de interacción del proceso de incialización.

3. El receptor de invocaciones RDI_d (aJSImplementation) interrumpe entonces todas las operaciones in() y rd().

4. RDI_d espera la finalización en la atención de las operaciones out(), rdp() e inp() en progreso.

5. A continuación, el receptor de invocaciones recoge la instancia del tipo *BarrierSynch* del espacio en operación e incrementa su contador interno (increment()).

6. RDI_d ingresa de nuevo la instancia de *BarrierSynch* al espacio en operación.

7. A partir de este momento, RDI_d espera la noticia de que el traspaso se realizo de manera exitosa (brd(fcondition)) para actualizar sus referencias (TupleSpace0() y TupleSpace1()) y reanudar las invocaciones suspendidas (transfercompleted()).

El pseudocódigo 5.8.1 muestra las lineas de código del hilo de control encargado del monitoreo y la participación de un `Front-End` en el algoritmo de recuperación.

Pseudocódigo 5.8.1 (Hilo de Monitoreo)

```
public void run(hilomonitor) {
```

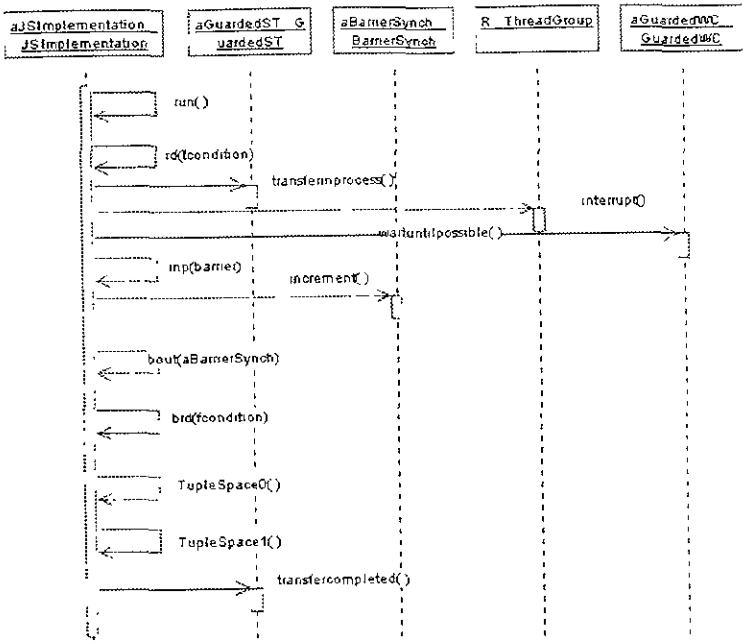

Figura 5.10: Participación de un *Front-End* en el algoritmo de recuperación.

```
       for(;;) {
        try {
          System.out.println("STARTING THREAD");
          System.out.println("ABOUT TO READ STCONDITION");

//REVISION DE EJECUCION DE ALGORITMO DE RECUPERACION
          do {
          continueCond = (StateTransferONOFF)rdp(tcondition);
          System.out.println(continueCond);
          } while(continueCond == null);
          continueCond = null;
          System.out.println("AFTER READ STCONDITON TRUE");

 //DETENER NUEVAS OPERACIONES
          ST.transferinprocess();

//INTERRUMPIR READS
          this.R.interrupt();

//ESPERAR A QUE NO HAY WRITES
```

```
        WC.waituntilpossible();

//PARTICIPACION EN ALGORITMO DE RECUPERACION
        BarrierSynch temporal = (BarrierSynch)bin(barrier);
        System.out.println("ABOUT TO INCREMENT BARRIERSYNCH");
        temporal.increment();
        bout(temporal);
        System.out.println("ABOUT TO WAIT FOR CONDITION TO RESTART")
        brd(fcondition);
//ACTUALIZACION DE REFERENCIAS
        super.TupleSpace0();
        super.TupleSpace1();
        ST.transfercompleted();
       } catch (Exception e) {
         System.out.println(e);
       }
      }
   }
```

5.8.3 Recuperación

El procedimiento central en el algoritmo de recuperación distribuido es el de recuperación. En ese sentido, un objeto particular encargado de la recuperación del espacio primario (aPegasoRecuperation) o de respaldo (aDebianRecuperation) es el encargado del mismo. Los pasos que sigue dicho objeto son (ver figura 5.11):

1. El objeto encargado de la recuperación Obj_x (e.g., aPegasoRecuperation) retira la *entrada* t_{st} de la clase *StateTransferONOFF* del espacio en operación.

2. Obj_x cambia la condición de t_{st} de manera que refleje el inicio de una recuperación (conditionON()) y la devuelve en seguida al espacio (out(aStateTransfer)).

3. Obj_x cuenta el número de *Front-Ends* en funcionamiento (countprocesses.count()) y espera a que todos estén preparados para realizar el traspaso de estado interno (in(processynch)).

4. Una vez todo listo (in(processynch)), Obj_x (aPegasoRecuperation) efectúa el traspaso de estado interno.

5. En seguida, Obj_x (aPegasoRecuperation) retira de nuevo la *entrada* t_{st} y la restaura a su condición original con el fin de reanudar todas

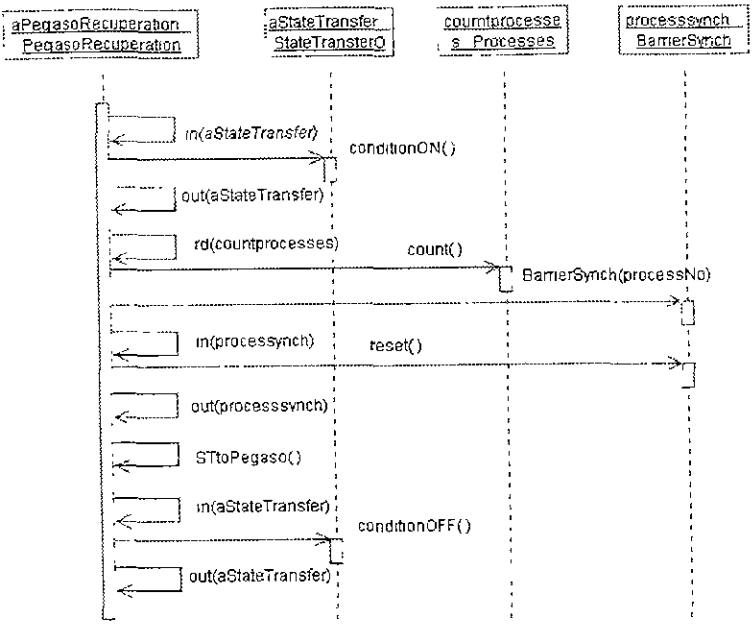

Figura 5.11: Diagrama de interacción del procedimiento de recuperación.

las operaciones suspendidas en los *Front-End*s (conditionOFF()).

6. Por último, Obj_x almacena una copia de t_{st} en el par de espacios de tuplas, incluido el recuperado con antelación.

5.9 Discusión

Este capítulo describe la implementación de FT-Javaspaces en términos de los paquetes y clases que lo conforman. A su vez, explica mediante diagramas de interacción la relación entre los objetos presentes en la atención de una invocación y la recuperación de un espacio.

Las dos implementaciones de un espacio de tuplas que utiliza FT-JavaSpaces son del tipo JavaSpaces. JavaSpaces es un espacio de tuplas orientado a objetos y que permite incorporar sus primitivas[4], equivalentes a las de Linda, en el lenguaje de programación JAVA. Las ventajas de JavaSpaces son básicamente su fácil obtención, su participación en el lenguaje JAVA y

[4]Con excepción de eval()

su manejo de excepciones. Estos tres puntos fueron la clave para su elección en la implementación de FT-JavaSpaces.

Las diferentes capas que conforman a `FT-JavaSpaces` corresponden a un diseño modular donde se encapsulan diferentes acciones y elementos que requieren de una vida independiente. De esta forma, cualquier modificación necesaria incluye únicamente a una sección del código. Esto presenta obviamente ventajas a la hora de modificar o de agregar una funcionalidad a `FT-JavaSpaces` junto con la prestación de una mayor facilidad para la comprensión de su funcionamiento.

Los diagramas de interacción que presenta siguen el modelo UML para la modelación de sistemas, lo cual permite describir a FT-JavaSpaces en términos de un lenguaje de modelación aceptado internacionalmente.

En conclusión, la implementación de `FT-JavaSpaces` sigue los consejos para un desarrollo escalable o reutilizable. Sin embargo, su principal atributo es que lleva completamente a la práctica su diseño y, en particular, cumple adecuadamente con sus requerimientos.

Capítulo 6

Resultados

Este capítulo presenta la evaluación correspondiente a la implementación de FT-JavaSpaces. El propósito de esta evaluación consiste principalmente en presentar las característica logradas en términos de sus especificaciones. La evaluación se divide esencialmente en términos cualitativos y cuantitativos.

6.1 Evaluación cualitativa

La evaluación cualitativa de FT-JavaSpaces abarca el cumplimiento de sus requerimientos y, en particular, la funcionalidad y semántica establecidas en el diseño. Con este fin, dos aplicaciones de prueba fueron empleadas y escogidas con el fin de reflejar implícitamente los beneficios de FT-JavaSpaces: el cálculo del fractal de Mandelbrot y un contador infinito, los cuales se explicarán más adelante.

6.1.1 Modelo experimental

En cuanto a hardware, el equipo utilizado en las pruebas de FT-JavaSpaces consiste en:

1. Cinco computadoras Pentium III (X-86), con 64 MB en memoria RAM y con una velocidad de procesador igual a 500 Mhz
2. Las cinco computadoras se conectan por medio de un HUB para redes Ethernet LAN y cables del tipo PDS.

En relación al software empleado, éste puede dividirse en:

1. La maquina virtual de JAVA (JVM) de Sun Microsystems.

2. Los programas y clases de la herramienta para el desarrollo de programas en JAVA denominada JDK1.2.1, original de Sun Microsystems.

3. Los programas y clases pertenecientes al modelo *Jini 1.1* para la construcción de aplicaciones distribuidas de Sun Microsystems.

4. Los programas y clases que implementan el servicio de *JavaSpaces* de Sun Microsystems.

5. Los clases de `FT-JavaSpaces`.

En particular, dos máquinas con los nombres *Pegaso* y *Debian* son quienes albergan respectivamente las implementaciones primaria y de respaldo de *JavaSpaces*. Las tres máquinas restantes sirven como Front-Ends de la implementación del servicio de `FT-JavaSpaces` en su versión distribuida. En cuanto a la versión centralizada de `FT-JavaSpaces`, una máquina es únicamente necesaria como Front-End para su implementación.

Por último, las aplicaciones fueron puestas a prueba en ambas versiones de `FT-JavaSpaces`. Durante el funcionamiento de las aplicaciones, el experimento consistió básicamente en

1. Provocar un fallo en el espacio *primario*.

2. Verificar la continuación en el funcionamiento de la aplicación.

3. Recuperar el espacio *primario*.

4. Provocar un fallo en el espacio de *respaldo*.

5. Verificar nuevamente la continuación en el funcionamiento de la aplicación.

6. Provocar que ambos espacios en el grupo no proporcionen su servicio simultáneamente.

A continuación, se explican las aplicaciones de prueba escogidas y se muestran los resultados cualitativos particulares de cada una.

6.1.2 Fractal de Mandelbrot

El fractal de Mandelbrot es una imagen que se construye a partir de la iteración a través de una fórmula cuyos valores de entrada son nn número de puntos (x, y) y cuyos resultados son un conjunto de valores enteros.

El cálculo del fractal de Mandelbrot consiste en asignar a cada pixel de una región en la pantalla un valor entero específico y correspondiente a un color.

Este problema es uno que se puede dividir entre varios procesadores y organizar de acuerdo al patrón denominado "replicated-worker". Este patrón se compone específicamente de un proceso *maestro* (master) y un número definido de procesos denominados *trabajadores* (workers).

En esta implementación, el *maestro* divide la región de la pantalla en areas con tamaños similares y representadas por *tuplas*,o tareas, definidas que deposita en el espacio. Los *trabajadores* esperan constantemente por que se les asigne una tarea, ejecutando un in(), y al terminar de calcular un resultado lo insertan en el espacio en forma de otra *tupla*. El proceso *maestro* recolecta finalmente todos los resultados y los combina en la imagen 6.1.

Resultados

FT-JavaSpaces permite resolver este problema con un código similar al de su solución en *JavaSpaces*. Los pseudocódigos 6.1.1 y 6.1.3 correspondientes al maestro y un trabajador en *JavaSpaces* son prácticamente idénticos a sus versiones en FT-JavaSpaces 6.1.2 y 6.1.4, respectivamente.

Pseudocódigo 6.1.1 (Master de JavaSpaces)

```
private void generateTasks() {
  space.write(task, null, Lease.FOREVER);
}

private void collectResults() {
  space.take(template, null  Long.MAX_VALUE);
}
```

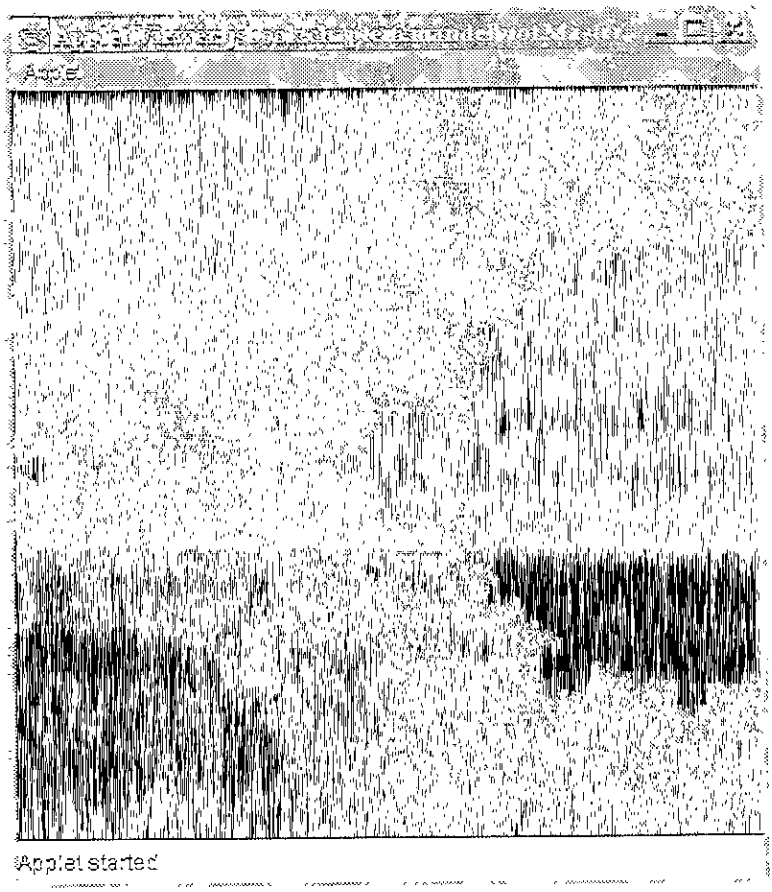

Figura 6.1: Fractal de Mandelbrot.

Pseudocódigo 6.1.2 (Master de FT-JavaSpaces)

```
private void generateTasks() {
  space.out(task);
}

private void collectResults() {
 space.in(template);
}
```

Pseudocódigo 6.1.3 (Worker de *JavaSpaces*)

```
public static void main(String[] args) {
 space.take(template, null, Long.MAX_VALUE);
 // Calculo de Fractal
 space.write(result, null, Lease.FOREVER);
}
```

Pseudocódigo 6.1.4 (Worker de FT-JavaSpaces)

```
public static void main(String[] args) {
 space.in(template);
 // Calculo de Fractal
 space.out(result);
}
```

De esta forma, la programación con FT-JavaSpaces no requiere de un conocimiento adicional fuera del paradigma de espacios de tuplas.

El resultado del experimento fue que el funcionamiento de la aplicación continuó en todos sus pasos y los fallos en los espacios se dieron de manera transparente para los clientes de FT-JavaSpaces. En este sentido, la confiabilidad que ofrece FT-JavaSpaces en la terminación del cálculo del fractal es mayor a la de su solución en una sola implementación de *JavaSpaces*.

6.1.3 Contador infinito

La segunda aplicación utilizada en la evaluación de FT-JavaSpaces consiste en un contador manejado por dos tipos de procesos: un servidor y un determinado número de clientes.

Defínase una tupla $\mathcal{C}_l$ que contiene un campo *cont* formado por una variable entera de valor inicial cero que funciona como contador.

1. El servidor ingresa inicialmente a $\mathcal{C}_l$ al espacio y monitorea constantemente el valor de su campo *cont* para desplegarlo en pantalla (ver figura 6.2).

2. Los procesos clientes retiran la tupla $\mathcal{C}_l$ e incrementan el valor de [illegible]

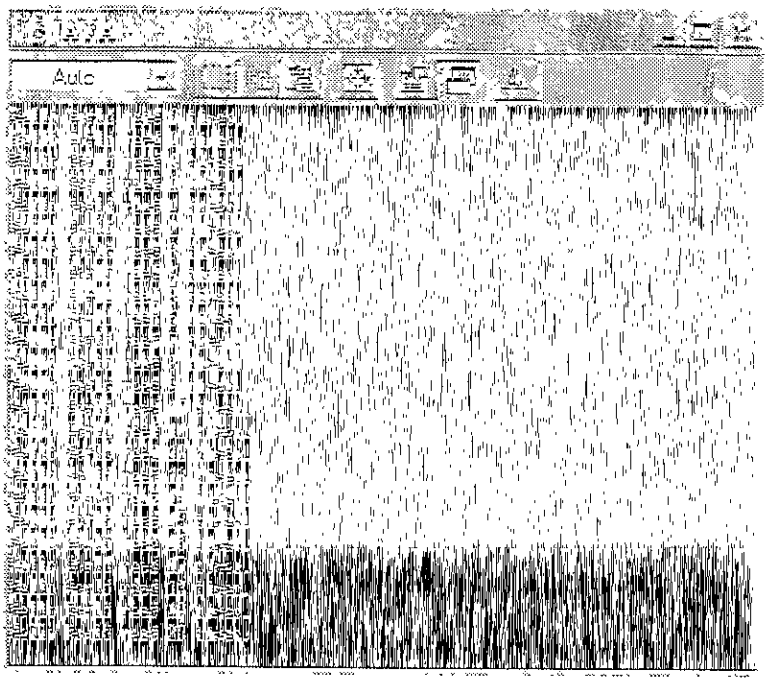

Figura 6.2: Salida del proceso servidor.

La cuenta que lleva la *tupla* C_t puede considerarse como los pasos de una computación que involucra un número de procesos distribuidos.

Resultados

Durante el transcurso del experimento, la cuenta que llevó C_t continuó sin interrupción y los fallos en los espacios de tuplas fueron transparentes para el servidor y los clientes.

De esta forma, FT-JavaSpaces ofrece una mayor confiabilidad en que esta particularmente "tardada" computación continúe en comparación a la de su solución utilizando una implementación de *JavaSpaces*.

6.1.4 Resultados cualitativos

De acuerdo las aplicaciones de prueba utilizadas, la evaluación cualitativa arrojó los siguientes resultados respecto a los requerimientos alcanzados:

1. FT-JavaSpaces exporta las operaciones out(), in(), rd(), rdp() e inp() del modelo Linda por medio de un representante (proxy) y que sus clientes utilizan localmente.

2. FT-JavaSpaces implementa un servicio de espacio de tuplas del tipo Linda y cumple con los requerimientos y características de este último modelo. FT-JavaSpaces garantiza este requerimiento en sus modos de operación completo (sin fallos que tolerar) y reducido (con un fallo que tolerar).

3. Alta Disponibilidad. La disponibilidad en el servicio de FT-JavaSpaces es mayor a la de una implementación de *JavaSpaces* por su uso de un grupo redundaute formado por dos implementaciones de un espacio de tuplas. En este sentido, la probabilidad de que el servicio de FT-JavaSpaces se encuentre en servicio en un instante cualquiera es mayor a la de una sola implementación de uu espacio de tuplas.

4. Alta confiabilidad. La confiabilidad de FT-JavaSpaces en el transcurso de las aplicaciones de prueba fue mayor en comparación con la utilización de una implementación única de un espacio de tuplas gracias a la ejecución del algoritmo de recuperación. De esta forma, la probabilidad de que las aplicaciones de prueba terminaran satisfactoriamente es mayor a el caso en donde hubieran empleado una sola implementación de un espacio de tuplas.

5. FT-JavaSpaces ofrece una vista transparente de sus mecanismos propios. En este sentido, los clientes del mismo no son capaces de diferenciarlo de una implementación de un espacio de tnplas no tolerante a fallas.

De esta forma, la implementación de FT-JavaSpaces cumple con todos los requerimientos establecidos en su diseño.

6.2 Evaluación cuantitativa

La evaluación cuantitativa de FT-JavaSpaces se realiza básicamente en términos de su disponibilidad y confiabilidad en el servicio: así como en el costo empírico en tiempo de las operaciones y del cálculo del fractal de Mandelbrot

6.2.1 Disponibilidad

Todas las operaciones de FT-JavaSpaces pueden realizarse siempre y cuando exista al menos un espacio de tuplas (primario o de respaldo), o implementación de *JavaSpaces*.

Si una implementación de *JavaSpaces* tiene una probabilidad instántanea P_f de sufrir un fallo, la disponibilidad de FT-JavaSpaces A_{FT} es igual a la probabilidad de que por lo menos un espacio de tuplas, el primario o de respaldo, prosiga ofreciendo su servicio en un instánte:

$$A_{FT} = 1 - (P_f)^2 \tag{6.1}$$

La tabla 6.1 muestra el nivel de tolerancia a fallas de FT-JavaSpaces en términos de su disponibilidad y en función de la probabilidad instántanea de fallo, P_f, de sus implementaciones de *JavaSpaces* (2).

Prob. Instántanea de Fallo P_f (%) individual para los espacios primario y de respaldo.	Disponibilidad de FT-JavaSpaces (A_{FT})
0	1.0000000000
5	0.9975000000
10	0.9900000000
15	0.9775000000
20	0.9600000000
25	0.9375000000
30	0.9100000000
35	0.8775000000
40	0.8400000000
45	0.7975000000
50	0.7500000000

Tabla 6.1: Disponibilidad de FT-JavaSpaces de acuerdo a una misma probabilidad instántanea de fallo P_f para cada espacio de tuplas primario o de respaldo.

6.2.2 Confiabilidad

La confiabilidad de un servicio se refiere normalmente a la probabilidad de que el mismo continúe funcionando durante un intervalo de tiempo en el cual realiza una cierta computación.

Si los espacios primario y de respaldo son idénticos, la confiabilidad de FT-JavaSpaces, en términos del *tiempo promedio de aparición de un fallo* $MTTF$ y el *tiempo promedio de recuperación* $MTTR$ de sus implementaciones primaria y de respaldo, es igual a:

$$MTTF_{FT-JavaSpaces} = \frac{MTTF}{2} * \frac{MTTF}{MTTR} \tag{6.2}$$

En resumen, la ecuación 6.2 es el producto del tiempo promedio para la aparición del primer fallo por el inverso de la probabilidad de presentarse un segundo fallo en uno de los espacios del grupo durante el $MTTR$ del espacio restante [BG88]. De manera intuitiva, la ecuación anterior puede deducirse a partir de la probabilidad de presentarse dos fallos consecutivos por parte de los espacios primario y de respaldo en un lapso de tiempo igual a 2 $MTTR$ y que obliguen a FT-JavaSpaces a sufrir una caída.

Debido a que la confiabilidad es dependiente del tipo de trabajo computacional que se encuentre realizando el sistema, la evaluación de la confiabilidad de FT-JavaSpaces no toma en cuenta ninguna tarea computacional en particular. En cambio, utiliza un conjunto de valores de ejemplo para el MTTF del par de implementaciones de JavaSpaces junto con un MTTR de las mismas y determinado de manera empírica durante el desarrollo de FT-JavaSpace. Es decir, el MTTR propuesto para ambos espacios es el tiempo que en la práctica se comprobó que garantiza que el espacio estará de nuevo en funcionamiento.

La tabla 6.2 muestra una diferencia sustancial entre el MTTF de FT-JavaSpaces y el MTTF de una implementación de *JavaSpaces*.

Por ejemplo, si el *tiempo promedio de aparición de un fallo* en *JavaSpaces* es de una semana entonces el valor del mismo parámetro en FT-JavaSpaces es de 38 años.

La tabla 6.2 muestra los valores correspondientes a la ecuación 6.2

MTTF-JavaSpaces (Hrs.)	MTTR-JavaSpaces = 7 Min. (Hrs.)	MTTF-FT-JavaSpaces (Hrs.)	MTTF-FT-JavaSpaces (Años)
24 (1 día)	0.116700	2467.866324	0.2817199
168 (1 Sem.)	0.116700	120925.4499	13.8042751
672 (4 Sem.)	0.116700	1964807.198	220.8684013
1344 (8 Sem.)	0.116700	7789228.792	889.4736064
2016 (12 Sem.)	0.116700	17413264.78	1987.815614
2688 (16 Sem.)	0.116700	30956915.17	3533.894425
3360 (20 Sem.)	0.116700	48370179.95	5521.71004
4032 (24 Sem.)	0.116700	69653059.13	7951.262457
8064 (48 Sem.)	0.116700	278612236.5	31805.04983
8736 (52 Sem.)	0.116700	326932416.5	37323.75987
9408 (56 Sem.)	0.116700	379222210.8	43290.20671

Tabla 6.2: MTTF de FT-JavaSpaces vs MTTF de *JavaSpaces*.

6.2.3 Costo empírico de las operaciones

El costo empírico en tiempo promedio para la realización de una operación en JavaSpaces y en las versiones centralizada y distribuida de FT-JavaSpaces se resume en la tabla 6.3. Cada valor en esta tabla es un promedio de nueve valores actuales obtenidos en la práctica (ver apéndice A). En el caso de las versiones de FT-JavaSpaces, no se asume ningún fallo en el sistema.

Op.	JavaSpaces (ms)	FT-JavaSpaces, V. Cent. (ms)	FT-JavaSpaces V. Dist. (ms
out()	19.7 <	80.8 (410%) <	154.22 (782%)
in()	16.8 <	76.0 (452%) <	357.77 (2337%)
rd()	16.3 <	41.4 (254%) <	343.55 (2107%)
inp()	18.1 <	73.5 (406%) <	338.00 (1867%)
rdp()	16.4 <	25.0 (152%) <	316.33 (1928%)

Tabla 6.3: Cuadro comparativo del costo en tiempo de las operaciones.

En la tabla 6.3 puede observarse como el costo mas bajo se tiene en JavaSpaces y el más caro en la versión distribuida de FT-JavaSpaces. En porcentaje, el costo en tiempo de las operaciones es siempre superior en el caso de FT-JavaSpaces en comparación con JavaSpaces. Esta situación es explicable tomando en cuenta el nivel de abstracción añadido. Es decir,

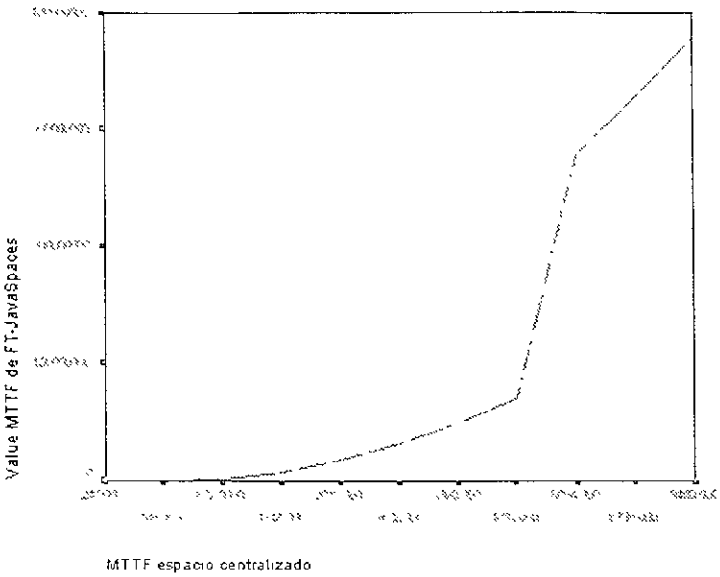

Figura 6.3: MTTF de FT-JavaSpaces vs MTTF de JavaSpaces.

la adición de los mecanismos que encierran los *Front-Ends* se añaden a los básicos de JavaSpaces.

6.2.4 Costo empírico del cálculo del fractal de Mandelbrot

La tabla 6.4 resume el costo promedio en tiempo del cálculo del fractal de Mandelbrot en JavaSpaces y las dos versiones de FT-JavaSpaces. En todos los caos, existe únicamente un proceso *maestro* y otro *trabajador*.En el caso de las versiones de FT-JavaSpaces se asume también que no existe ningún fallo en el sistema.

Como puede observarse en la tabla 6.4, el costo del cálculo del fractal aumenta en FT-JavaSpaces. De manera porcentual, el costo más grande se obtiene, al igual que en el caso de las operaciones individuales, en la versión distribuida de FT-JavaSpaces. De nuevo, la abstracción añadida en FT-JavaSpaces para el manejo de fallos incrementa en costo en rendimiento.

JavaSpaces (seg)	FT-JavaSpaces. V. Cent (seg)	FT-JavaSpaces V. Dist. (seg)
6.81	13.10	37.81
5.66	13.23	31.35
6.20	12.99	32.43
6.35	12.52	33.33
6.22	12.32	31.11
6.55	14.30	33.82
5.78	13.00	33.52
5.97	14.02	31.00
6.12	13.25	37.77
	Promedio en seg	
6.18 <	13.27 (214%) <	33.90 (548%)

Tabla 6.4: Tiempo en seg. para el cálculo del fractal de Mandelbrot.

6.3 Discusión

Este capítulo describe los resultados cualitativos y cuantitativos en la evaluación de la implementación de FT-JavaSpaces. La evaluación se fundamenta en el funcionamiento de dos aplicaciones de prueba durante las cuales el sistema sufrió toda combinación posible de los fallos tolerados en las especificaciones.

En particular, la evaluación cualitativa examinó el cumplimiento de los requerimientos mientras que la cuantitativa se enfocó en la medición de confiabilidad y disponibilidad de la implementación.

Además, la evaluación empírica de los costos en tiempo de las operaciones y en el cálculo del fractal de Mandelbrot permitieron comparar en la práctica a FT-JavaSpaces con una versión no tolerante a fallos.

En conclusión, FT-JavaSpaces cumplió con todos los requerimientos y ofrece un nivel de tolerancia a fallas mayor al de una implementación de *JavaSpaces* en términos de confiabilidad y disponibilidad.

En cuanto a su comparación con otras implementaciones de un espacio de tuplas tolerante a fallas, FT-JavaSpaces se caracteriza por ser el primero en utilizar el esquema primario-respaldo, contar con un algoritmo para la recuperación de los espacios en el grupo, funcionar en una plataforma como la Internet y ofrecer una vista transparente de sus mecanismos de tolerancia a fallas.

Capítulo 7

Proyectos Relacionados

Este capítulo trata de los proyectos más importantes que han buscado la implementación de un espacio de tuplas tolerante a fallas, sus características principales y su clasificación.

La propuesta de esta tesis, `FT-JavaSpaces`, es comparada con todas las demás al final del capítulo.

7.1 Introducción

El estudio sobre los espacios de tuplas se centra en los problemas que tienen sus implementaciones y, en particular, en los aspectos de:

- Tolerancia a fallas y
- Escalabilidad.

El objetivo de esta tesis deja a un lado el tema de la escalabilidad por considerar que su estudio requiere realmente de un análisis completo por sí mismo. En consecuencia, este capítulo incluye únicamente proyectos que buscan resolver el problema de la tolerancia a fallas.

Los *espacios de tuplas* tolerantes a fallas pueden clasificarse de acuerdo a los siguientes criterios de diseño :

Fallas

Origen de las fallas : fallas en el espacio de tuplas (distribuido o centraliza

Tipo

Tipo de fallas abordadas – es decir, de acuerdo a si se busca enmascarar fallos del tipo *caída* en sistemas del tipo "fail-stop", de respuesta, fallos *arbitrarios*, etc.

Mecanismo

El mecanismo de tolerancia a fallas empleado – es decir, si para enmascarar las fallas abordadas se utilizan transacciones o la replicación de datos y/o procesos.

Ops./Func.

La adición de nuevas operaciones o funcionalidades – algunos proyectos incluyen nuevas operaciones y otros aumentan la funcionalidad de las existentes.

De acuerdo a estos parámetros es como se describirán los trabajos siguientes.

7.2 FT-Linda

FT-Linda es una versión distribuida de Linda desarrollada por Bakken y Schlichting [BS95]. En general, FT-Linda incrementa la disponibilidad de los datos en un espacio de tuplas y provee de un servicio de transacciones en la ejecución de las operaciones originales de Linda. Este lenguaje[1] supone una infraestructura física formada por una colección de procesadores conectados por una red y sin una memoria física compartida.

Mecanismo

FT-Linda se basa en la replicación completa de un espacio de tuplas utilizando el enfoque de *máquina de estados* [Sch90], o replicación activa. Un protocolo multicast atómico y ordenado ("ordered atomic multicast") es utilizado para distribuir las operaciones entre las réplicas, lo cual significa un menor número de mensajes en comparación con el uso de un protocolo como el de compromiso de dos fases (2PC).

[1] Recuérdese que Linda puede considerarse como un lenguaje para la coordinación entre procesos.

Fallas y tipo

FT-Linda es tolerante a fallas en el espacio de tuplas y, en particular, a fallos del tipo caída en los procesadores del sistema.

Ops./Func.

La especificación de FT-Linda asume que todos los nodos del sistema pertenecen a la clase "fail-silent"[2]. FT-Linda *asume* eventualmente la caída de dichos nodos e inserta una tupla especial y correspondiente a dicho fallo en el espacio de tuplas. Sin embargo, los nodos o espacios de tuplas que se asume han experimentado una caída no son automáticamente recuperables.

La versión de FT-Linda de un espacio "*estable*" cumple el mismo objetivo del modelo de *almacenamiento estable* de transacciones y permite además un número configurable de réplicas, *N*. En su implementación de un espacio *estable* y con *N* réplicas, FT-Linda soporta hasta *N-1* caídas de procesadores sin considerar particiones de la red. Un espacio *estable* permite al programador construir procesos y espacios de tuplas recuperables.

7.3 Desarrollo de Tam y Woodward

Otro desarrollo denominado también FT-Linda es un proyecto descrito en [TW95] e implementado por Francis Tam y Mike Woodward. El propósito de esta nueva versión distribuida de Linda es contar con espacios "*estables*" y facilitar la construcción de los esquemas de replicación activa ("active-replication") y primario-respaldo ("primary-backup") en base a un conjunto de espacios de tuplas.

Fallas y tipo

La preocupación de este proyecto es básicamente los fallos de tipo caída en [illegible] espacio de tuplas.

Ops./Func.

FT-Linda incorpora nuevas operaciones y tipos de datos al conjunto de primitivas de Linda: `tuplespace`, `stable`, `tsCreate`, `×`, `÷` y `gvote`. Los autores señalan correctamente que estas extensiones son necesarias para resolver [illegible]

adecuadamente problemas comúnes en la tolerancia a fallas.

La definición de los autores para un espacio de tuplas del tipo abstracto `stable` es la de un espacio no volátil. Su implementación no está realizada, pero contempla posiblemente el uso de un sistema de archivos, una técnica de *disk shadowing* o arreglos RAID[3] a futuro.

El nuevo tipo de datos `tuplespace` permite crear copias de un espacio de tuplas y utilizarlo como registro histórico de las operaciones realizadas por una aplicación[4] ("Checkpointing a computation"). Por su parte, `tsCreate` permite crear un nuevo espacio de tuplas para sustituir una réplica con la ayuda de `tuplespace`.

Mecanismo

`gvote` realiza simultáneamente una operación del modelo original de Linda en un número de espacios y con ayuda de un mecanismo votación. En este sentido, utiliza un esquema del tipo de votación ("voting.") para el manejo de réplicas.

$\times \rightarrow$ cuenta con la siguiente sintáxis:

$LindaOp_1 \times \rightarrow LindaOP_2$ {*dondeLindaOp*$_n$*es*[*in*|*out*|*rd*]}

$LindOp_2$ se ejecuta únicamente después de que $LindaOp_1$ no pudo realizarse satisfactoriamente. Esta operación expresa la semántica del esquema primario-respaldo ("primary-backup").

La ventaja principal de este lenguaje es que ofrece una gran flexibilidad y poder de programación a lo programadores de servicios tolerantes a fallas.

7.4 Desarrollo de Xu y Liskov

Xu y Liskov son los creadores de un espacio de tuplas distribuido y tolerante a fallas descrito en [XL89]. Esta implementación se basa en la replicación total de un espacio de tuplas y en la adición de nuevas funcionalidades a las operaciones básicas de Linda[5] –`in`, `rd` y `out`.

[3] Redundant Array of Inexpensive Disks.

[4] En este contexto, una aplicación conlleva un número de procesos.

[5] Las operaciones `inp`, `rdp` y `eval` no están incluidas por considerarse que su anexión resultaría en un sistema más lento.

Mecanismo

El diseño contempla la replicación completa de los espacios, con un número de 3 a 5 réplicas, y utiliza mecanismos de bloqueo y un protocolo de compromiso para las actualizaciones.

Fallas y tipo

El sistema es tolerante a fallas por que enmascara fallos del tipo *caída* en nodos del tipo "fail-stop" encargados de un espacio de tuplas, maneja particiones de la red y asegura la atomicidad de las operacioues sobre los espacios. Sin embargo, el sistema no distingue entre fallas en los nodos o la red debido a que no cuenta con mecanismos de detección particulares. La justificación de los autores radica en que el efecto es el mismo en ambos casos – la ausencia de mensajes.

Ops./Func.

Las réplicas con fallas y fuera del sistema se asume que vuelven a recobrarse para integrarse de nuevo. Un algoritmo especial ("*view change algorithm*") es el encargado de la reconfiguración dinámica el sistema y solamente se permite la continuación de las operaciones cuando una mayoría de las réplicas puede comunicarse entre si.

7.5 MOM

MOM [CD94] es un modelo de programación tolerante a fallas que agrega un mecanismo de transacciones[6] al modelo clásico de Linda.

Fallas y tipo

Hansen y Cannon convierten a MOM en una versión distribuida con una técnica de grupos similar al esquema primario-respaldo y lo hacen capaz de soportar caídas tanto de procesos clientes ("workers"), como de una replica del espacio. En ambas situaciones anteriores, el sistema es capaz de recuperarse en cualquier estado de operación [HC].

[6] El modelo utiliza el protocolo de compromiso de dos fases (2PC)

Mecanismo

El nuevo modelo de Hansen y Cannon maneja un espacio de tuplas, llamado del *sistema* y un número de espacios *locales*. Un espacio del *sistema* contiene todo el conjunto de tuplas, mientras que un espacio *local* guarda únicamente un subconjunto. Cada espacio *local* replica las tuplas y el estado de los procesos locales. Las peticiones son entonces atendidas inicialmente por un manejador – o agente– local y después enviadas a un manejador del sistema. Una de las limitaciones de este modelo es la consideración de un solo espacio de tuplas del *sistema*. Sin embargo, éste último puede residir en mas de un solo nodo.

Este modelo considera también la reconstrucción de los espacios de tuplas *local* y del *sistema*. El manejador del sistema reconstruye un espacio *local* utilizando tuplas del espacio del *sistema* y el último estado asociado al espacio caído. Un espacio del *sistema* se vuelve restaurar a partir de la unión de los conjuntos de tuplas locales.

Ops./Func.

Por último, el modelo agrega la operación `done()` mediante la cual un proceso puede *comprometer* un conjunto de operaciones y permitir a otros procesos observar y manipular la salida (o resultado) de las mismas. El manejo de tiempos de respuesta ("time-outs") permite también tolerar fallas como la pérdida de mensajes y retardos en la comunicación.

7.6 PLinda 2.0

Plinda 2.0 es un proyecto de Jeon y Shasha descrito en [SJ] y que extiende el modelo de Linda.

Fallas y tipo

Este modelo facilita la programación de un espacio de tuplas tolerante a caídas del procesador y ofrece operaciones atómicas sobre el mismo. Sin embargo, este proyecto permite contar además con procesos tolerantes a caídas del nodo en que residen. El tipo de falla en ambos casos es considerado como detectable. Es decir, dichas fallas deben presentarse en un nodo del tipo "fail-stop".

Mecanismo

El mecanismo de tolerancia a fallas de PLinda 2.0 utiliza transacciones para los espacios y una bitácora privada ("process-private logging") para los procesos.

Los espacios de tuplas pueden ser de dos tipos:

Espacio de Tupla Estable , donde todas las actualizaciones son almacenadas en disco autes de ser comprometidas.

Espacio de Tupla Protegido , donde todas las tuplas del espacio son periódicamente almacenadas en disco.

Un proceso, en general, puede ser tolerante a fallas ("resilient process") utilizando réplicas del mismo o guardaudo su estado actual en almacenamiento estable y contar con su rápida recuperación. PLinda 2.0 utiliza la segunda opción. La creación de la bitácora y su contenido no son transparentes, los cuales son responsabilidad del programador del proceso.

Ops./Func.

PLinda 2.0 requiere que todas las operaciones sobre un espacio se realicen dentro de una transacción. El modelo, por lo tanto, añade las primitivas **xstart** y **xcommit** para iniciar y comprometer una transacción, respectivamente. El almacenamiento del estado de un proceso se lleva a cabo utilizando la primitiva **log_inout**, mientras que su recuperación requiere de **log_rdp**.

El diseño de PLinda 2.0 considera una arquitectura cliente-servidor donde un proceso servidor maneja las peticiones de múltiples procesos clientes.

7.7 GLOBE

Globe es un proyecto desarrollado por Jakob Eg Larsen y Jesper Honing Spring [LS99]. Globe es una implementación muy ambiciosa de un espacio de tuplas distribuido, escalable y tolerante a fallas. Su desarrollo fue realizado en lenguaje Java, por lo cual funciona también en ambientes heterogéneos. Al igual que PLinda, su infraestructura física se compone de un conjunto de procesadores conectados en red y sin una memoria física común o "loosely coupled network".

Fallas y tipo

En cuanto a su origen, las caídas de los procesadores en el sistema son principalmente las abordadas por Globe. Globe tolera también particiones de la red[7].

Mecanismo

Globe emplea una técnica de enmascaramiento de fallas utilizando un esquema de grupos del tipo replicación activa. De esta forma, Larsen y Honing aumentan también la *confiabilidad* de un espacio de tuplas. En cuanto a *disponibilidad*, utilizan un algoritmo denominado "anti-entropy" que permite *eventualmente* traspasar el estado de una réplica del espacio a otra recién incorporada al sistema, ya sea nueva o recuperada.

En el caso de particiones de la red y en el caso de anexión de una nueva réplica, Globe emplea un algoritmo que reconfigura dinámicamente el estado de la red.

Ops./Func.

Uno de los dos tipos de fallos que tolera es la caída de los procesadores del sistema. Sin embargo, Globe utiliza una técnica de votación para el manejo de réplicas y que hace necesario que existan siempre tres réplicas funcionales como mínimo. De lo contrario, el espacio de tuplas distribuido sufre a su vez una caída.

Cada nodo del sistema trabaja con una versión orientada a objetos similar a Linda del tipo *Fail-stop* llamada Javaspaces, original de Sun Microsystems.

Globe exporta únicamente las cinco operaciones tradicionales del modelo Linda (in(), inp(), rd(), rdp() y out()); sin embargo, no incorpora tuplas *activas*.

7.8 Discusión

Este capítulo presenta seis proyectos representativos de las propuestas existentes para construir un espacio de tuplas tolerante a fallas. La descripción

[7] La topología de la red es de la clase estrella

de cada proyecto se basa en cuatro parámetros[8]:

- Origen de las Fallas (**Fallas**).
- Tipo de Fallas (**Tipo**).
- Mecanismo de Tolerancia a Fallas (**Mecanismo**).
- Adición de nuevas Funcionalidades u Operaciones al Modelo de Linda (**Ops./Func.**).

y se puede resumir en la tabla 7.1.

Proyecto	Fallas	Tipo	Mecanismo	Ops./Func.
FT-Linda	Espacio	"Crash"	Replicación	Func.
Tam y Woodward	Espacio	"Crash"	Replicación	Ops.
PLinda 2.0	Espacio	"Crash"	Transacciones	Ops.
	Procesos	"Crash"	"Private-logging"	
MOM	Espacio	"Crash"	Replicación	Ops.
Xu/Liskov	Espacio	"Crash"	Replicación	Func.
	Red	Partición	"View Change Algorithm" (Reconfignración)	
Globe	Espacio	"Crash"	Replicación	Func.
	Red	Partición	Reconfiguración	
FT-JavaSpaces	Espacio	"Crash"	Replicación	Func.

Tabla 7.1: Características principales de los proyectos relacionados

Fallas y Tipo

En general, los proyectos trabajan con fallos del tipo caída detectables (sistemas del tipo "fail-stop") en la implementación de un espacio de tuplas, los procesos que interactúan con el mismo o particiones de la red.

FT-JavaSpaces aborda solamente la caída de la implementación de un espacio de tuplas del tipo JavaSpaces.

Mecanismo

FT-Linda, Plinda2.0 y MCM son los únicos en ofrecer un servicio de transacciones. Con excepción de Plinda2.0, el resto de los desarrollos utilizan un mecanismo de replicación como medida para aumentar la disponibilidad del servicio.

FT-JavaSpaces no ofrece un servicio de transacciones y emplea una técnica de replicación.

Plinda2.0, Globe, MCM y FT-JavaSpaces se caracterizan específicamente por permitir la recuperación de un espacio de tuplas. Plinda2.0, MCM y FT-JavaSpaces realizan inmediatamente el traspaso del último estado interno consistente a un espacio nuevo; mientras que Globe lo hace de forma gradual.

Ops./Func.

Una clasificación extra de los proyectos puede realizarse si se toma en cuenta el número de operaciones del modelo de Linda que incorporan. FT-Linda, Globe, FT-JavaSpaces[9] y PLinda 2.0 incluyen todas las operaciones de Linda, mientras que MOM, el trabajo de Tam y Woodward y el trabajo de Xu y Liskov sólo consideran las operaciones básicas – in, out y rd.

Los trabajos en [ea93a] y [Kam91b] proponen también un espacio de tuplas tolerante a fallas utilizando la replicación. Sin embargo, estas propuestas son similares a las anteriores e incluso cuentan con algunas desventajas – algoritmos centralizados o se basan únicamente en simulaciones.

Con excepción de Globe y FT-JavaSpaces, el resto de las implementaciones mencionadas en este capítulo se basan en *hardware* o redes de área local. En contraste, Globe y FT-JavaSpaces son desarrollos hechos en software y que funcionan también en redes como la Internet.

En la actualidad existen otros proyectos que trabajan con réplicas de espacios de tuplas, aunque con motivos distintos. El proyecto titulado *S/Net Linda Kernel* [CG86b] utiliza también un mecanismo de replicación, pero con el propósito de facilitar la computación paralela. El trabajo en [ea92] utiliza distintos protocolos para la actualización de réplicas ("replica update

[9] Con excepción de eval().

protocols"), dependiendo de las características de rendimiento ("performance") requeridas. Krishnaswany [V.91], por otra parte, propone un nuevo espacio de tuplas escalable utilizando la replicación parcial de los espacios.

Capítulo 8

Conclusiones

Esta tesis estudia el problema de la tolerancia a fallas en la implementación de un espacio de tuplas. El objetivo final era proponer el diseño y desarrollar un espacio de tuplas tolerante a fallas.

La propuesta de esta tesis se denomina FT-JavaSpaces y comienza por el establecimiento de sus requerimientos y de los supuestos principales en su ambiente de implementación.

El siguiente paso consistió en elegir una técnica de tolerancia a fallas específica para la implementación. FT-JavaSpaces utiliza un método de enmascaramiento de fallos que utiliza dos implementaciones de un espacio de tuplas; es decir, un tipo de redundancia física. En este sentido, uno de los espacios es denominado primario y el restante como de respaldo. El servicio de FT-JavaSpaces se mantiene de esta forma mientras exista al menos uno de los espacios en funcionamiento. Además, FT-JavaSpaces logra reincorporar a uno de los espacios que haya sufrido un fallo mediante la actualización de su estado interno con la ayuda del otro espacio.

FT-JavaSpaces utiliza una versión orientada a objetos de un espacio de tuplas para sus espacios primario y de respaldo denominada JavaSpaces y original de Sun Microsystems. El manejo de ambos se realiza por medio de un Front-End que a su vez exporta cinco de las operaciones del modelo Linda, es decir, rd(), in(), rdp() e inp().

Internamente, FT-JavaSpaces trabaja con el espacio primario para cumplir [illegible]

interno del espacio de respaldo. Sin fallos que tolerar, FT-JavaSpaces garantiza una evolución consistente del estado interno del espacio de respaldo en relación con la del primario. De esta forma, el espacio de respaldo puede tomar el lugar del primario en el caso de que éste último experimente un fallo. En la situación de que el espacio de respaldo falle, FT-JavaSpaces depende solamente del espacio primario para seguir proporcionando su servicio.

En cuanto a los resultados, las aplicaciones del cálculo del fractal de Mandelbrot y el contador infinito permitieron comprobar que FT-JavaSpaces enmascara efectivamente la caída del servidor o del canal de comunicación así como un fallo de omisión en una implementación de un espacio de tuplas.

La evaluación cualitativa de FT-Javaspaces ratificó el cumplimiento de los requerimientos de diseño. En primer lugar, FT-JavaSpaces conserva la funcionalidad y semántica del modelo Linda. Por otra parte, la disponibilidad en el servicio de FT-JavaSpaces es mayor a la de una implementación única de un espacio de tuplas gracias a sus espacios primario y de respaldo. Es decir, la probabilidad de que FT-JavaSpaces preste su servicio es mayor a la de una sola implementación no tolerante a fallas de un espacio de tuplas. En cuanto a su disponibilidad, FT-JavaSpaces ofrece a sus aplicaciones una mayor probabilidad de terminar con éxito su trabajo debido a que es capaz de recuperarse de un fallo en el transcurso de la operación de las mismas.

Por otra parte, FT-JavaSpaces ofrece también una vista transparente de sus mecanismos propios. De esta forma, los clientes del mismo no son capaces de diferenciarlo de una implementación de un espacio de tuplas no tolerante a fallas. Es decir, un fallo en el espacio de primario o de respaldo es completamente transparente a los clientes de FT-JavaSpaces.

La evaluación cuantitativa de FT-JavaSpaces demuestra teóricamente como sus niveles de disponibilidad y confiabilidad son mayores a las de una implementación no tolerante a fallas de un espacio de tuplas. En cuanto a su grado de confiabilidad por ejemplo, el tiempo promedio entre fallos (MTTF) de FT-JavaSpaces es de 18 años cuando utiliza para sus espacios primario y de respaldo una implementación con un tiempo promedio entre fallos de apenas una semana y de siete minutos como tiempo promedio de reparación. Estos números demuestran la ventaja de utilizar FT-JavaSpaces en lugar de una implementación no tolerante a fallas de un espacio de tuplas.

8.1 Trabajo a Futuro

El diseño e implementación de FT-JavaSpaces provee de posibilidades de extenderse en diferentes aspectos:

- Procesos tolerantes a fallas. En la actualidad, FT-JavaSpaces no cuenta con un algoritmo para que los *Front-Ends* guarden su estado interno actual en una computación dentro del grupo redundante. Con un algoritmo de esta clase, esta implementación ofrecería también procesos tolerantes a fallas.

- Escalabilidad. Eventualmente, el uso únicamente de un espacio de tuplas *primario* restringe el número de clientes que pueden recibir una atención de calidad. Por otra parte, un espacio *primario* es capaz de almacenar hasta cierto número de datos de acuerdo al tamaño de la memoria que posee la máquina donde reside. Sin embargo, ningún tamaño de memoria es suficiente para una aplicación comercial que funcione en Internet, por ejemplo. En conclusión, es necesario encontrar un esquema para el crecimiento en el número de los datos y en el número de procesos participantes.

- Nuevas operaciones. La creación de nuevas operaciones que trasladen el contenido completo de una implementación a otra puede ser deseable en ciertas situaciones. Por ejemplo, el manejo de versiones de un documento.

- Mayor cantidad de réplicas de respaldo. El número de réplicas de respaldo es deseable que sea modificable conforme a las necesidades cambiantes en el grado de disponibilidad y confiabilidad de una aplicación.

- El manejo de transacciones. Una manera interesante de completar el desarrollo de FT-JavaSpaces es investigar la forma de añadir un mecanismo de transacciones en las operaciones de FT-JavaSpaces. De esta forma, la actualización de una tupla o modificación del estado interno de FT-JavaSpaces podría ser tolerante a fallos en sus clientes.

- Utilización en el trabajo colaborativo. La WWW es actualmente una infraestructura con gran potencial para el desarrollo de aplicaciones distribuidas gracias a su difusión, sus estándares abiertos y a las características programables de sus servicios. En particular, la WWW

promueve la creación de nuevas tecnologías cuyo objetivo es el de facilitar el trabajo en equipo a través de la red y sin barreras de espacio o de tiempo. En general, estas nuevas tecnologías se clasifican como "groupware". Algunos ejemplos de aplicaciones del tipo "groupware" son los sistemas de manejo compartido de documentos de texto, gráficos, multimedia, etc.

En la práctica, el trabajo cooperativo asistido por computadora ("Computer Supported Cooperative Work", o CSCW) estudia el diseño, adopción y uso de las tecnologías del tipo "groupware". Este estudio incluye aspectos técnicos de colaboración y trabajo en conjunto junto con aspectos sociales, psicológicos y de organización.

Por otra parte, el estado actual de la WWW presenta también dificultades para la coordinación entre los distintos agentes[1] participantes de una aplicación que lleva a la práctica el trabajo cooperativo asistido por computadora: la necesidad de protocolos para la comunicación de grupos, el establecimiento de un orden global, la resolución de consenso, la atomicidad de las operaciones, la conciencia de grupo, interfaces de usuario, etc.

El modelo Linda ofrece de manera singular una herramienta útil para la coordinación de diferentes agentes. En particular, el modelo de Linda es adecuado para la implementación de patrones de flujo ("Workflow patterns"). Un patrón de flujo se caracteriza por que cuando un agente termina de manipular un elemento de información, éste último debe ser replicado o actualizado para su utilización posterior por otros agentes.

La información sobre los agentes y los mensajes entre los mismos pueden representarse mediante tuplas que residen en uno o varios espacios y que pueden ser manipuladas por un número reducido de primitivas de coordinación[2], propias del modelo de Linda. Una tupla que representa a un agente puede guardar información relevante al mismo como su nombre, características, permisos, etc. Por su parte, un mensaje

[1] En este contexto un agente es una entidad que puede actuar de manera autónoma, así como enviar y recibir mensajes de acuerdo a un protocolo de comunicación establecido.
[2] Out(), in(), rd(), inp() y rdp().

puede contener información acerca de sus destinatarios, un resumen del mismo, etc.

En general, los agentes que manejan las tuplas pueden contar con diferentes propósitos: como interfaz de un sistema con un cliente, como representante característico de un usuario, como prestador de un servicio o de enlace con otros servidores.

De esta forma, un espacio de tuplas sirve como un ambiente virtual de cooperación para la interacacción entre varios agentes. Esta abstracción sirve para describir una plataforma de trabajo cooperativo, la cual provee de un marco para que los usuarios se comuniquen entre si y manejen conjuntamente recursos como documentos, audio, video, etc.

La implementación de un ambiente virtual de cooperación mediante un espacio de tuplas se basa comúnmente en el concepto de salón ("Rooms") y jugadores ("Players"), o usuarios ("users"). Existen distintos salones con propósitos y temas distintos. Cada usuario puede ingresar a un salón e interactuar únicamente con los elementos de datos presentes (i.e, modificar, agregar, eliminar información, etc.). Además, los usuarios dentro de un salón se comunican solamente con otros agentes dentro del mismo. Evidentemente, un usuario puede contar con diferentes sesiones de salón.

En la práctica, cada salón corresponde a la implementación de un espacio de tuplas y los elementos de datos pueden representarse mediante una o mas tuplas.

Entre los ejemplos de aplicaciones que pueden beneficiarse de este patron de flujo entre agentes están los juegos en tercera dimensión, el proceso de recopilación de artículos para revistas, servicios de mensajes, etc.

Apéndice A

Rendimiento

Op.	Tiempo en ms									Promedio (ms)
out()	15	15	34	18	19	20	24	30	22	19.7
in()	20	16	17	32	17	16	16	16	18	16.8
rd()	16	19	16	18	33	16	15	15	15	16.3
inp()	19	16	17	22	19	19	20	16	23	18.1
rd()	22	37	16	16	15	16	17	17	18	16.4

Tabla A.1: Tiempo en ms para las operaciones con JavaSpaces.

Op.	Tiempo en ms									Promedio (ms)
out()	107	86	71	111	93	76	97	83	84	80.8
in()	67	64	117	71	68	68	66	95	144	76 0
rd()	44	37	58	40	41	40	66	48	40	41.4
inp()	69	70	68	83	94	65	64	63	159	73.5
rdp()	25	25	26	26	26	26	25	25	46	25.0

Tabla A.2: Tiempo en ms para las operaciones con FT-Javaspaces, versión centralizada.

puede contener información acerca de sus destinatarios, un resumen del mismo, etc.

En general, los agentes que manejan las tuplas pueden contar con diferentes propósitos: como interfaz de un sistema con un cliente, como representante característico de un usuario, como prestador de un servicio o de enlace con otros servidores.

De esta forma, un espacio de tuplas sirve como un ambiente virtual de cooperación para la interacacción entre varios agentes. Esta abstracción sirve para describir una plataforma de trabajo cooperativo, la cual provee de un marco para que los usuarios se comuniquen entre si y manejen conjuntamente recursos como documentos, audio, video, etc.

La implementación de un ambiente virtual de cooperación mediante un espacio de tuplas se basa comúnmente en el concepto de salón ("Rooms") y jugadores ("Players"), o usuarios ("users"). Existen distintos salones con propósitos y temas distintos. Cada usuario puede ingresar a un salón e interactuar únicamente con los elementos de datos presentes (i.e, modificar, agregar, eliminar información, etc.). Además, los usuarios dentro de un salón se comunican solamente con otros agentes dentro del mismo. Evidentemente, un usuario puede contar con diferentes sesiones de salón.

En la práctica, cada salón corresponde a la implementación de un espacio de tuplas y los elementos de datos pueden representarse mediante una o mas tuplas.

Entre los ejemplos de aplicaciones que pueden beneficiarse de este patrón de flujo entre agentes están los juegos en tercera dimensión, el proceso de recopilación de artículos para revistas, servicios de mensajes, etc.

Op.	Tiempo en ms									Promedio (ms)
out()	159	133	155	171	132	161	184	152	141	154.22
in()	340	444	336	367	450	287	490	304	562	397.77
rd()	461	287	441	244	372	283	379	296	329	343.55
inp()	426	283	295	336	392	271	444	281	314	338.0
rdp()	288	348	410	329	339	273	263	357	240	316.33

Tabla A.3: Tiempo en ms para las operaciones con FT-Javaspaces, versión distribuida.

Bibliografía

[Bab] O. Babaoglu. The engineering of fault-tolerant distributed computing systems. In *Fault-Tolerant Distributed Computing*, number 448 in Lectures Notes in Computer Science, pages 262–274. Springer Verlag.

[BG88] D. Bitton and J. Gray. Disk shadowing. In *Proceedings of the 14th VLDB Conference*, pages 331–338, 1988.

[Bir93] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–103, Decembre 1993.

[BJ] K.P. Birman and T.A. Joseph. Communication support for reliable distributed computing. In *Fault-Tolerant Distributed Computing*, number 448 in Lectures Notes in Computer Science, pages 124–138. Springer Verlag.

[Bjo93] R.D. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, 1993.

[BM95] P. Bhatt and R. McBride. A front-end for fault tolerant distributed systems. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 411–414. IEEE, 1995.

[Bri98] F. Briggs. Synchronization, coherence, and event ordering in multiprocessors. In Butler J., editor, *The Cache Coherence Problem in Shared-Memory Multiprocessors*, Software Solutions. IEEE Computer Society Press, 1998.

[BS95] D.E. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, 1995.

BV] F. Baiardi and M. Vanneschi. Desing of highly decentralized operating systems. In *Distributed Operating Systems*, volume 28 of *NATO ASI Series*. Springer-Verlag.

CD94] S. Cannon and D. Dunn. Adding fault-tolerant transaction processing to linda. *Software-Practice and Experience*, 24(5):449–466, May 1994.

CG86a] N. Carriero and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

CG86b] N. Carriero and D. Gelernter. The s/net's linda kernel. *ACM Transactions on Computer Systems*, 4(2), 1986.

CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

CG93] N. Carriero and D. Gelernter. Linda and message passing: What have we learned? Technical report, Yale University, August 1993.

Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Comms. ACM*, 34(2), 1991.

CT96] T.D. Chandra and S. Toueg. Unreliable failures detectors for reliable distributed systems. *Journal of the ACM*, 43(2):245–267, 1996.

eaa] C. Fuhrman et al. Hardware/software fault tolerance with multiple task modular redundancy. In *IEEE Symposium on Computers and Communications*, pages 171–177. IEEE.

eab] L. V. Kalé et al. Parallel prolog on the intel ipsc/2. In *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*.

eac] U. Fritze et al. Fault-tolerant total order multicast to asynchronous groups. In *Symposium on Reliable Distributed Systems*, pages 228–234. ACM.

ea85] D. Gelernter et al. Parallel programming in linda. Technical Report YALEU/DCS/RR No. 359, Yale University, January 1985.

ea91] S.K. Shrivastava et al. An overview of the arjuna distributed programming system. *IEEE Software*, 1991.

[ea92] C. Shigeru et al. Exploiting a weak consistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 416–423. IEEE Computer Society Press, June 1992.

[ea93a] L.I. Patterson et al. Construction of a fault-tolerant distributed tuple-space. In ACM/SIGAPP, editor, *Proc. 1993 Symp. Appl. Comput.*, pages 279–285, February 1993.

[ea93b] N. Budhiraja et al. *Distributed Systems*, chapter 8. Addison-Wesley, 1993.

[ea95a] G. Coulouris et al. *Distributed Systems, Concepts and Design.* Addison-Wesley Publishing Company, Harlow, Inglaterra, 2nd edition, 1995.

[ea95b] G. Coulouris et al. *Distributed Systems Concepts and Design*, chapter 17, page 520. 1995.

[ea96a] J. Vlissides et al. *Pattern Languages of Program Design.* Number 2. Addison-Wesley, 1996.

[ea96b] P. Ciancarini et al. Pagaspace: An architecture to coordinate distributed applications on the web. In *Fifth International World Wide Web Conference Conference*, http://www.w3.org/Conferences/5, 1996.

[ea98a] J. Protić et al. *An Overview of Distributed Shared Memory*, chapter 1, pages 12–41. IEEE Computer Society, 1998.

[ea98b] J. Protić et al. An overview of distributed shared memory. In Milutinovic V. Protic J., Tomasevic M., editor, *Distributed shared memory: concepts and systems*, chapter 1. IEEE Computer Society Press, 1998.

[ea98c] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Distributed shared memory: concepts and systems*, pages 84–95. IEEE Computer Society, 1998.

[ea98d] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In M. Milutinovic V.

Protic J., Tomasevic, editor, *Distributed shared memory: concepts and systems*, chapter 3. IEEE Computer Society Press, 1998.

[ea98e] P. Wyckoff et al. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[ea99] E. Freeman et al. *JavaSpaces Principles, Patterns, and Practice*. The Jini Technology Series. Addison-Wesley Publishing Company, 1999.

[Fin] S.J. Finkelstein. Algorithms and system design in the highly available systems project. In *Fault-Tolerant Distributed Computing*, number 448 in Lectures Notes in Computer Science, pages 138–147. Springer Verlag.

[Gar98] B. Garbinato. *Protocol Objects and Patterns for Structuring Reliable Distributed Systems*. PhD thesis, Ecole Polytechnique Federale de Lausanne, 1998.

[Gel85] D. Gelernter. Generative communication in linda. *Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Gol88] B. F. Goldberg. *Multiprocessor Execution of Funcional Programs*. PhD thesis, Yale University, 1988.

[Gra] J. Gray. A comparison of the byzantine agreement problem. In *Fault-Tolerant Distributed Computing*, number 448 in Lectures Notes in Computer Science, pages 10–18. Springer Verlag.

[GS] G. A. Geist and V. S. Sunderam. The pvm system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Proceedings of the Sixth Distributed Memory Computing Conference*.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, Abril 1997.

[Had] V. Hadzilacos. On the relationship between the atomic commitment. In *Fault-Tolerant Distributed Computing*, number 448 in Lectures Notes in Computer Science, pages 201–209. Springer Verlag.

[HC] R.K. Hansen and S.R. Cannon. An efficient fault-tolerant tuple space. In *Fault-Tolerant Parallel and Distributed Systems*, pages 210–225.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Dept. of Computer Science, Cornell University, USA, 1994.

[Joh89] B. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*, chapter Fundamental Definitions. Addison-Wesley Publishing Company, 1989.

[Kam91a] S. Kambhatla. Replication issues for a distributed and highly available linda tuple space. Master's thesis, Birla Institute of Tecnhology and Science, Pilani, India, February 1991.

[Kam91b] S. Kambhatla. Replication issues for a distributed and highly available linda tuple space. Master's thesis, Dep. Comput. Sci., Oregon Graduate Inst., February 1991.

[KP93] H. Kopetz and V. Paulo. Real time and dependability concepts. In Mulder Sape, editor, *Distributed Systems*, chapter 16, pages 411–444. Addison-Wesley, 2nd edition, 1993.

[Lam78a] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks 2*, pages 95–114, 1978.

[Lam78b] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lei89] J.S. Leichter. Shared tuple memories, shared memories, buses and lans-linda implementations across the spectrum of connectivity. Technical report, Yale Univeristy, 1989.

[LS] K. Li and R. Schaefer. Shared virtual memory for a hypercube multiprocessor. In *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications.*

[LS99] J.E. Larsen and J.H. Spring. Globe, global object exchange. Master's thesis, University of Copenhagen, Copenhagen, Denmark, October 1999.

Mic99] Sun Microsystems. *Java Remote Method Invocation Specification.* http://www.sun.com/products/1.2/docs/guide/rmi/index.html, 1999.

Mul93] S. Mullender, editor. *Distributed Systems.* ACM Press, 2nd edition, 1993.

Nit98] B. Nitzberg. Distributed shared memory: A survey of issues and algorithms. In M. Milutinovic V. Protic J., Tomasevic, editor, *Distributed shared memory: concepts and systems*, chapter 1, pages 42–50. IEEE Computer Society Press, 1998.

Pan94] J. Pankaj. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1994.

Par93] J. Paris. The management of replicated data. In *Proc. Workshop on Hardware and Software Architectures for Fault-Tolerance: Perspectives and Towards a Synthesis*, 1993.

[Par94] J. Paris. An available copy protocol tolerating network partitions. In *Conference on Computer and Communications*, pages 77–83, 1994.

[Pow96] D. Powell. Group communication. *Communications of the ACM*, 1996.

[Ray] M. Raynal. Non-blocking atomic commitment in distributed systems: A tutorial based on a generic protocol. Technical report, IRISA.

[RW96] A. Rowstron and A. Wood. Solving the linda multiple rd problem. In *Lecture Notes in Computer Science 1061*, pages 357–367. Springer-Verlag, 1996.

[Sch84] F.B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems 2(2)*, pages 145–154, 1984.

[Sch90] F. Schneider. Implementing fault-tolerant services using the state machine approach. In *ACM Comput. Surv.*, volume 22, pages 299–319. December 1990.

[Sch93] F. Schneider. What good are models and what models are good. In Sape Mulder, editor. *Distributed Systems*, chapter 2, pages 17–26. Addison Wesley, 2nd edition, 1993.

[SJ] D. Shasha and K. Jeong. Plinda 2.0: A transactional/checkpointing approach to fault tolerant linda. In *Proc. Thirteenth Symp. on Reliable Distrib. Syst.*, pages 96–105.

[SM96] J. Sussman and K. Marzullo. Comparing primary-backup and state machines for crash failures. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, page 90, 1996.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[SV97] A. Somani and N. Vaidya. Understanding fault tolerance and reliability. *Computer*, 30(4):45–50, April 1997.

[TR85] Andrew S. Tanenbaum and Robert Van Renesse. Distributed operating systems. In *Computing Surveys*, volume 17. Association for Computing Machinery, December 1985.

[TW95] F. Tam and M. Woodward. Ft-linda: A coordination language for programming distributed fault-tolerance. In *Proceedings of IEEE Singapore International Conference on Networks*. IEEE, 1995.

[V.91] Krishnaswamy. V. *A Language Based Architecture for Parallel Computing*. PhD thesis, Yale University, 1991.

[XL89] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of linda. In *Proc. Nineteenth Int. Symp. Fault-Tolerant Comput.*, pages 199–206, June 1989.