UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"LA TOLERANCIA A FALLAS OPCIONAL EN EL MIDDLEWARE"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA (COMPUTACIÓN)

P R E S E N T A :

MARIO ERNESTO ISAURO MARTÍNEZ

DIRECTORA DE TESIS: DRA. ELIZABETH PÉREZ CORTÉS

MÉXICO, D.F. NOVIEMBRE, 2004

Tesis desarrollada en el marco del Proyecto CONACyT: 34230 A
"Infraestructura para el Desarrollo de Aplicaciones Distribuidas"

Resumen

Como consecuencia de la utilización generalizada de los sistemas de cómputo distribuido en la vida cotidiana los requerimientos de fiabilidad de una aplicación distribuida han aumentado. Por esta razón se observa una creciente preocupación por dotar a las aplicaciones de elementos que atenúen o eliminen las consecuencias de las fallas (tolerancia a fallas) en los elementos que las componen.

La construcción de una aplicación distribuida tolerante a fallas no es una tarea simple, ya que debido a la cantidad y la naturaleza de los componentes que la integran, el tipo y la cantidad de fallas aumentan. Por otro lado, dependiendo de muchos factores, la tolerancia a fallas tiene un costo que no se puede imponer a todas las aplicaciones.

El trabajo que aquí se presenta explora la posibilidad de descargar al constructor de aplicaciones distribuidas de la programación de la tolerancia a fallas dejándola a cargo del middleware. Además, propone un esquema que permite al constructor de aplicaciones incluir los mecanismos de tolerancia a fallas disponible en el grado que le parezca más adecuado: para toda la aplicación o bien para algunos de los elementos.

Este documento contiene las síntesis de los estudios realizados para alcanzar nuestro objetivo, la descripción de la propuesta así como su validación. Hemos obtenido entonces:

- Un panorama amplio de los mecanismos de tolerancia a fallas y los sistemas distribuidos en general.
- Una síntesis de la forma en que funcionan los middlewares DCOM, CORBA, CORBAFT, EJB, CCM y Séneca así como la identificación de los mecanismos que estos ofrecen para tolerar las fallas.
- Una propuesta para construir un middleware que ofrezca mecanismos de tolerancia a fallas a los constructores de aplicaciones distribuidas permitiéndoles diferentes grados de integración.
- La validación de la propuesta en las especificaciones de EJB y Séneca, dando origen a las especificaciones de EJBTaF y SénecaTaF, donde la percepción del nivel de tolerancia por parte de los desarrolladores de aplicaciones distribuidas es distinta para cada especificación.
- Un prototipo de SénecaTaF.

El estudio termina concluyendo que es viable contar con un middleware a componentes tolerante a fallas que simplifique la tarea de construir aplicaciones distribuidas fiables. Al mismo tiempo corroboramos que no es necesario imponer a todas las aplicaciones usuarias el costo, o el mismo costo, de implementar la tolerancia a fallas.

SUMARIO DE CONTENIDO

CONTENIDO

1 INTRODUCCION

1.1 Antecedentes del problema

El auge de las redes de computadoras y su amplio uso en el mundo dieron origen a una nueva forma de cómputo: el cómputo distribuido. En este nuevo contexto, el cómputo se fragmenta y se distribuye a través de la red en varias computadoras que no comparten memoria, ni reloj – computadoras débilmente acopladas- y que se comunican entre ellas a través mensajes que son transmitidos por diversos medios. Esto dio origen a los sistemas distribuidos. El término sistema distribuido designa a un conjunto de procesos, que se comunican y sincronizan para realizar una tarea en común y que se ejecutan sobre un conjunto de computadoras interconectadas.

La construcción de una aplicación distribuida es una tarea compleja. Una propuesta importante para reducir dicha complejidad, por una parte, ha sido la separación de las tareas a resolver en este tipo de aplicaciones. Esta separación permite distinguir las tareas aplicativas de aquéllas que no lo son. El resultado de esta separación da origen a la arquitectura middleware. En esta arquitectura la capa alta de un sistema distribuido lleva a cabo las tareas aplicativas y la capa de en medio o middleware, en general, se encarga de todas aquéllas que surgen del uso de la infraestructura distribuida. En el nivel más bajo de la arquitectura se tiene finalmente al sistema operativo. Por otra parte, los paradigmas de programación evolucionaron para contender con la complejidad de los sistemas distribuidos. En particular nos referimos al desarrollo de software basado en componentes (DSBC), que trata de sentar las bases para el diseño y desarrollo de aplicaciones distribuidas basadas en software reutilizable.

Dado el creciente uso de sistemas distribuidos en misiones donde la confiabilidad es crítica, implica evitar que las fallas tengan un impacto en el funcionamiento del sistema. Como consecuencia, cada vez es más importante construir aplicaciones tolerantes a fallas. La tolerancia a fallas nos permite mejorar la disponibilidad de los sistemas e incrementa su confiabilidad. Esto se logra principalmente a través de redundancia en piezas clave de hardware o software, de modo que si una de estas piezas falla, otra puede ocupar su lugar. Implementar una redundancia consistente se logra a través de diversos mecanismos que tienen una complejidad considerable a la hora de su implementación.

Por esta razón, se puede decir que un problema interesante a resolver es el de disminuir el trabajo de un constructor de aplicaciones distribuidas haciendo que las tareas relativas a la tolerancia a fallas queden a cargo del middleware.

1.2 Situación actual del problema y objetivo de la tesis

A pesar de la actual utilización sistemática de la arquitectura middleware, la mayoría de las especificaciones de esos middlewares no ofrecen un soporte explícito para la tolerancia a fallas. En este caso, el programador de aplicaciones distribuídas debe de programar la tolerancia a fallas de la manera tradicional. Esta estrategia para abordar el problema no es coherente con la tendencia actual de descargar al programador de aplicaciones de las tareas no aplicativas.

Por otro lado, cabe mencionar que algunas implementaciones, sí integran soporte para esta problemática. Desafortunadamente, cuando esto sucede, normalmente ocurre bajo un esquema de todo o nada en el sentido que el constructor decide si integra o no los mecanismos propuestos a la aplicación. Consideramos que esta no es la solución adecuada pues no todas las aplicaciones requieren el mismo nivel de tolerancia a fallas ni disponen de la misma infraestructura física. Por lo

tanto, la inclusión de la tolerancia a fallas en un middleware debe de ser opcional y con distintos niveles de integración.

Objetivo del trabajo de tesis:

> Estudiar las especificaciones de middleware más usadas en el mercado, y hacer una propuesta para integrar en los middlewares a componentes un servicio específico de tolerancia a fallas que permita elegir la granularidad a la que se ejecuta según las necesidades de la aplicación.

1.3 Propuesta

Hacemos una propuesta de un modelo específico de tolerancia a fallas el cual se agrega a un middleware a componentes buscando hacerle los menores cambios posibles.

El servicio es opcional y puede incluirse transparentemente en el middleware y ejecutarse para todas las aplicaciones, o bien incluirse sólo si el constructor lo especifica explícitamente.

En este último caso, el constructor decide la granularidad a la que se aplicará el servicio: a la aplicación completa o bien sólo a algunos de los componentes.

Para nuestro trabajo, el modelo elegido tolera una falla de servidor de tipo caída-pausa, ver sección 2.2.2, mientras que la redundancia se hace a nivel de componentes usando un respaldo para cada uno de ellos.

Para la validación de la propuesta, el servicio de las características enunciadas se integra a dos modelos de componentes que cumplen con la arquitectura cliente-servidor-contenedor (ver sección 4.2). Posteriormente se realiza la implementación parcial en uno de ellos.

Por último se hace una evaluación cualitativa de la propuesta. Cabe mencionar que la evaluación cuantitativa (rendimiento, confiabilidad, etc) no es contemplada en este trabajo.

1.4 Estructura de la tesis

Parte 1 – Conceptos básicos y estado del arte

En esta parte, se muestra la definición de fallas y su clasificación, junto con los que consideramos como principales mecanismos de tolerancia a fallas. También se muestran las principales especificaciones de middlewares en el mercado como son: DCOM, CORBA, CORBA tolerante a fallas, CCM, EJB y Séneca, así como los mecanismos que incorporan estas especificaciones para la tolerancia a fallas.

Parte 2 – Propuesta sobre tolerancia a fallas e incorporación en modelos ya existentes

En esta parte, se muestra una propuesta de cómo combinar los mecanismos de redundancia, uso de bitácoras, puntos de revisión y técnicas de respaldo primario para agregar tolerancia a fallas en los middlewares, esta propuesta es llamada modelo abstracto. También se muestra la forma en que se integra dicha propuesta en el modelo EJB, que es una especificación de Sun, y en el modelo Séneca,

que es un proyecto académico, ambos son modelos de middlewares a componentes. Los modelos resultantes de la integración son llamados EJB tolerante a fallas y Séneca tolerante a falla respectivamente.

Parte 3 –Conclusiones

En esta parte, se menciona qué mecanismos del modelo abstracto pudieron ser integrados en EJBTaF y SénecaTaF y la dificultad que se encontró para hacerlo. También se mencionan las conclusiones a las que se llega después de implementar parcialmente el prototipo de SénecaTaF-j así como las perspectivas en trabajos futuros.

2 LOS SISTEMAS DISTRIBUIDOS Y LA TOLERANCIA A FALLAS

El avance de las telecomunicaciones y la informática así como la utilización sistemática de las redes de computadoras en el mundo de los negocios dieron origen a una nueva forma de cómputo: el cómputo distribuido. En este nuevo contexto, el cómputo se fragmenta y se distribuye a través de la red en varias computadoras que no comparten memoria, ni reloj -*computadoras débilmente acopladas*-. Cada computadora tiene su propia memoria **local** y se comunican entre ellas a través de diversos medios, como buses de alta velocidad, líneas telefónicas, etc.

Entre las razones más importantes para distribuir el cómputo tenemos:

- **Compartir recursos**. Los recursos de una computadora están disponibles para todos los usuarios de la red.
- **Aumentar eficiencia**. Si un cálculo se puede dividir en varios subcálculos susceptibles de ejecutarse en paralelo, el cálculo podría hacerse en distintas computadoras aumentando la eficiencia.
- **Aumentar la confiabilidad**. Varias computadoras podrían ejecutar el mismo cómputo y al finalizar comparar resultados.
- **Aumentar la disponibilidad**. Si una computadora falla, las computadoras restantes podrían hacer la parte del cómputo que le corresponde a la que falló.

Como ya lo mencionamos, el cómputo distribuido requiere de varias computadoras (hardware) interconectadas, y de un sistema que administre la distribución de los datos y la carga de trabajo en las distintas computadoras (software). En el ánimo de estandarizar el concepto de lo que es un sistema distribuido, muchos autores se han dado a la tarea de definirlo sin llegar a un consenso [2].

En el contexto de este trabajo, el término **sistema distribuido** designa un conjunto de procesos, que se comunican y sincronizan para realizar una tarea en común y que se ejecutan sobre un conjunto de computadoras intercomunicadas. Asimismo, el término **sitio** designa a una de las computadoras que forman parte de un sistema distribuido.

Un objetivo ampliamente perseguido en el cómputo distribuido es hacer transparente la distribución. Esto es, que los usuarios puedan hacer uso del sistema como si se tratara de un sistema centralizado. En particular, se desea hacer trasparente la ocurrencia de fallas, aún cuando, por la forma en que están construidos y la cantidad de elementos que los componen, los sistemas distribuidos son más susceptibles que los centralizados a exhibir comportamientos que difieren de su especificación. Afortunadamente, a la fecha, ya se han realizado muchos esfuerzos con logros importantes para construir sistemas distribuidos más confiables.

En este capítulo presentamos las bases para la construcción de sistemas distribuidos (en la sección 2.1) así como las principales técnicas para paliar su naturaleza falible (sección 2.2). Finalmente, en la sección 2.3 se presenta una discusión acerca de la integración de las técnicas estudiadas en los sistemas actuales.

2.1 Construcción de aplicaciones distribuidas

Un sistema distribuido puede ser concebido de acuerdo a varios modelos e implementado utilizando diversos mecanismos. En esta sección se presentan brevemente los modelos y mecanismos que consideramos más relevantes.

2.1.1 Modelo cliente/servidor

En el modelo cliente/servidor, el sistema se estructura como un grupo de procesos en cooperación, llamados servidores, que ofrecen servicios a los usuarios, llamados clientes. Un sitio puede ejecutar uno o varios clientes, uno o varios servidores, o una combinación de ambos. Un proceso puede actuar como servidor y a la vez ser cliente de otros servidores.

El modelo cliente/servidor se basa en el protocolo sencillo de solicitud/respuesta sin conexión, esto es, el cliente envía una solicitud al servidor para ejecutar cierto servicio, entonces el servidor ejecuta la petición y regresa el resultado. La principal ventaja de este modelo es su sencillez

Debido a la sencillez del modelo, los mecanismos de comunicación necesarios para implantarlo, se pueden reducir a dos llamadas al sistema: una para el envío de mensajes y otra para la recepción. Esta llamadas al sistema se pueden realizar a través de procedimientos de biblioteca, como son *send(dest, &mptr)* y *receive(addr, &mptr)*. Aún cuando esta interfaz es conveniente para la construcción de sistemas distribuidos tiene un problema: el paradigma en que está construida la comunicación es la entrada/salida (E/S). Normalmente, los procedimientos *send* y *receive* están reservados para la realización de E/S, y puesto que la E/S no es uno de los conceptos fundamentales de los sistemas centralizados, el uso de estos procedimientos se ve como un error en la búsqueda de la transparencia. En la dinámica para lograr que los sistemas distribuidos sean vistos por los usuarios como sistemas centralizados, sería más natural que los clientes pudiesen hacer llamados a los procedimientos de los servidores aunque estos se encuentren en un sitio distante.

2.1.2 Llamadas a procedimiento remoto (RPC)

Para ocultar la distribución del sistema, se sugiere permitir a los programas que hagan llamadas a procedimientos que se encuentran en otros sitios (véase Fig. 2.1). Esto es, si un proceso en el sitio A llama a un procedimiento en el sitio B, el proceso en el sitio A se suspende y se ejecuta el procedimiento en B. Cuando se termina la ejecución en B se regresa el resultado y el control al proceso en A. Así, la información puede ser transportada de un lado a otro a través de los parámetros, permitiendo que el programador no se preocupe por la transferencia de los mensajes o de las E/S. De esta forma, el RPC acerca la semántica de las llamadas a procedimientos que no se encuentran en el mismo sitio (entorno distribuido) a una semántica convencional (entorno centralizado) [7][20].

El elemento que hacen posible que un cliente en el sitio A (y que oculta el RPC) pueda hacer una llamada a un procedimiento remoto que se encuentra en un servidor en el sitio B es llamado Stub. Así, el cliente se comunica con el servidor usando un objeto Stub que es responsable de:

i. Inicia una conexión con el servidor que contiene al procedimiento remoto
ii. Empaqueta (marshaling) y transmite los parámetros
iii. Espera por el resultado de la invocación
iv. Desempaqueta (unmarshaling) el valor de retorno

v. Entrega el valor (o resultado) al cliente que lo llamó

Del lado del servidor está el objeto Skeleton que intercepta la petición. El Skeleton es responsable de:

i. Recibir la petición del cliente.
ii. Desempaquetar y reconstruir la llamada para entregarla al correspondiente procedimiento.
iii. Obtiene el resultado de la ejecución del procedimiento.
iv. Empaqueta los resultados y los envía al cliente.

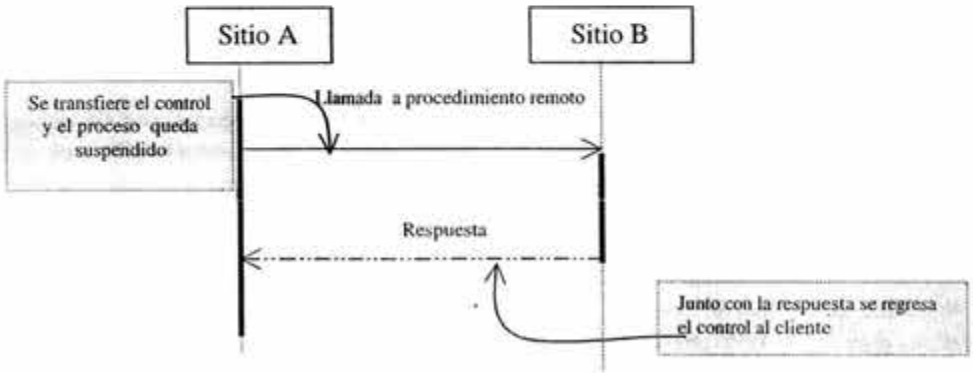


Fig. 2.1 Llamada a procedimiento remoto

La combinación del modelo cliente/servidor y las llamadas a procedimientos remotos son importantes en la construcción de aplicaciones distribuidas.

2.1.3 Capa de servicios o middleware

La integración de nuevas funcionalidades en los sistemas distribuidos ha incrementado su complejidad. Uno de los problemas más frecuentes es la redundancia de código. Por ejemplo, cuando se hace una implementación de un procedimiento remoto que requiriere restringir su uso a ciertos usuarios, y después se implementa otro procedimiento que también requiere tener las mismas precauciones de seguridad, normalmente se duplica el código. Para evitar este problema de duplicidad del código podríamos seguir los siguientes pasos:

i. distinguir y separar en la codificación de los procedimientos remotos, aquellas partes que sirven para el manejo de la red de aquéllas que no sirven para tal propósito,
ii. distinguir y separar de entre el código restante, aquel que es propio de la aplicación y aquellas acciones que se repiten sin importar el proceso, o aquellas cosas que se repiten para ciertas clases de procesos,
iii. "factorizar" y agrupar,

Se agrupan todas las acciones que son propias de la red junto con las tareas que se repiten en la solicitud de ejecución del procedimiento y se crea así una **capa de servicios**. Dicha capa de servicios se localizaría entre los que hacen las llamadas a los procedimientos remotos y aquellos que las ejecutan, (ver Fig. 2.2). A dicha capa de servicios agregando o quitando algunas funcionalidades, según los requerimientos de la aplicación, se le conoce como **middleware**.

La principal tarea de un middleware es esconder el complicado ambiente de la red, lo cual, evita que las aplicaciones manejen explícitamente los protocolos de comunicación, memorias disjuntas, formas de replicación, así como, el paralelismo y la concurrencia. Como segunda tarea importante, el middleware enmascara la heterogeneidad de los sitios, sistemas operativos, lenguajes de programación y la tecnología de la red. Lo que permite facilitar la programación de las aplicaciones y su mantenimiento. Generalmente los sistemas middleware soportan la arbitraria interacción de programas de aplicación. Otras funciones específicas tales como: acceso remoto a bases de datos y manejo de grupos, requieren un middleware especial.

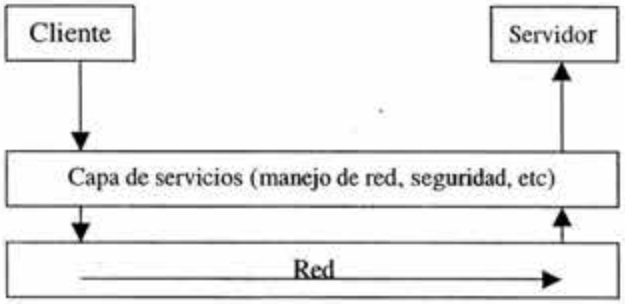


Fig. 2.2 Capa de servicio

Existen varias definiciones de middleware y todas tienen un problema en común [21]: dependiendo del ambiente de la aplicación, las definiciones difieren en cuanto a las partes que componen al middleware. En lo que sí coinciden es en que el middleware se puede dividir en al menos dos partes: una que estará situada en el sitio del cliente y otra en el sitio del servidor. Así que, cuando hablamos de middleware, en términos generales nos referimos a la capa de software que hay entre el sistema operativo –incluyendo los protocolos básicos de comunicación- y las aplicaciones distribuidas que interactúan vía la red. Este software facilita la interacción entre los módulos distribuidos del software de aplicación.

2.1.4 Objetos, objetos distribuidos y componentes

La complejidad derivada de la construcción del middleware y las aplicaciones que lo usan ha rebasado los paradigmas de programación, que fueron pensados para sistemas cerrados (o centralizados), dando paso a nuevas propuestas. En efecto, la aparición de los sistemas distribuidos, ha provocado, entre otras cosas, que los modelos de programación existentes, como la programación orientada a objetos, se vean desbordados, siendo incapaces de manejar de forma natural la complejidad de los requisitos que se les exigen a dichos sistemas [23]. Lo cual, ha provocado la aparición de nuevos paradigmas de programación, entre ellos se encuentra la programación orientada a componentes que intentan subsanar el problema.

Ya que el nombre de objeto y de componente con frecuencia suelen ser usados erróneamente como sinónimos, es necesario enunciar las diferencias entre uno y otro para evitar confusión:

> Un **objeto** clásico es un bloque en el que se encapsulan código y datos. Los objetos clásicos facilitan el reuso de código vía la herencia y el encapsulamiento. Sin embargo, estos objetos clásicos han sido el sustento de la ingeniería de software para los sistemas cerrados, en donde el compilador del lenguaje en que fue creado el objeto es el único que sabe de la existencia de este.

En contraste, un **objeto distribuido** es un objeto clásico pero con la diferencia de que este puede vivir en cualquier parte de la red y puede ser utilizado por cualquier aplicación. Los objetos distribuidos son empaquetados como piezas independientes de código que pueden ser utilizados por clientes remotos vía invocación de sus métodos. El lenguaje y compilador utilizado para crear el servidor de objetos son totalmente transparentes para sus clientes. Los clientes no necesitan saber dónde se encuentran residiendo los objetos distribuidos y/o qué operaciones del sistema se ejecutan sobre él.

Entre las principales características de los objetos distribuidos están las siguientes [29]:

- Un objeto distribuido es una unidad binaria.
- Tienen una *interfaz* bien definida. Como un objeto clásico, el objeto distribuido sólo puede ser manipulado a través de su interfaz.
- Las interfaces de un objeto distribuido determinan las operaciones que el objeto implementa.
- Los objetos distribuidos pueden verse como objetos clásicos a través de las direcciones de espacio, redes, lenguajes, sistemas operativos y herramientas.

Sin embargo, los objetos distribuidos no son suficientes, a menos que seamos capaces de reutilizarlos. Reutilizar un objeto distribuido no significa sólo usarlos más de una vez, sino implica la capacidad de utilizar el objeto distribuido en contextos distintos de aquellos para los que fue diseñado [23]. En esta línea fueron pensados los **componentes,** que cumplen con las características de un objeto distribuido, pero a los cuales se les han agregado algunas otras características que les permiten un mayor grado de reuso.

Aún cuando la diferenciación entre componente y objeto distribuido no es clara en la mayoría de la literatura incluso se usan de manera indistinta, aquí mostramos algunas características que se encuentran en los componentes y que los objetos distribuidos no tienen:

- Los objetos distribuidos presentan algunas limitaciones, por ejemplo: no permiten expresar claramente la diferencia entre los aspectos computacionales y meramente composicionales de la aplicación, definiendo interfaces de muy bajo nivel para que sirvan de contrato entre las distintas partes que deseen utilizar objetos.
- Los componentes no sólo describen los servicios que proveen, sino también describen los servicios que requieren de otros componentes durante su ejecución.
- Los implementadores de componentes no deben de preocuparse por el manejo de su ciclo de vida, ya que este está estandarizado y normalmente es manejado por una plataforma de soporte (con frecuencia por el middleware).
- Existen servicios automáticos con los cuales los proveedores de componentes no deben de preocuparse en programar, como son: seguridad, transacciones y persistencia. Dichos servicios varían dependiendo de su capa de servicios.
- El aumento de servicios automáticos estandariza aun más las interfaces, lo cual permite también que se incremente la *interoperabilidad* entre componentes.

Uno de los sueños de la Ingeniería de software es el de contar con un mercado global de componentes, al igual que ocurre con otras ingenierías (electrónica, civil, etc.). Para ello es necesario que los componentes estén empaquetados de forma que permitan su distribución y composición con otros componentes, especialmente con aquellos desarrollados por terceros.

2.2 Tolerancia a fallas

En esta sección se enuncian los conceptos básicos de la tolerancia a fallas y los mecanismos que permiten implementarla.

2.2.1 Definiciones básicas

La definición más ampliamente aceptada sobre el concepto de **falla** es la dada por Christian [5]: Una falla en un sistema de cómputo ocurre cuando el sistema se aparta de su correcta especificación.

Otros dos conceptos asociados con las fallas son los de falta y error:

> **Falta**.- imperfección de un componente del sistema (software o hardware).
> **Error**.- Manifestación de la falta

Las cuales guardan una relación de causa efecto en donde las faltas son causa de errores y los errores son la causa de las fallas [9].

2.2.2 Clasificación de fallas

Las fallas pueden ocurrir por múltiples circunstancias (como pueden ser falla en la energía eléctrica, defectos en fabricación en el hardware, defectos de diseño en el software, etc.). Sin embargo, el único mecanismo que tienen los sistemas distribuidos para saber si alguno de sus sitios (máquinas integrantes del sistema) ha sufrido una falla, es a través de mensajes, los cuales pueden ser o no ser contestados por el sitio. Cuando un sitio no contesta a un mensaje podría tener una falla o estar extremadamente ocupado de tal forma que no pueda contestar de inmediato el mensaje, o bien, el sitio está bien y la que tiene la falla es la red que los conecta. Christian [5] hace un estudio de las fallas desde el punto de vista de las respuestas que recibe un sitio cliente a sus peticiones y el estado en que se encuentra el servidor después de ejecutar la petición. Y llega a la siguiente clasificación:

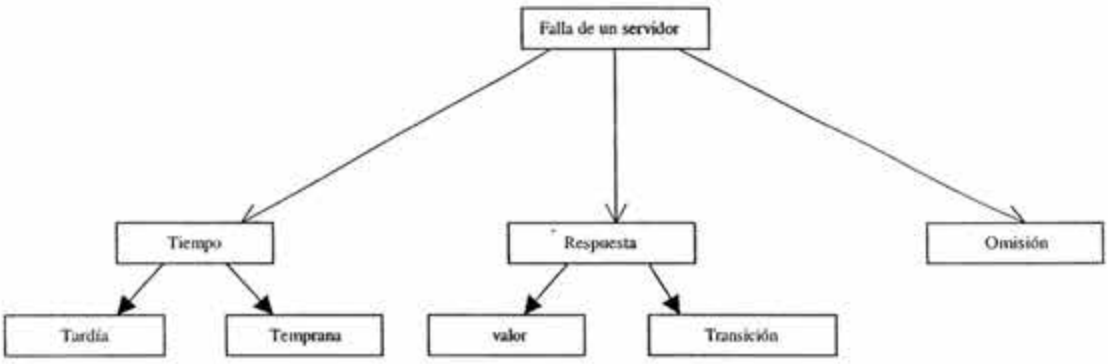


Fig. 2.3 Clasificación de fallas según Cristian (parte 1)

- Una *falla de omisión* ocurre cuando un servidor omite la respuesta a una entrada.
- Una *falla de tiempo* ocurre cuando un servidor responde correctamente pero fuera del intervalo de tiempo especificado. Este tipo de falla tiene una subdivisión que es:

- *falla temprana.*- responde antes del tiempo especificado.
- *falla tardía.*- responde después del intervalo de tiempo especificado.

- Una falla de *respuesta* ocurre cuando el servidor responde incorrectamente. Este tipo de falla tiene una subdivisión que es:
- falla de *valor.*- cuando el valor de salida es incorrecto.

  - falla de *transición.*- el estado que toma el servidor es incorrecto.

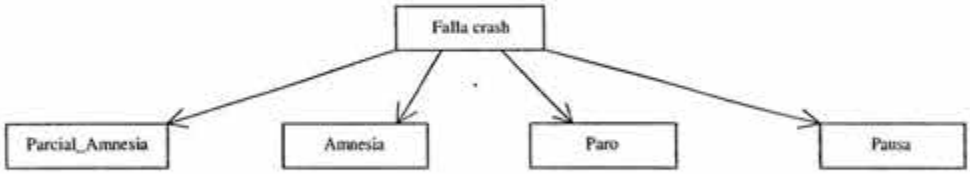


Fig. 2.4 Clasificación de fallas según Cristian (parte 2)

También existen otros tipos de fallas que son clasificadas de acuerdo a la detección de las anteriores fallas (Fig. 2.4). Por ejemplo, detrás de una serie de intentos, un servidor muestra fallas de omisión, entonces se dice que el servidor ha sufrido una falla de tipo caída. Este tipo de fallas se subdivide según el estado que guarda el servidor después de regresar de la falla.

- Una falla de *caída amnésica*, ocurre cuando el servidor regresa de la falla en un estado inicial predefinido, que no tiene que ver en nada con el estado que tenía el servidor hasta antes de la caída.
- Una falla de *caída amnésica parcial*, ocurre cuando el servidor regresa de la falla en un estado que es parte del que tenía antes de la caída y la parte restante es tomada de un estado inicial predefinido.
- Una falla de *caída-pausa*, ocurre cuando el servidor regresa de la falla en el estado en el que se encontraba hasta antes de la caída.
- Una falla de *caída definitiva*, ocurre cuando el servidor no regresa de la falla.

Cuando ocurre una falla en la red que no permite la comunicación entre los sitios, se dice que ocurrió una *partición de red.* Aún cuando no se mencionan en la clasificación anterior, y estén fuera del alcance de este trabajo, mencionaremos que podrían ser detectadas y solucionadas a través de hardware especial [30].

Un sistema ideal podría ser perfectamente fiable y nunca fallar. Sin embargo, esto es imposible de alcanzar en la práctica. No todas las fallas son inevitables, pero en algunos casos los sistemas o los operadores de los sistemas pueden tomar acciones correctivas que prevengan o mitiguen la falla [3]. Para lograr lo anterior, es necesario definir qué tipos de fallas podrán ser detectadas y soportadas por el sistema, a esto se le conoce como la **semántica de fallas del sistema**.

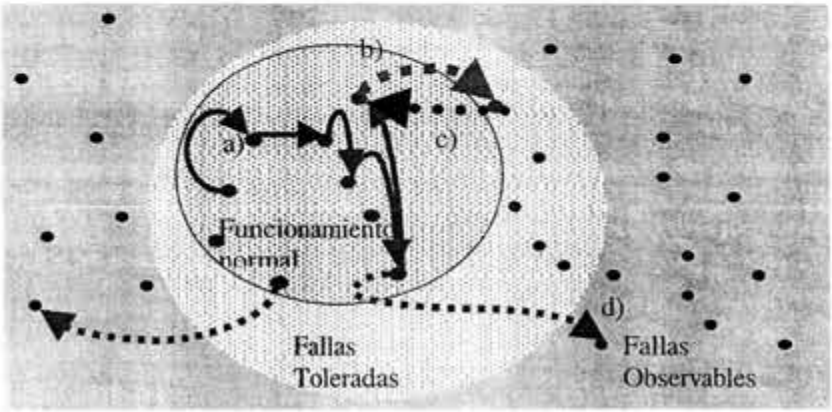
En otras palabras los **sistemas tolerantes a fallas** no son sistemas que no fallan sino que son sistemas que, en presencia de fallas previamente definidas de los servidores en los cuales se apoyan, tienen la habilidad de seguir funcionando de acuerdo a su especificación.

2.3 Técnicas de tolerancia a fallas

Existen varios mecanismos para dotar a los sistemas de una cierta semántica de fallas, en el resto del capítulo estudiaremos los que a nuestro juicio son los más importantes.

2.3.1 Reconfiguración del sistema

Esta técnica como medida de tolerancia a fallas, es el proceso de eliminar a los elementos con fallas de un sistema y devolverlos en un estado operacional, según su semántica de fallas.



Donde:

a) Funcionamiento normal del sistema
b) Ocurrencia de una falla que está definida en la semántica de fallas, por lo cual, el usuario no se percata de la falla
c) Sistema regresa de falla en un estado operacional
d) Ocurrencia de una falla que no está definida en la semántica de fallas, por lo cual, es observable por el usuario

Fig. 2.5 Reconfiguración en presencia de falla

La implementación de esta técnica involucra cuatro pasos secuenciales fundamentales, que son:

- La detección de las fallas
- La ubicación de las fallas
- El confinamiento de las fallas
- La recuperación de las fallas

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

Detección de fallas

La detección de fallas es el proceso de reconocer que una falla se ha producido y su resultado determina estrictamente la política de tolerancia a seguir.

En la práctica, la búsqueda de una falla depende en gran medida del tipo de falla que se busca prevenir y las características del sistema, lo cual da lugar a diferentes tipos de mecanismos de revisión, como son los siguientes:

- Revisión de réplicas. El uso de redundancia permite la comparación de resultados y el empleo de votaciones.
- Revisión de tiempo. El uso de límite de tiempo (time out) permite la detección de fallas en sistemas síncronos y la suposición de su caída en los asíncronos.
- Revisión de información. El empleo de información redundante, como los códigos de detección de errores, es un ejemplo de esta clase de revisión.
- Revisión de estado. Los mecanismos de revisión de tiempo de ejecución, que utiliza un componente para revisar su estado interno. Es un ejemplo.
- Revisión de elemento. La invocación de procedimientos de prueba, con salidas bien definidas para ciertas entradas, permiten corroborar la presencia de fallas en un componente.

Ubicación de la falla

La ubicación de errores es el proceso de determinar dónde se encuentran las fallas, así como su alcance.

Confinamiento de la falla

El confinamiento de la falla se refiere al proceso de aislar una falla y evitar su propagación a través del sistema. En este sentido, el confinamiento no implica una restauración de la falla, 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. Según [45], existen dos principales técnicas para conseguirlo, las cuales son: la recuperación con retroceso (backward recovery) y la recuperación hacia delante (forward recovery).

Recuperación de Fallas

El proceso de recuperación de fallas consiste en mantener o alcanzar de nuevo un estado operacional del sistema a pesar de la presencia de fallas.

Hasta antes de esta fase, la falla ha sido detectada, ubicadas sus fronteras y confinada. Sin embargo, la fuente de la misma 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.

La recuperación de un sistema se refiere concretamente a las acciones correctivas sobre el componente defectuoso. De manera concreta, dichas acciones pueden significar:

- La recuperación de la falla en el componente,
- Sustitución del mismo,
- El uso del componente de manera indistinta o
- Su eliminación por completo del sistema

2.3.2 Transacciones

Una transacción es una abstracción de alto nivel que permite que los programadores se concentren en los algoritmos y la forma en que los procesos trabajan juntos o en paralelo, y no en los detalles de bajo nivel como son los semáforos y manejo de regiones críticas.

Una transacción se define como un conjunto de operaciones que deben de cumplir las siguientes características:

- Atomicidad.- se ejecutan todas las operaciones o ninguna.
- Consistencia.- las transacciones no violan las invariantes del sistema.
- Aislamiento.- las transacciones concurrentes no interfieren entre sí.
- Durabilidad.- Una vez comprometida la transacción, los cambios son permanentes.

Una transacción inicia cuando un proceso anuncia que desea comenzar una transacción con uno o más procesos. Después de terminar de hacer todas las operaciones un proceso iniciador anuncia que desea que todos los demás se comprometan con el trabajo efectuado. Si todos se comprometen, entonces los resultados son permanentes. Si uno o más se niegan o fallan antes de expresar su acuerdo, entonces la situación regresa al estado que presentaba antes de comenzar la transacción, sin que existan efectos colaterales en los objetos, archivos, bases de datos, etc. Que fueron utilizados en la transacción.

Atomicidad en un ambiente distribuido

El aseguramiento de la atomicidad en un ambiente distribuido implica que los procesos involucrados en una transacción deben todos ponerse de acuerdo en si deben de anular o comprometer la transacción. Entonces, el problema de garantizar la atomicidad se reduce a alcanzar un consenso en las decisiones de todos los participantes en una transacción.

Protocolos de compromisos (Versión datos replicados)

Cuando trabajamos con **réplicas** (ver sección 2.3.3) se inyecta al sistema un nuevo problema que es el de la consistencia de los datos replicados. Esto es, que dos diferentes réplicas de un mismo dato tienen que ser iguales[1] cuando estén disponibles a los distintos clientes. Para solucionar los anteriores problemas se han diseñado protocolos (versión sistemas distribuidos y datos replicados) para garantizar bajo ciertas circunstancias la consistencia de las réplicas:

- Protocolo de Compromiso de Dos Fases (P2F)
- Protocolo de Compromiso de Tres Fases (P3F)

[1] Las réplicas de un mismo dato no tienen por que ser iguales en todo momento, sólo cuando estén disponibles a los clientes

La propiedad de atomicidad en una transacción se puede asegurar a través de un protocolo de compromiso que utiliza dos fases o tres fases. Existe otro protocolo menos usado, pero que evita la posibilidad de bloqueo.

Para poder implementar el protocolo de compromiso se sugiere utilizar la siguiente arquitectura: En cada sitio debe de haber un gestor de transacciones, un coordinador de transacciones, además de una bitácora, [1].

- El gestor de transacciones es el encargado de ejecutar las transacciones locales.
- El coordinador de transacciones, es el encargado de coordinar las transacciones distribuidas inicializadas en ese sitio.
- La bitácora, es un instrumento utilizado para la recuperación del estado consistente en un sistema distribuido después de que algún sitio regresa de la falla en ellas se almacenan sólo operaciones indempotentes de una transacción.

Bitácora de escritura anticipada

Una forma de implementar las transacciones es la de bitácora de escritura anticipada, también conocida como lista de intenciones [1]. Una **bitácora** es una área de almacenamiento estable asignada para guardar una secuencia de registros en los cuales se lleva una historia de todas las actividades de actualización de las variables. Con este método las variables se modifican, pero antes de cambiar cualquier valor, se escribe un registro en la bitácora que contiene la variable, bloque a modificar, los valores anteriores y/o nuevos de la variable. Sólo después de que se escribe en la bitácora se harán los cambios.

La bitácora es necesaria para la recuperación de fallas. Supongamos que no se implementa la bitácora y todos los cambios se efectúan directamente sobre las variables, si el sistema falla a la mitad de una transacción, cuando se recupere de la falla se tendría que ejecutar las operaciones que faltaron de la transacción o se tendrían que deshacer las operaciones que se ejecutaron para asegurar la propiedad de atomicidad. Pero si no se cuenta con la bitácora, el sistema no sabría cuales operaciones faltan por hacer o cuales debe de deshacer, y peor aún no conoce los valores anteriores de las variables que son afectadas.

2.3.3 Replicación

Para poder construir sistemas tolerantes a fallas, es frecuente hacer uso de replicación [12], en tendiendo por replicación la existencia simultanea de varias instancias de un servicio. Esta característica es necesaria pero no suficiente, esto quiere decir, que un sistema tolerante a fallas necesariamente usa *réplicas* de información o de recursos, pero que un sistema que usa réplicas de información o de recursos, no necesariamente es tolerante a fallas.

En un sistema distribuido, existen servicios o datos que son más utilizados que otros, y por tal motivo, se vuelven un cuello de botella para sus clientes, obligándolos a esperar su turno. Mientras que si el servicio o dato se réplica en otro sitio, el tiempo de espera para los clientes solicitantes disminuiría aumentando así la disponibilidad.

Por otro lado, cuando un cliente hace una petición de un servicio o datos al sistema, esta petición no se hará a un sólo servidor sino que se dirige a un grupo de ellos que tengan replicados los

recursos necesarios para llevarla a cabo. Esto permite, bajo ciertas circunstancias, garantizar la correcta respuesta a la petición en caso de que uno o más de los servidores fallen.

En resumen, el uso de múltiples ejemplares de recursos además de incrementar la velocidad de ejecución y aumentar la disponibilidad del recurso y los datos en los sistemas de cómputo, también sirve para implementar tolerancia a fallas en los sistemas.

El aspecto más importante en el **manejo simultáneo de múltiples réplicas** (ver sección 2.3.4) es el criterio de *linealidad* [33]. Este criterio exige que el efecto resultante de los accesos de un número de clientes sobre un conjunto de elementos replicados sea el mismo a que si estos se realizaran de manera secuencial y sobre un solo elemento. La importancia de este criterio radica en que su cumplimiento preserva la semántica de cualquier programa que no tome en cuenta la replicación de los elementos.

Existen tres principales técnicas para hacer uso de réplicas, la diferencia entre ellas es debido a la forma y el momento en que se manejan los mensajes entre ellas y su cliente, estas técnicas son:

- Réplicas Activas
- Respaldo Primario
- Réplicas Perezosas

El usar una u otra de las anteriores técnicas esta en consideración de las necesidades del sistema, además de requerirse de una adecuada utilización de la técnica por parte del implementador para no entrar en conflictos con las características deseables de un sistema distribuido, como podría ser la transparencia de la distribución.

Réplicas activas

La replicación activa es una técnica muy conocida de redundancia, donde los clientes no contactan a un solo servidor (en un sitio en particular), sino a un grupo de ellos (de preferencia se encuentran en distintos sitios) en los cuales se alojan las réplicas.

Los pasos que se siguen para usar réplicas activas (ver fig. 2.6), son:

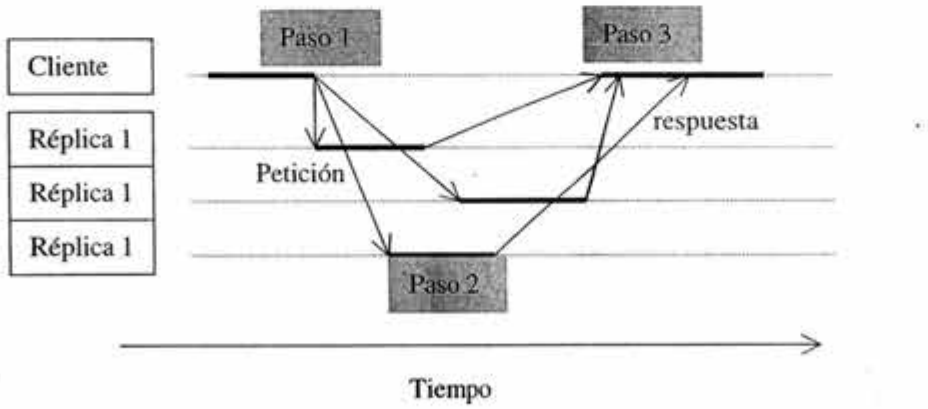


Fig. 2.6 Réplicas activas

1. El cliente debe de mandar un mensaje a todas réplicas.
2. Cada réplica procesa su petición, actualiza su estado y responde al cliente.
3. El cliente espera a recibir la primera respuesta, o espera a tener todas las respuestas y toma como verdadera a la que tenga mayor número de apariciones (sí es que existieran respuestas diferentes).

Este tipo de replicación requiere de una técnica de comunicación que garantice la llegada de los mensajes a las distintas réplicas en el mismo orden. Así al final de la ejecución los servidores que tengan las réplicas se encontrarán en el mismo estado y podrán dar la misma respuesta. Esta tarea queda a cargo de las capas de comunicación de grupos quienes dan garantías de orden usando relojes lógicos entre otras técnicas [41] y [22].

En este tipo de replicación si un servidor falla, las réplicas entregarán el resultado al cliente evitando que tenga que detenerse el sistema. Cuando la réplica que tuvo la falla se reactiva, sólo se actualizará con las demás réplicas volviendo a estar lista.

Respaldo primario (réplica pasiva)

En el respaldo primario, también conocido como réplica pasiva, el cliente envía su solicitud a una sola réplica llamada *réplica primaria* (las otras se denominan réplicas secundarias) la cual se encarga de:

1. Ejecutar la solicitud.
2. Enviar mensajes a las réplicas secundarias con los resultados para que se actualicen sin que tengan que ejecutar la solicitud nuevamente.
3. Una vez que todas las réplicas se han actualizado, envía la respuesta al cliente.

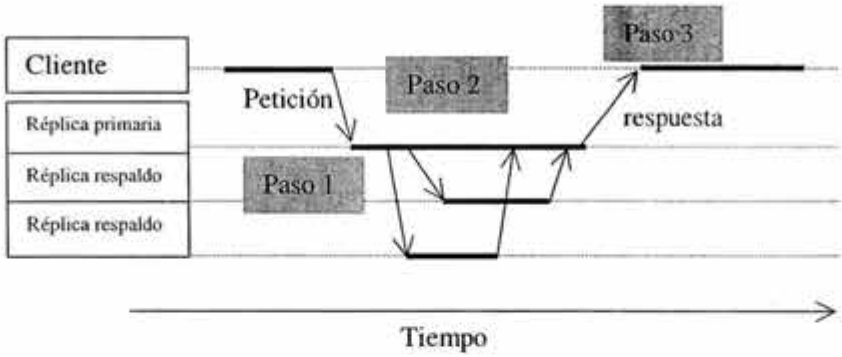


Fig. 2.7 Réplica primaria

En este caso, cuando la réplica primaria falla, una de las secundarias toma su lugar inmediatamente ya que están actualizadas. La principal ventaja que tiene esta implementación de tolerancia a fallas a diferencia de la replicación activa se encuentra es una mayor sencillez durante la operación normal, ya que solo un elemento ejecuta la operación y las demás sólo se actualizan evitando así el uso de complejos procedimientos para asegurar la llegada en orden de los mensajes a las diferentes réplicas.

Replicación perezosa

Esta técnica de replicación es similar a la de respaldo primario, ya que sólo una réplica hace todo el trabajo y las secundarias sólo se actualizan. La diferencia entre las dos técnicas de replicación, se encuentra en el momento en que se actualizan las réplicas secundarias. Mientras que en la réplica primaria se actualizan las secundarias antes de que se entregue la respuesta al cliente, en la replicación perezosa, las réplicas secundarias se actualizan después de que se entrega el resultado al cliente [24].

Aquí la implementación de tolerancia a fallas requiere de más trabajo por parte del administrador de réplicas, ya que si falla la réplica primaria antes de enviar el mensaje de actualización a las demás réplicas, éstas no podrán tomar su lugar ya que existe inconsistencia entre todas las réplicas.

2.3.4 Grupos de comunicación

Como vimos en la sección 2.3.3 presenta las técnicas de uso de réplicas, la comunicación que hay entre sitios que contienen réplicas es muy importante. Además, en algunos casos todas las réplicas deben de recibir todos los mensajes concernientes al servicio replicado y no es necesario que aquellos sitios que no contienen a la réplica se enteren de esos mensajes. Esto nos lleva a la definición de grupo. En este contexto, un *grupo* es una colección de sitios que actúan conjuntamente para ofrecer un servicio, y requieren que cuando se envíe un mensaje al grupo, todos sus miembros lo reciban. Un grupo permite a los clientes de un servicio replicado dirigirse sólo a una entidad. Así que el cliente envía un mensaje a un grupo de sitios sin tener que conocer el número de sus integrantes y/o su localización.

Existen dos tipos fundamentales de grupos: grupos dinámicos y grupos estáticos:

- Los grupos estáticos mantienen intacta su composición durante el tiempo de vida del sistema. Es decir, sus miembros no pueden cambiar aún cuando haya fallado.
- Los grupos dinámicos pueden cambiar a sus miembros durante el tiempo de vida del sistema, esto es, la vista de los miembros de un grupo dinámico está en función del tiempo. En este caso, para llevar un registro histórico de los miembros de un grupo es necesario observar los momentos en que un elemento se da de baja o alta en el grupo y guardarlo en almacenamiento estable.

2.3.5 Enmascaramiento (o transparencia) de fallas

Enmascarar una falla significa no permitir que sus efectos sean percibidos por el usuario del sistema. El enmascaramiento de fallas es una de las técnicas más usadas para implementar la tolerancia a fallas. Existen dos enfoques fundamentales para enmascararlas:

- **Enmascaramiento jerárquico de fallas**. Esto ocurre cuando existe una relación de jerarquía entre los servicios del sistema, lo que implica que una falla en un nivel pueda ocasionar una falla en un nivel de mayor jerarquía. Cuando se diseña un sistema que maneje enmascaramiento jerárquico, lo más conveniente es utilizar el manejo de excepciones para propagar la noticia de una falla entre los diferentes niveles. El manejo de excepciones puede definirse de acuerdo a diferentes modelos, uno de los ejemplos más representativos se describe en [42].
- **Enmascaramiento de fallas utilizando grupos**. En este caso el enmascaramiento se realiza utilizando sitios redundantes y físicamente independientes, los cuales se agrupan

lógicamente. Se pueden distinguir principalmente dos tipos de redundancia –activa y pasiva- según la forma en que los miembros de cada sitio se coordinen para enmascarar las fallas (véase el ver sección 2.3.3). Los mecanismos específicos para coordinar y manejar a los miembros de un grupo de réplicas dependen, en gran medida, de la semántica de fallas del grupo y los servicios de comunicación utilizados.

2.4 Discusión

Actualmente es posible identificar algunas subcategorías dentro de los middlewares en función del paradigma que manejan las aplicaciones que los utilizan. Por ejemplo, los middlewares que han sido diseñados basándose en las abstracciones cliente/servidor y RPC, que son los más utilizados hoy en día, los podemos dividir en dos subcategorías: middleware para objetos distribuidos y middleware para componentes.

Dentro de los middleware para objetos distribuidos podemos encontrar las siguientes plataformas:

- COM/DCOM de Microsoft
- CORBA y CORBATaF de la OMG.

Y dentro de los middleware para componentes podemos encontrar las siguientes plataformas:

- CCM de la OMG
- EJB de SUN
- Séneca, que es un proyecto académico, ver [49]

Pero también es bien sabido por todos, que las necesidades de los usuarios son diversas, según el tipo de sistema que se utiliza y para qué se utiliza. En este sentido, existen usuarios como los hospitales y las casas de bolsa que requieren que sus sistemas no se detengan en momento críticos, ya que de hacerlo se pueden perder en un caso vidas y en otros millones de dólares. Es por eso que los middlewares que ofrecen los servicios de tolerancia a fallas son importantes en este tipo de escenario, por tal razón, este tipo de tolerancia es buscada y estudiada en los capítulos siguientes como principal objetivo en este estudio.

3 MIDDLEWARES: En busca de T. a F.

Con el objetivo de descargar al desarrollador de aplicaciones distribuidas de la programación de la tolerancia a fallas y dejar esta tarea a cargo del middleware. En esta sección hacemos una revisión, en busca de mecanismos de tolerancia a fallas, de las especificaciones de middlewares de mayor presencia en el mercado de software enfocándonos en aquéllas en el que su diseño esta basado en el modelo cliente/servidor y RCP (ver sección 2.1.2). Además, haremos una revisión al proyecto académico Séneca que también posee estas características.

Los middlewares estudiados son: DICOM de Microsoft, EJB de SUN, CORBA, CCM CORBATaF de la OMG, y Séneca [49], De los cuales, en el resto del capítulo se dará un panorama de su arquitectura y de los servicios que prestan a los componentes u objetos distribuidos según el paradigma de programación que soportan.

3.1 COM y DCOM

El Modelo Objeto Componente (COM) [4] de Microsoft soporta la interacción entre un cliente y un servidor de objetos que podrían residir en la misma dirección de espacio o en diferentes procesos en el mismo sitio, esta interacción es definida tal que la conexión entre los componentes es transparente para el programador

Por otra parte, el Modelo de Objetos Componentes Distribuidos (DCOM) de Microsoft [8], es la extensión que permite habilitar de forma remota un COM. El DCOM es un protocolo de alto nivel de la red y este es usado para conectar un Cliente COM con un Servidor remoto COM. DCOM esconde completamente la localización de los componentes a los clientes.

3.1.1 Tipo de modelo

DCOM sigue un modelo a objetos distribuidos, el cual describe la creación, identidad, búsqueda y operación de un objeto en un ambiente distribuido

3.1.2 Descripción de un objeto

El término objeto en COM significa la instancia de un objeto. Un objeto COM es también llamado componente, Un objeto COM es identificado por un CLSID, a través del cual el cliente puede acceder a él. El objeto físicamente reside en cualquier parte de la red en un servidor que permite sólo ver sus interfaces.

Dado que la especificación de un COM se da a nivel binario, esto permite que los objetos COM/DCOM puedan ser escritos en diversos lenguajes de programación tales como C++, JAVA, y Visual Basic. En DCOM las operaciones de objetos funcionan sobre plataformas que pueden interactuar con Microsoft Windows.

3.1.3 Arquitectura

A diferencia de COM, al usar DCOM no sólo se deben de cruzar los límites de los procesos, sino también se deben de enviar mensajes a través de la red que conectan diferentes computadoras.

Para su estudio DCOM es separado en tres niveles; alto, medio y bajo.

Nivel Alto

Este nivel corresponde a la vista que tiene el programador de la arquitectura de DCOM, en donde la distribución es completamente transparente.

Lado del Cliente. Para crear un objeto remoto, un cliente debe de conocer el identificador de clase (CLSID) del objeto. El CLSID es un identificador único universal (UUID). Cuando un cliente llama la función de la biblioteca COM (COM library) y pide la creación de un objeto, el CLSID es un parámetro de entrada que dice al COM qué tipo de objeto será instanciado. La biblioteca COM busca en el *Registro de Windows* (Registro) y encuentra información acerca de su CLSID, entonces COM pide instanciar el objeto. Cuando un objeto es creado, se crea con él su referencia llamada *apuntador de interfaz* (interface pointer), que es regresada al cliente. Usando este apuntador, el cliente puede llamar a los métodos descritos en la interfaz del objeto la cual se obtiene a través del *Lenguaje de definición de objetos* (IDL).

Lado del Servidor. La información acerca del objeto debe de ser agregada tanto en el Registro del cliente como en el Registro del servidor. Cuando el servidor recibe la petición de crear un objeto, este busca a través del CLSID su correspondiente *clase creadora* (class factory), la cual tiene la habilidad de crear los objetos de ese tipo. El objeto es instanciado en el servidor y su referencia es enviada al cliente.

Nivel medio

El nivel medio es transparente al cliente y al servidor y da la ilusión a estos de que residen en el mismo proceso. Para el intercambio de información entre el cliente y el servidor, los datos son empaquetados para ser enviados sobre la red. Así, el que envía debe de hacerlo en un formato correcto de tal forma que cuando llegue a su destino este paquete sea desempaquetado e interpretado correctamente por el que recibe. El empaquetado y desempaquetado es provisto por COM a través de RCP, ver sección 2.1.2, y es transparente para el programador, ver fig. 3-1.

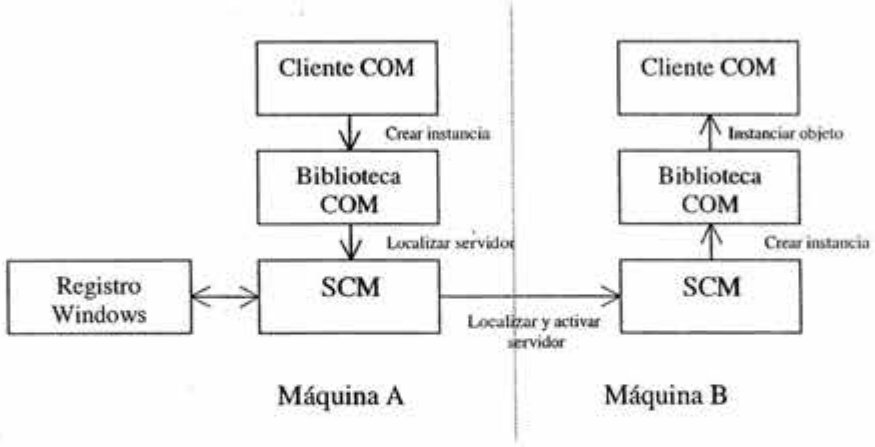


Fig. 3-1 Registro de windows y SCM

Lado del Cliente. Cuando el cliente pide a la biblioteca COM la creación de un objeto, COM llama al administrador de control de servicios (SCM). El SCM consulta el Registro para obtener el nombre en la red del servidor, pide al servidor crear el objeto y regresar un apuntador de interfaz. Esta interfaz es entregada al cliente. Sin embargo, este apuntador no apunta directamente al componente, pero sí a un objeto sustituto que se encuentra en el sitio del cliente. El objeto sustituyo es llamado proxy.

Lado del Servidor. Cuando del lado del servidor el SCM es requerido este llama a su clase creadora y recibe la referencia (apuntador de interfaz) del nuevo objeto creado. Entonces, el servidor instancia un stub, quien es un sustituto del cliente en el servidor. El stub empaqueta la referencia y la envía al cliente.

Nivel bajo

El nivel bajo consiste en el protocolo de comunicación que se usa para empaquetar mensajes entre el stub y skeleton (llamados proxy y stub en esta especificación).

DCOM es considerado un protocolo de red de alto nivel ya que este está sobre varios niveles de otros protocolos. El protocolo de conexión DCOM es basado en el Ambiente de cómputo distribuido (DCE) de la Fundación de software abierto (OSF) y RPC. DCOM realmente no es un protocolo independiente en el nivel alto de RPC, pero tiene una extensión del nivel RPC. Sin embargo, el protocolo de red de DCOM es frecuentemente llamado RPC Objeto, o ORPC, para mostrar la relación con RPC. DCOM usa el empaquetado de especificación de tipos de datos simples para el OSF DCE RPC, pero este estándar puede ser extendido para soportar empaquetado de apuntadores de interfaz.

3.1.4 Separación entre interfaces e implementación

Un cliente nunca accede directamente a un objeto COM, y sólo puede acceder a él a través de sus interfaces.

Las interfaces de objeto COM son definidas a través del Lenguaje de definición de interfaces (IDL), con el cual se crea un archivo IDL que al ser compilado crea una biblioteca, que describe al objeto y sus interfaces.

3.1.5 Servicios

Los servicios entregados por DCOM a los programadores de objetos es básicamente la transparencia de la red. Ya que los servicios de seguridad, transacciones, ciclo de vida, etc. son provistos por el programador, o por objetos COM que son creados para proveer esos servicios a otros objetos COM.

3.1.6 Tolerancia a fallas

La tolerancia a fallas, aún siendo un servicio, lo veremos como un tema aparte, ya que es el principal tema de este estudio.

DCOM no ofrece el servicio de tolerancia a fallas. Sin embargo, ofrece algunas herramientas que podrían ser usadas para dicho propósito. Esas herramientas son: el servicio de *pings* y periodos de

tiempo predeterminados (time out). Los tiempos predeterminados se utilizan para verificar si es accesible un servidor y si la máquina servidora está activa en la red, mientras que los pings son utilizados cuando el cliente ya tiene un servidor y quiere saber si este está funcionando y puede responder, estos mecanismos no están al alcance de los programadores ya que están previamente establecidos.

Es claro que con sólo los pings y los tiempos predeterminados, la programación de tolerancia a fallas es muy limitada en DCOM.

La tabla 3-1 resume las características generales de DCOM.

DCOM	
Propietario	Microsoft
Tipo de middleware	A objeto distribuidos
Lenguaje	Multilenguaje
Plataforma	Multiplataformas
Complejidad para construir grandes aplicaciones	Alta
Cambiar, agregar, o eliminar servicios en el middleware	Es permitido y sólo para aquellos servicios que tienen una interfaz bien definida.
Forma en que los objetos o componentes obtienen los servicios	A través de APIs (excepto la transparencia de distribución)
Tolerancia a fallas	No contempla tolerancia a fallas. Sin embargo, tiene mecanismos como: los ping y timeout. Estos mecanismos podrían ayudar al programador a implementar este servicio.

Tabla 3-1 DCOM

3.2 Enterprise JavaBeans (EJB)

El gran auge de redes como intranet e internet y su necesidad de clientes más ligeros han propiciado la separación de los servicios que son inherentes al sistema (como son; Sistemas de comunicación, de seguridad, etc.) de la parte funcional o lógica de negocios (como son; abonar en cuenta, dar de alta cliente, etc.), dando paso al cambio de modelo de aplicación cliente-servidor de dos capas al modelo de aplicación de multi-capas como EJB.

3.2.1 Tipo de modelo

La tecnología Enterprise Java Beans (EJB) [10] define un modelo de desarrollo y publicación de servidores de componentes Java (JavaBeans). EJB extiende el modelo de componentes Java para soportar servidores de componentes. Un servidor de componentes es una pieza de software pre-empaquetada que se ejecuta en una aplicación servidor. El modelo EJB permite al desarrollador de componentes beans concentrase en la parte funcional sin distraerse en los detalles del soporte del sistema.

3.2.2 Descripción de un componente

Los componentes Enterprise JavaBeans (Beans), son componentes de software que representa un concepto de negocio que se ejecutan en un entorno llamado "contenedor", el cual los aísla del mundo exterior, ver sección 3.2.3. Los beans son accedidos por las aplicaciones clientes haciendo uso de las interfaces EJBHome y EJBObject.

Tipos de beans

La especificación de EJB marca dos tipos de básicos de beans: los beans de entidad, que representan datos en una base de datos, y los beans de sesión que representan objetos o agentes que representan tareas. Cuando se construyen aplicaciones EJB se crean muchos conceptos diferentes del negocio, y cada concepto del negocio será manifestado como un bean de entidad o de sesión.

Un **Bean de sesión** es un objeto no persistente que implementa la parte funcional que se ejecuta del lado del servidor. El tiempo de vida del bean de sesión es limitado por el cliente, pero su ciclo de vida es administrado por el contenedor. Típicamente, un bean no sobrevive a la falla del sistema.

Los beans de sesión se subdividen en: beans de sesión con estado y beans de sesión sin estado.

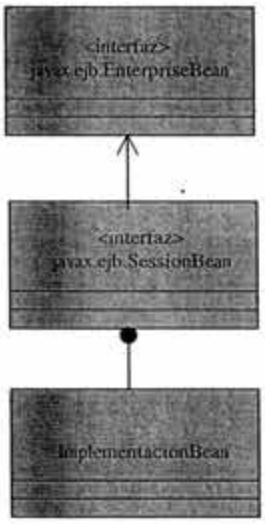


Fig. 3-2 Diagrama de clases para implementar el bean de sesión

- **Bean de sesión con estado.** El estado es necesario para que un bean de sesión que ejecuta trabajo para un cliente que hace llamadas a múltiples métodos mantenga un estado-convencional entre llamadas a métodos del bean. Para esto, el contenedor le asigna una identidad única a la hora de ser creado.

- **Bean de sesión sin estado.** Este tipo de bean no tiene un identificador único, y por lo tanto no puede ser "pooleado" y reusado repetidamente. Cada llamada a un método de negocio es independiente de las llamadas anteriores, esto es, los beans de sesión sin estado no recuerdan nada entre una llamada a método y la siguiente.

Por otro lado, el **bean de entidad** es usado para representar datos en una base de datos y proporciona una interfaz orientada a objetos a esos datos que normalmente serían accedidos mediante el JDBC u otro API. Además, los beans de entidad proporcionan un modelo de componente que permite a los desarrolladores de beans enfocar su atención en la parte funcional del bean, mientras el contenedor tiene la tarea de manejar la persistencia, la transacción y el control de acceso.

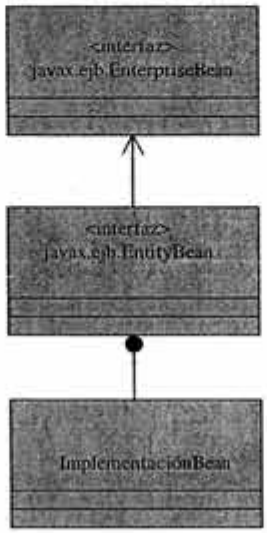


Fig. 3-3 Diagrama de clases para implementar el bean de entidad

Existen dos tipos de beans de entidad, esta subdivisión se hace según se señale al responsable del manejo de la persistencia: Persistencia manejada por el contenedor y persistencia manejada por el bean.

- **Persistencia manejada por el contenedor.** En este caso, el contenedor maneja la persistencia del bean identidad. Las herramientas de los vendedores se usan para manejar los campos de entidad a la base de datos y no se escribe ninguna línea de código de acceso a las bases de datos en la clase bean.

- **Persistencia manejada por el bean**. En este caso, el bean maneja la sincronización de su estado directamente en la base de datos, para ello debe de usar el API de conexión a la base de datos para leer y actualizar sus campos en la base de datos, pero el contenedor le dice cuándo hacer cada operación de sincronización y maneja las transacciones por el bean de manera automática.

3.2.3 Arquitectura EJB

La siguiente figura muestra la arquitectura básica de EJB

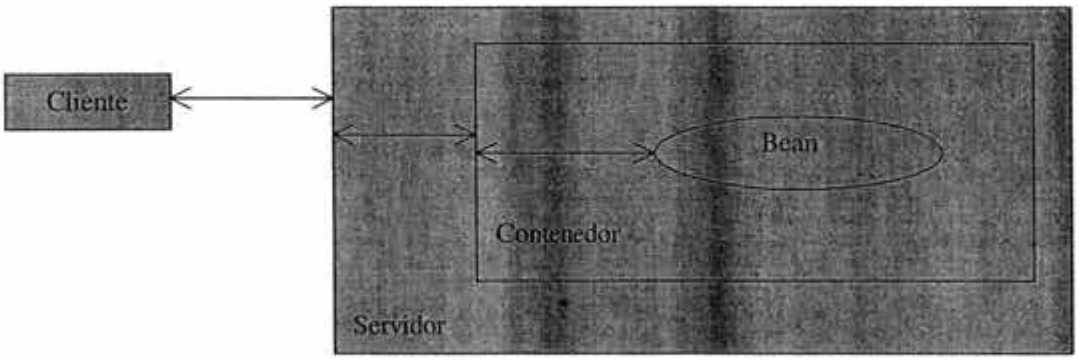


Fig. 3-4 Arquitectura EJB

La interacción entre los distintos elementos de la especificación se analiza a continuación.

Contenedor EJB

El contenedor EJB conocido como "contenedor", actúa como un intermediario entre un Enterprise bean y el mundo externo. Un bean no puede funcionar fuera de contenedor. El contenedor controla cada aspecto del bean en tiempo de ejecución incluyendo ciclo de vida, accesos remotos, concurrencia, seguridad, persistencia, transacciones y manejo de estados.

La tarea del contenedor es aislar al bean del acceso directo por parte de las aplicaciones cliente. Por lo tanto, el desarrollador de beans sólo debe de concentrase en encapsular la parte funcional, mientras que el contenedor intercepta, a través de las interfaces del bean, las invocaciones a los métodos que hacen las aplicaciones cliente con el fin de asegurar que los servicio middleware sean proporcionados automáticamente.

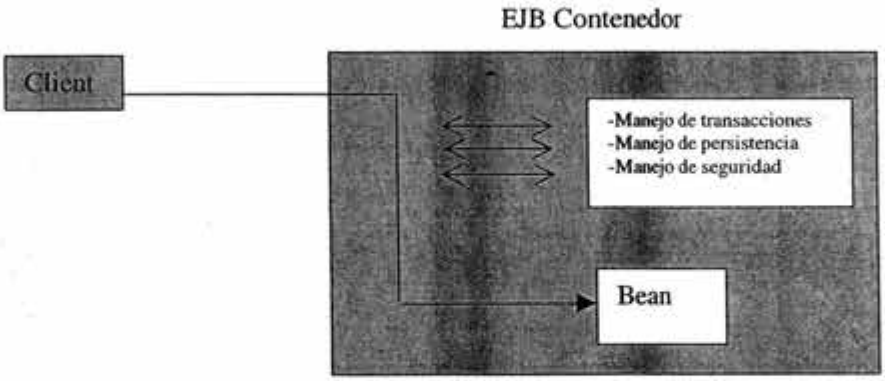


Fig. 3-5 Manejo automático de los servicios middleware

Los contenedores pueden manejar muchos beans simultáneamente, por lo mismo los recursos podrían ser saturados rápidamente. Para reducir el consumo de memoria y de procesador, el contenedor puede almacenar de manera temporal los beans y recuperarlos, de una manera transparente para el cliente, cuando sean requeridos.

Una de las principales características que tienen los EJB es la portabilidad. La portabilidad asegura que un bean pueda ejecutarse de manera correcta en cualquier contenedor sin necesidad de hacer cambios o recompilar su código, esto es una herramienta muy poderosa tanto para desarrolladores como consumidores de beans. Esta portabilidad se asegura a través de contratos entre el contenedor y el bean.

Servidor EJB

El servidor de EJB provee de un ambiente de ejecución para uno o más contenedores EJB. El servidor de EJB provee al sistema de los servicios de balanceo de carga, acceso a dispositivos y multiprocesamiento.

La interfaz entre el servidor y el contenedor no está definida en la especificación EJB, por lo cual en la mayoría de los casos, el proveedor del servidor es el mismo que el del contenedor.

3.2.4 Separación entre interfaces e implementación

Las interfaces son el único medio que pueden utilizar los clientes para poder invocar los métodos de los beans (componentes). Las interfaces de un bean son definidas a través del Lenguaje de definición de interfaces (IDL). Las interfaces están separadas de su implementación, que a diferencia de otros middlewares, sólo puede ser escrita en lenguaje java.

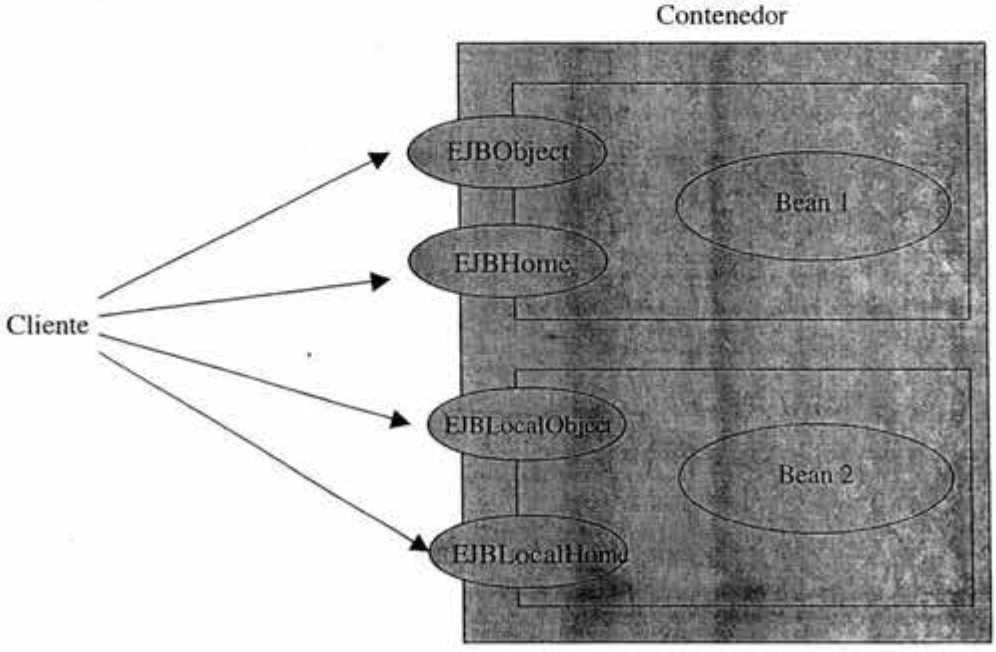


Fig. 3-6 Vista conceptual de la arquitectura EJB

Tipos de interfaces del bean

Existen cuatro tipos de interfaces especificadas en EJB, las cuales son:

- **La interfaz EJBObject** (remota) define los métodos de acceso, modificación, lectura y actualización de información sobre un concepto de negocio. Cada vez que un cliente invoca un método en un EJBObject, la petición pasa primero a través del contenedor quien lo entrega al bean. El desarrollador del bean define la interfaz usando Java Remota Method Invocation (Java RMI) y herramientas para que el contenedor genere la implementación del EJBObject. El paso de parámetros en la invocación de los métodos en el EJBObject es hecho por valor.

- **La interfaz EJBLocalObject** (local) es idéntica que una interfaz EJBObject sólo que la primera no utiliza Java RMI ya que para utilizar esta interfaz el cliente está obligado a estar ubicado en la misma máquina virtual que el bean, lo cual no es cierto para EJBObject. El paso de parámetros en la invocación de los métodos en el EJBLocalObject es hecho por referencia.

- **La interfaz EJBHome** (remota) representa la vista y el único medio de comunicación con el que cuenta un cliente para hacer uso de un bean. Esta interfaz define métodos de creación, borrado y localización del bean. Cada vez que un cliente invoca un método en un EJBObject, la petición pasa primero a través del contenedor quien lo entrega al bean. El desarrollador del bean define la interfaz usando Java RMI y herramientas para que el contenedor genere la implementación del EJBObject. El paso de parámetros en la invocación de los métodos en el EJBHome es hecho por valor.

- **La interfaz EJBLocalHome** (local) es idéntica que una interfaz EJBHome sólo que la primera no utiliza Java RMI ya que para utilizar esta interfaz el cliente está obligado a estar ubicado en la misma máquina virtual que el bean, lo cual no es cierto para EJBHome. El paso de parámetros en la invocación de los métodos en el EJBLocalHome es hecho por referencia.

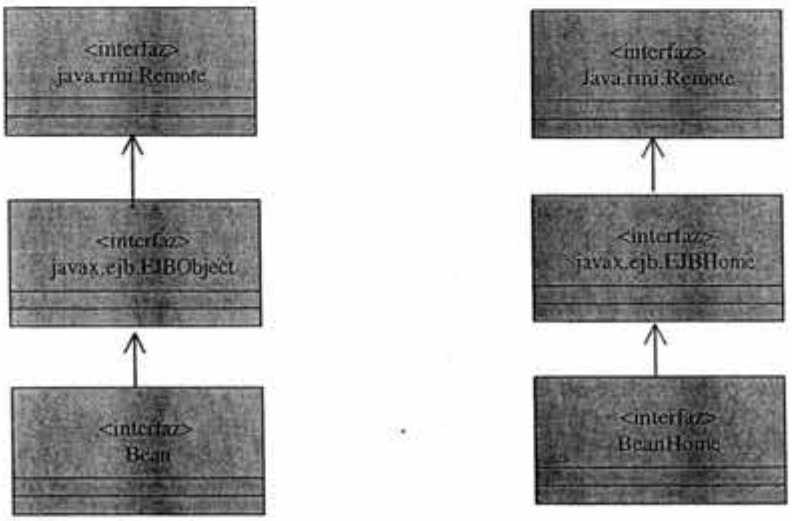


Fig. 3-7 Diagrama de clases para las interfaces

3.2.5 Servicios

En EJB los servicios especificados son: ciclo de vida, accesos remotos, concurrencia, seguridad, persistencia, transacciones y manejo de estados. Además, en EJB se define la forma en que se soportará la distribución de los beans y provee una especificación de un descriptor de desplegado (Deployment descriptor) que contiene la información declarativa del bean (esto es, la información que no es incluida directamente en el código del bean) que es de importancia para el contenedor y que también es necesaria para ensamblar una aplicación.

Pero quizás uno de los servicios más relevantes en EJB es el soporte para transacciones. De esta forma el proveedor del bean y el programador de la aplicación cliente se pueden olvidar de los complejos detalles de las transacciones, pero dejando (si ellos lo necesitan) que estos participen (de alguna manera) en ellas.

	EJB
Propietario	SUN
Tipo de middleware	A componentes
Lenguaje	Java
Plataforma	Multiplataformas y heterogéneos
Complejidad para construir grandes aplicaciones	Media
Cambiar, agregar o eliminar servicios en el middleware	No permitido
Forma en que los objetos o componentes obtienen los servios del middleware	De forma automática y en algunos casos a petición del desarrollador
Tolerancia a fallas	No contempla tolerancia a fallas y sólo cuenta con excepciones para evitar la propagación de las mismas.

Tabla 3-2 EJB

La arquitectura Enterprise JavaBeans soporta sólo transacciones planas, esto es, las transacciones no pueden tener transacciones anidadas. El proveedor del bean puede escoger entre utilizar marcas de transacciones, que señalan su inicio y su fin, en el código del bean (bean-managed transaction demarcation) y que es ejecutado por el contenedor, o utilizar transacciones declarativas que se activan en la invocación de algún método en especifico, y que son ejecutadas automáticamente por

el contenedor (container-managed transaction demarcation), También, en algunos casos se podrá utilizar marcas de transacción en el código del cliente (client-maneged transaction demarcation).

Sólo los beans de sesión pueden ser implementados con *bean-managed transaction demarcation* o *container-managed transaction demarcation*, ya que los bean de entidad siempre serán implementados con *container-managed transaction demarcation.*

3.2.6 Tolerancia a fallas

En la actual especificación EJB no se cuenta con el servicio de tolerancia a fallas como tal, aunque cuenta con algunos mecanismos, tales como: manejo de transacciones y uso de excepciones para poder implementar dicho servicio, e incluso, EJB podría llegar a manejar réplicas gracias al modelo de desplegado, pero el desarrollador de la implementación debe de hacerse cargo de las políticas de consistencia así como de su programación.

La tabla 3-2 resume las características generales de EJB.

3.3 CORBA

CORBA es una especificación de un middleware a objetos distribuidos [44] que permite la interoperabilidad de sitio heterogéneos, esto es, CORBA permite que los sistemas distribuidos que contengan sitios con arquitectura de hardware, sistemas operativos, y lenguajes de implementación distintos puedan cooperar entre sí para la ejecución de una aplicación.

3.3.1 Tipo de modelo

CORBA sigue un modelo a objetos distribuidos, el cual describe la creación, identidad, búsqueda y operación de un objeto en un ambiente distribuido.

3.3.2 Descripción de un objeto

Un objeto CORBA es una entidad encapsulada e identificable (a través de su referencia de objeto) que vive en la red y que provee uno o más servicios que un cliente puede requerir a través de la invocación de sus métodos. Los objetos CORBA soportan herencia y polimorfismo.

3.3.3 Arquitectura

Al igual que DCOM, los detalles y funcionalidad de CORBA pueden ser estudiadas en tres diferentes niveles: alto, medio y bajo.

Nivel alto

Este nivel corresponde a la vista que tiene el programador de la arquitectura CORBA.

Lado del cliente Desde el punto de vista de un cliente. El cliente hace uso de un objeto CORBA (que se encuentra en el sitio del cliente o un sitio distinto) e invoca sus métodos de la misma forma como si ambos se encontraran en el mismo sitio. Para que un cliente pueda instanciar un objeto e invocar sus métodos primero requiere conocer su referencia, esta referencia en CORBA se conoce

como *referencia de objeto*, y al proceso para obtener esta referencia se le conoce como *reconocimiento* (binding). Un cliente interactúa con un objeto CORBA sólo a través de sus métodos descritos en un *Lenguaje de definición de interfaces* (IDL), el cual sirve como contrato entre el cliente y el servidor.

La petición de invocación que hace un cliente a un objeto CORBA no pasa directamente del cliente al objeto, sino a través del *Object request broker* (ORB). El ORB o "bus de objetos" sirve como un canal de comunicación entre el cliente y el servidor, el cual se encarga (del lado del cliente) de recibir la invocación y localizar el objeto apropiado.

Lado del servidor El servidor crea una instancia de un objeto y hace que ésta sea alcanzable al cliente registrándola en el ORB quien envía los parámetros a la instancia del objeto y pasa el control a este, además una vez terminada la invocación, envía el resultado y el control al cliente.

Nivel medio

La arquitectura de CORBA a nivel medio se muestra en la Fig. 3-9, que es necesaria para proveer al cliente y al servidor la ilusión de que ambos se encuentran en el mismo sitio.

Lado del cliente. Cuando el cliente inicia una petición, este recibe la interfaz del objeto que se encuentra en *repositorio de interfaces*, el cual provee un lugar seguro en memoria persistente para las definiciones de interfaces. Una vez que el cliente encuentra la interfaz del objeto, busca su implementación en el *repositorio de implementaciones*. Los repositorios de interfaces e implementaciones pueden ser accedidos directamente vía la *interfaz del ORB* e indirectamente a través de la invocación de los métodos vía la *interfaces de invocación estática* (SII) e *interfaces de invocación dinámica* (DII).

Lado del servidor. El ORB localiza el servidor apropiado donde se encuentra la implementación del objeto, transmite los parámetros y transfiere el control a través del *IDL skeleton* o la *interfaz dinámica skeleton* (DSI), estos skeletons son especificados tanto en las interfaces como en el *adaptador de objetos (OA)* quien se encarga de proveer un ambiente (en tiempo de ejecución) para instanciar servidores de objetos. Además el OA se encarga de obtener los servicios provistos por el ORB y entregarlos al objeto.

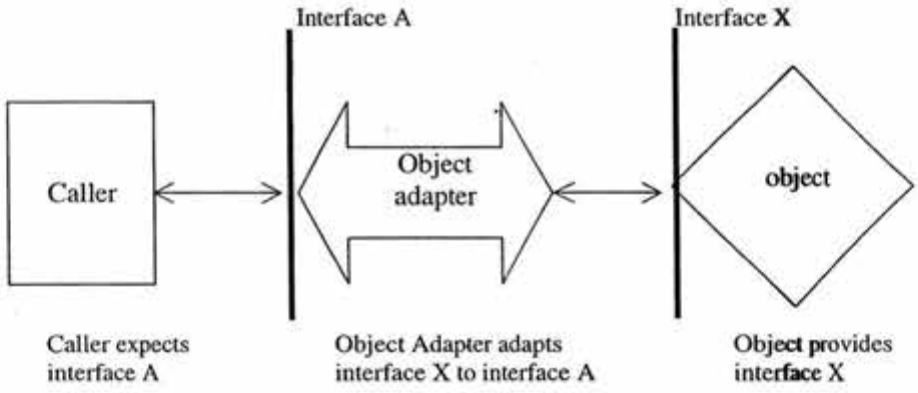


Fig. 3-8 Ubicación del objeto adaptador [37]

También del lado del servidor un elemento importante es el POA. Quien se encarga de conectar a la invocación de un cliente con su sirviente[2], el POA es parte de la implementación del objeto, ya que la implementación de un objeto es la combinación de un POA y un sirviente.

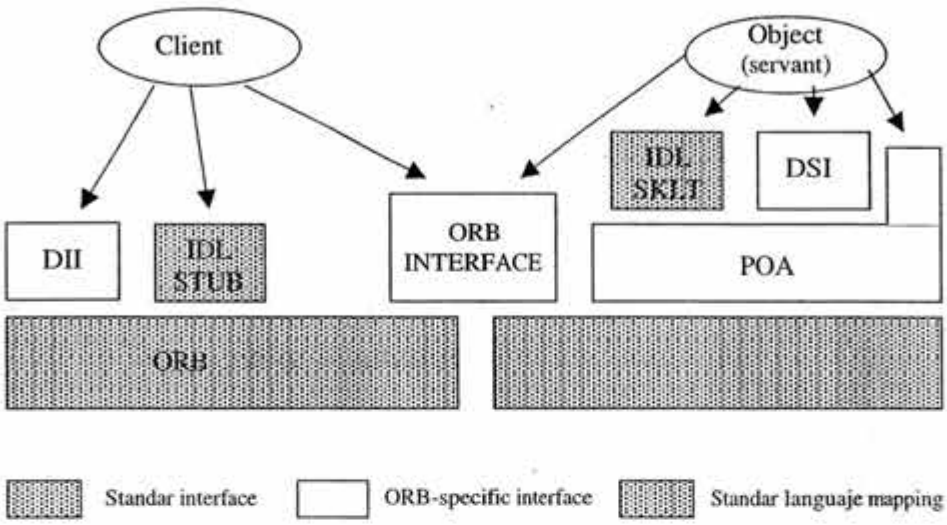


Fig. 3-9 Arquitectura CORBA [39]

Nivel bajo

El nivel bajo en CORBA se refiere al uso de protocolos para realizar la comunicación e interoperabilidad entre el cliente y el servidor.

En general la interoperabilidad de la arquitectura del ORB es basada en el protocolo general inter-ORB (GIOP), este protocolo especifica la sintaxis y un conjunto de formatos estándares de mensajes para realizar la interoperabilidad del ORB sobre una conexión orientada al transporte. GIOP consiste de tres especificaciones que son: Representación común de datos (CDR), GIOP formato de mensajes, GIOP transporte.

Además del GIOP, en CORBA se especifican los siguientes protocolos: protocolo *de Internet inter-ORB* (IIOP) que es un protocolo estándar para el manejo de Internet entre distintos ORB's y especifica como el GIOP debe ser construido sobre la capa de TCP/IP, y los *protocolos de ambiente específico inter-ORB* (ESIOPs), que permiten construir ORB para situaciones donde la infraestructura del cómputo distribuido está ya en uso.

[2] Un sirviente es el código escrito por el programador que contiene la lógica de negocios, pero sin ser por si mismo un objeto CORBA.

3.3.4 Separación entre interfaces e implementación

Las interfaces de un objeto CORBA son definidas a través del Lenguaje de definición de interfaces (IDL) donde las interfaces están separadas de su implementación, permitiendo así que las implementaciones puedan ser hechas en diferentes lenguajes (c, pascal, etc.) sin perder su interoperabilidad. El cliente sólo puede acceder al objeto a través de estas interfaces con ayuda del ORB.

CORBA	
Propietario	OMG
Tipo de middleware	A objeto distribuidos
Lenguaje	Multilenguajes
Plataforma	Multiplataformas
Complejidad para construir grandes aplicaciones	Alta
Cambiar, agregar, o eliminar servicios	Es permitido para los servicios CORBA
Forma en que los objetos o componentes obtienen los servicios	A través de APIs (Excepto la transparencia de distribución y la interoperabilidad los cuales son automáticos)
Tolerancia a fallas	No contempla tolerancia a fallas. Sin embargo, tiene mecanismos como las excepciones y el servicio de colección de objetos, que podrían ayudar al programador a implementar este servicio.

Tabla 3-3 CORBA

3.3.5 Servicios

El ORB de CORBA ofrece a los programadores de objetos los servicios de la red, como son: la localización de servidores de objetos, manejo de mensajes, y manejo de protocolos y ocultamiento de la red. Otros servicios son provistos por el programador, incluso hasta con otros objetos CORBA.

También la OMG provee de especificaciones de *servicios CORBA* (CORBAservices) pero sin ser parte del ORB. Entre los servicios que se ofrecen están [43]: el manejo de transacciones, la seguridad, el manejo de grupo de objetos, el manejo de eventos, el manejo de concurrencia, etc.

3.3.6 Tolerancia a fallas

En la especificación de CORBA no se contempla la tolerancia a fallas[1] como tal, pero sí contempla algunos mecanismos que podrían utilizarse para dicho propósito. Estos mecanismos son el manejo de excepciones dentro del lenguaje IDL, y aún cuando no es parte del ORB, existe la especificación de *Servicio de colección de objetos* (Object Collection Specification) [36]. La cual permite manejar grupos de objetos permitiendo mejorar el balance de carga y aumentar la disponibilidad.

La tabla 3-3 resume las características generales de CORBA.

3.4 CORBA Tolerante a Fallas (CORBATaF)

El objetivo de CORBA Tolerante a Fallas (CORBATaF) [11] es proveer, a través de una especificación "abierta" y tomando como base la especificación de CORBA soporte para aplicaciones que, por su naturaleza, requieran infraestructura tolerante a fallas, este soporte se da a través de la replicación de objetos en las aplicaciones. La característica principal de esta infraestructura es la de mantener la consistencia entre las réplicas, tanto en condiciones normales como en condiciones de falla.

3.4.1 Tipo de modelo

CORBATaF sigue el mismo modelo a objetos distribuidos que CORBA, el cual describe la creación, identidad, búsqueda y operación de un objeto en un ambiente distribuido.

Así también, en esta especificación se define el concepto de **dominio de tolerancia** a fallas. Donde cada dominio de tolerancia podría tener varios sitios y varios grupos de objetos, ver sección 3.4.2. Todos los grupos de objetos dentro del dominio de tolerancia son creados y manejado por un sólo Manejador de Réplicas (Replication Manager), pero esto no impide que los objetos pertenecientes a distintos dominios de tolerancia se puedan comunicar entre sí.

Además, cada grupo de objetos tiene asociado un conjunto de propiedades tolerantes a fallas, tales como: estilo de replicación, número inicial de miembros, número mínimo de miembros, etc.

3.4.2 Descripción de un objeto

Un objeto en CORBATaF sigue la misma descripción que un objeto en CORBA más la integración de interfaces con las cuales se puede obtener o cambiar su estado.

CORBATaF al utilizar la redundancia en los objetos como mecanismo de tolerancia a fallas y creando grupos del mismo objeto replicado. Define un **objeto tolerante a fallas,** como aquel que es replicado, creado y manejado dentro de un grupo de objetos. Dado que el grupo de objetos es una abstracción, el cliente no tiene conciencia de las réplicas de los objetos (transparencia de replicación) y por lo mismo, si falla una de las réplicas no lo sabrá (transparencia de fallas).

[1] Se hacer la observación que se utilizo la referencia [44] y que la tolerancia a fallas en CORBA sale como un capítulo aparte en la referencia [11] y se incorpora a la especificación de CORBA hasta la referencia [26].

3.4.3 Arquitectura

El ejemplo mostrado en la Fig. 3-10 es un ejemplo de implementación de la especificación para CORBA Tolerante a Fallas; otras implementaciones pueden ser posibles.

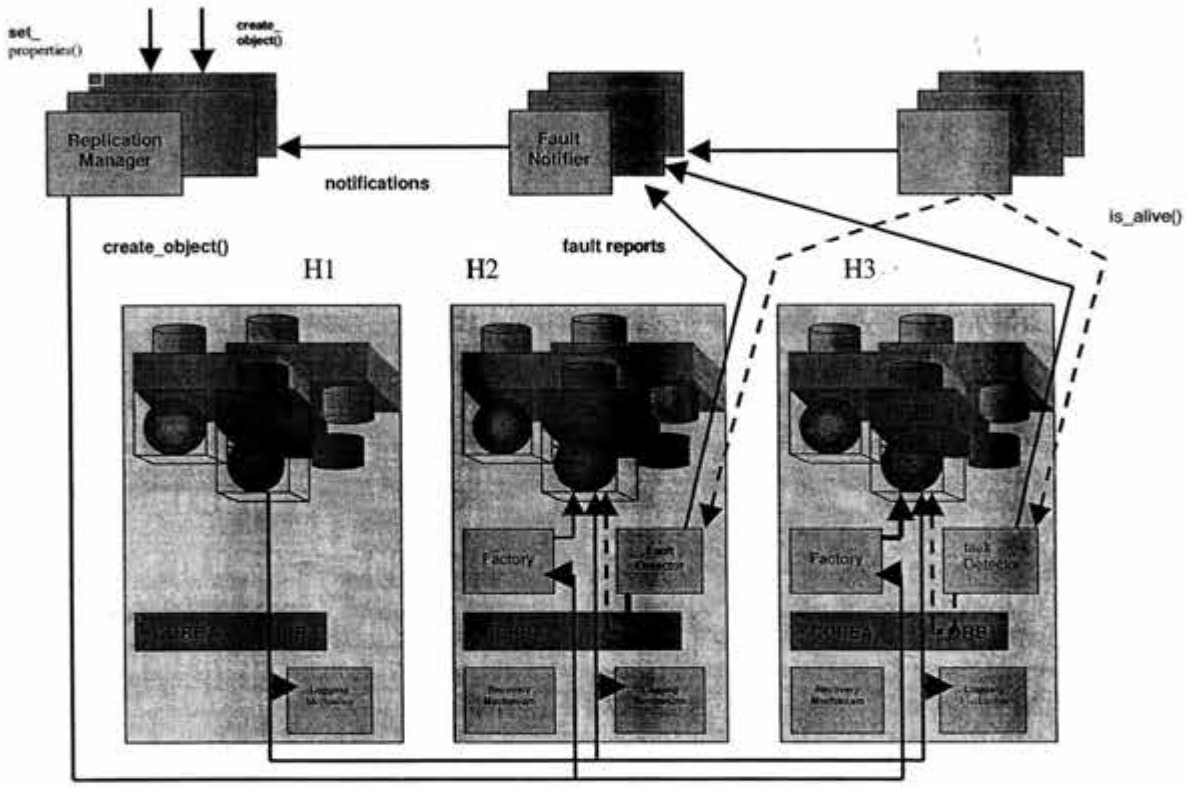


Fig. 3-10 Arquitectura de CORBA Tolerante a Fallas [11]

En la parte alta de la Fig. 3-10 se muestran varios componentes de la infraestructura tolerante a fallas (Replication Manager, Fault Notifier, Fault Detector), todos ellos son implementados como objetos CORBA. A nivel lógico, sólo hay un Manejador de Réplicas (Replication Manager) y un Notificador de Fallas (Fault Notifier) en cada dominio de tolerancia a fallas, pero a nivel físico existe un representante de ellos en cada sitio para poder soportar las fallas. El Manejador de Réplicas hereda de las interfaces PropertyManager, ObjectGroupManager y GenericFactory.

En la parte baja de la Fig. 3-10 se muestran tres sitios, como sigue:

- Una aplicación Cliente C en el sitio H1 invoca a un servidor de objetos con dos réplicas
- S1 es la réplica 1 que se encuentra en el sitio H2
- S2 es la réplica 2 que se encuentra en el sitio H3

En la misma figura, se muestra la Fábrica de objetos (Factory) y el Detector de fallas (Fault Detector) que deben de estar presentes en cada sitio y son específicos para cada uno de ellos. También se muestra la Bitácora (logging) y el Recuperador (Recovery), mecanismos que son necesarios en cada sitio. Estos últimos no son objetos CORBA pero si son parte del ORB, o se localizan entre el ORB y el sistema operativo.

Separación entre interfaces e implementación

Al igual que en CORBA, ver sección 3.3.4, las interfaces de los objetos CORBATaF están separadas de su implementación, permitiendo así que las implementaciones puedan ser hechas en diferentes lenguajes sin perder su interoperabilidad.

Servicios

CORBATaF ofrece los mismos servicios que CORBA, ver sección 3.3.5, más el servicio de tolerancia a fallas.

3.4.4 Tolerancia a fallas

El estándar provee soporte para la detección, notificación y análisis de fallas en las réplicas de los objetos, también provee puntos de verificación (checkpointing) automáticos, bitácoras y mecanismos de recuperación de fallas.

Esta especificación también provee la interfaz **PropertyManager** que permite al usuario definir las propiedades tolerantes a fallas de los grupos de objetos, tales como: Estilo de Replicación (**ReplicationStyle**), Estilo de Membresía (**MembershipStyle**), Estilo de Consistencia (**ConsistencyStyle**), Número Inicial de Réplicas (**InitialNumberReplicas**), Número Mínimo de Réplicas (**MinimumNumberReplicas**).

Estilos de replicación

Los grupos de objetos en CORBA Tolerante a Fallas soportan los siguientes estilos de replicación:

- **Sin estado** (STATELESS).- En donde el comportamiento del grupo de objetos no es afectado por las invocaciones.
- **Pasiva** (PASSIVE).- Existen dos subestilos de replicación pasiva:
  - COLD_PASSIVE.- El estado del miembro primario es extraído de una bitácora y cargado en los miembros cuando se presenta una falla en el miembro primario.
  - WARM_PASSIVE.- El estado del miembro primario es cargado en uno o más miembros periódicamente durante su operación normal.
- **Activa** (ACTIVE).- Este tipo de replicación requiere de comunicación en grupos, *multicast* totalmente-ordenado.
- **Activa con voto**[1] (ACTIVE_WITH_VOTING).- Todos los miembros del grupo ejecutan la invocación y los resultados son comparados (VOTING), y es entregado a los destinatarios sólo si la mayoría de los miembros obtuvo el mismo resultado.

[1] Este estilo de replicación no es soportado en la actual especificación, pero es una extensión anticipada para futuras versiones.

Estilos de Membresía

La actual especificación cuenta con los siguientes dos estilos de membresía en los grupos de objetos:

- **Membresía Controlada por la Aplicación** (MEMB_APP_CTRL). En este tipo de membresía la aplicación es la responsable de crear, agregar y eliminar los objetos en un grupo, así como, de definir el número inicial y el número mínimo de miembros del grupo, violando de esta forma la transparencia de replicación.

- **Membresía Controlada por la Infraestructura** (MEMB_INF_CTRL). En este caso, la replicación de objetos es transparente para el programa de aplicación. El Manejador de Réplica es el responsable de crear y agregar los miembros de un grupo, así como el número inicial y número de miembros, usando la interfaz **ObjectGroupManager.**

Detección y notificación de fallas

El primer paso para la tolerancia a fallas es la detección de las mismas, para tal propósito CORBA Tolerante a Fallas define la interfaz **PullMonitorable** la cual, contiene el método **is_alive()** que es invocado por el Detector de Fallas (Fault Detector).

Se tienen contemplados tres estilos de monitoreo (FaultMonitoringStyle) dentro de la interfaz PullMonitorable:

- PULL: En este tipo de monitoreo el Detector de Fallas interroga periódicamente al objeto (o grupos de objetos) en cuestión, a través del método is_alive () para saber si este ha sufrido una falla o no.
- PUSH[2]: En este tipo de monitoreo el objeto (o grupo de objetos) en cuestión reporta periódicamente al Detector de fallas que no ha sufrido falla alguna.
- ORED: En este tipo no hay monitoreo de fallas.

Estilos de Consistencia

En esta especificación la consistencia puede ser manejada de las siguientes dos formas:

- **Consistencia controlada por la aplicación** (CONS_APP_CTRL), en la cual, la aplicación es la responsable de la consistencia y donde el Manejador de Réplicas no promueve la recuperación.

- **Consistencia Controlada por la infraestructura** (CONS_INF_CTRL), en la cual, la infraestructura de Tolerancia a Fallas es la responsable de los puntos de verificación, bitácora y el mecanismo de recuperación, así como, el correcto manejo de Estilo de Replicación y Estilo de Membresía. En este estilo de control de réplicas, cada mensaje que es enviado al grupo de objetos es pasado a la bitácora y al Mecanismo de Recuperación de forma transparente a la aplicación. El Mecanismo de Bitácora (Logging Mechanism) guarda el mensaje en la bitácora, para que durante la recuperación en caso de fallas el Mecanismo de Recuperación (Recovery Mechanism) pueda utilizarlo. Ejemplo, en el Estilo de Replicación Pasiva, en caso de fallas, la

[2] El estilo PUSH no es soportado en la actual especificación, pero es una extensión anticipada para futuras versiones.

nueva réplica primaria comienza su operación con el estado actualizado, y ejecuta la misma secuencia de invocaciones que hubiera ejecutado la anterior réplica primaria si no hubiera fallado.

Bitácora y recuperación

Los objetos de programas de aplicación deben de heredar de las interfaces **Checkpointable** y **Updateable,** las cuales, cuentan con métodos que permiten obtener de los objetos el estado actual para guardarlo en la bitácora o actualizar su estado con uno que se encuentra en ella para poderlo recuperar en caso de que se presente una falla.

Granularidad de Monitoreo

Como ya lo hemos mencionado, el **monitoreo** en CORBA Tolerante a Fallas, se refiere a la acción que realiza el Detector de Fallas para verificar que los objetos no hayan sufrido fallas. CORBA Tolerante a Fallas propone tres tipos de granularidad de monitoreo:

- MEMB: En este tipo de monitoreo cada miembro del grupo de objetos es monitoreado de manera individual.
- LOC: En este tipo de monitoreo se elige un conjunto de objetos en distintas localidades que servirán de muestra representativa de todos los objetos de cada localidad, y en caso de que falle uno de los objetos de la muestra representativa, se supondrá que todos los objetos de su respectiva localidad han fallado.
- LOC_AND_TYPE, este caso es similar al caso anterior, con la diferencia de que al tomar la muestra representativa se elige un conjunto de objetos de un solo tipo[1], y en caso de que falle uno de los objetos de la muestra, se supondrá que todos los objetos del mismo tipo de su respectiva localidad han fallado.

Intervalos de tiempo en el monitoreo

El Detector de Fallas usará una estructura que se encuentra en **FaultMonitoringIntervalAndTimeout**, dicha estructura contiene una variable que le permitirá saber al Detector de Fallas el intervalo de tiempo que habrá de usar entre una invocación y otra al método **is_alive()**, durante el monitoreo a un objeto. Además, la estructura contendrá otra variable que le permitirá saber cuánto tiempo esperará la respuesta a la invocación antes de determinar que hay una falla.

Intervalos de tiempo en los puntos de verificación

El intervalo de tiempo que existe entre el almacenamiento del estado de un objeto en la bitácora y el almacenamiento del siguiente estado del mismo objeto, está dado por una variable que se encuentra en **CheckpointInterval** y es usado por la Bitácora.

La tabla 3-4 resume las características generales de CORBATaF.

[1] Cada objeto en el momento de ser creado se le asigna un **tipo_id** (tipo) el cual corresponde al identificador del repositorio del sitio en el cual será colocado.

CORBATaF			
Propietario			OMG
Tipo de middleware			A objeto distribuidos
Lenguaje			Multilengujes
Plataforma			Multiplataformas
Complejidad para construir grandes aplicaciones			Alta
Optimizar, agregar, o eliminar servicios			Es permitido para los servicios CORBA
Forma en que los objetos o componentes obtienen los servios			A través de APIs. Excepto la transparencia de distribución y la interoperabilidad que son automáticos
Toleran-cia a fallas	Tipo de fallas		Todas [5] excepto bizantinas y de partición de red
	n-tolerante		$n \geq 2$
	Meca-nismos	monitoreo	Pull, por miembros de grupo, por localidad o por muestra representativa
		redundancia	Activa y pasiva
		recuperación	Pasiva.- Se recupera el estado del objeto que falla de una bitácora Activa.- Cada réplica tiene el mismo estado.

Tabla 3-4 CORBA Tolerante a fallas

3.5 El Modelo de Componentes CORBA (CCM)

El modelo de objetos CORBA, debido a sus limitaciones para soportar correctamente sistemas configurables a gran escala, tuvo que evolucionar y dio origen al Modelo de Componentes CORBA (CCM)[27], el cual es un modelo a componentes del lado del servidor que permite desarrollar y desplegar aplicaciones CORBA [26]. Este modelo es la parte central de la especificación CORBA 3.0 [45].

3.5.1 Tipo de modelo

CCM es una especificación de un middleware a componentes, la cual está basada en la especificación CORBA. En CORBA las operaciones de objetos funcionan sobre plataformas que no deben de estar casadas con algún sistema en particular (Multiplataformas). Esto mismo es heredado por CCM con la diferencia de que no son objetos sino componentes los que se ejecutan.

3.5.2 Descripción de un componente

CCM sigue un modelo a componentes, el cual describe la creación, identidad, búsqueda y operación de un componente en un ambiente distribuido

Un componente CORBA es el software que representa la parte funcional (lógica de negocios) de una aplicación distribuida y que se ejecuta en un entorno llamado contenedor, el cual lo aísla del mundo exterior. Los componentes CORBA sólo son accedidos por los clientes haciendo uso de sus interfaces (o puertos).

Tipos de componentes

La especificación define cuatro categorías (tipos) de componentes, las cuales se muestran en la tabla 3-4. Las categorías de servicio, de sesión y de entidad son idénticas a su equivalente en EJB, pero la de tipo proceso, tiene las mismas características que un bean de tipo entidad, pero que no muestra su llave primaria al cliente, ver tabla 3-5 y sección 3.2.2.

Modelo de Uso De CORBA	Tipo de API Del Contenedor	Llave Primaria	Categoría de Componente	EJB Equivalente
Sin estado	Sesión	No	Servicio	Sesión S/E
Conversacional	Sesión	No	Sesión	Sesión C/E
Persistente	Entidad	No	Proceso	-
Persistente	Entidad	Sí	Entidad	Entidad

Tabla 3-5. Categorías de componentes

Para aquellos componentes que requieren persistencia, ésta puede ser manejada de las siguientes maneras:

- **Persistencia manejada por el contenedor** (*Container-managed persistence, CMP*). El desarrollador del componente simplemente define el estado que tiene que ser persistente, y el contenedor automáticamente salva o restaura el estado a un almacenamiento permanente.
- **Persistencia manejada por el usuario** (*Self-Managed Persistence, SMP*). El desarrollador del componente tiene la responsabilidad de salvar y restaurar el estado cuando el contenedor lo solicite (interfaces *callback*).

3.5.3 Arquitectura CCM

La fig. 3-11 muestra los elementos que conforman la arquitectura de CCM.

Contenedor

Un contenedor es el ambiente de ejecución para las instancias de un componente CORBA. Un contenedor, al administrar los aspectos no funcionales de un componente, oculta la complejidad de la mayoría de los servicios del sistema como son el POA (ver sección 3.3.3), transacciones, seguridad, persistencia y notificación.

El Modelo de Programación del Contenedor se compone de los siguientes elementos [28]:

- Los **tipos de APIs externos,** son el contrato entre el desarrollador del componente y el cliente del componente.
- Los **tipos de API's del contenedor,** que representa el contrato entre un componente específico y su contenedor, y son de dos tipos: **Interfaces de tipo interno** que son interfaces locales, las cuales proporcionan los servicios que el contenedor ofrece a los componentes CORBA, y las **interfaces de tipo callback** que son también interfaces locales invocadas por el contenedor pero implementadas por el componente CORBA
- El Modelo de uso de CORBA (CORBA Usage Model)

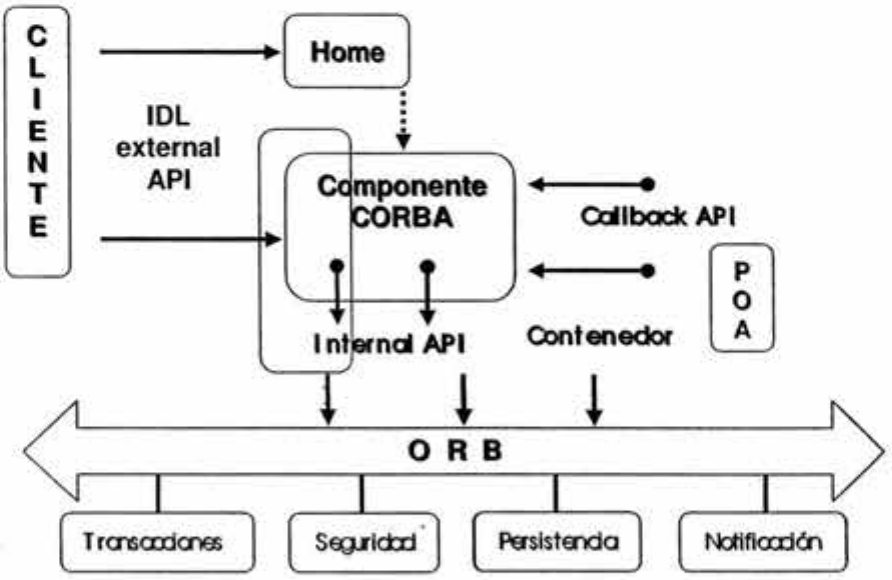


Fig. 3-11. La arquitectura CCM

3.5.4 Separación entre interfaces e implementación

En CCM las definiciones de las interfaces y sus implementaciones se realiza de manera separada, y al igual que en CORBA, la implementación de las interfaces se puede realizar en diferentes lenguajes.

CCM cuenta con el CIF, CIDL y PSDL, que son tres herramientas con las cuales un implementador de componentes puede contar para la creación de interfaces y con ello hacer más sencillo su trabajo. La obtención de estas herramientas se debe a la clara división que existe entre los tipos de componentes.

Tipo de interfaces (Puertos) en los componentes

El modelo abstracto de CCM ofrece a los diseñadores de aplicaciones los medios para definir las múltiples interfaces que ofrece y requiere un componente, así como sus propiedades. El modelo abstracto también introduce los administradores de instancias (*component homes*), los cuales se basan en dos patrones de diseño [27]: ***factory*** y ***finder.*** El *home* de un componente es el administrador de todas las instancias de un tipo específico de componente permitiendo así, entre otras funciones, su creación y recuperación.

Para definir las características de un componente, la especificación CCM introduce un lenguaje orientado a componentes, el OMG IDL3. El cual para referirse al conjunto de características del componente, introduce el término ***puerto***. La especificación define los siguientes tipos de puertos, ver Fig. 3-12.

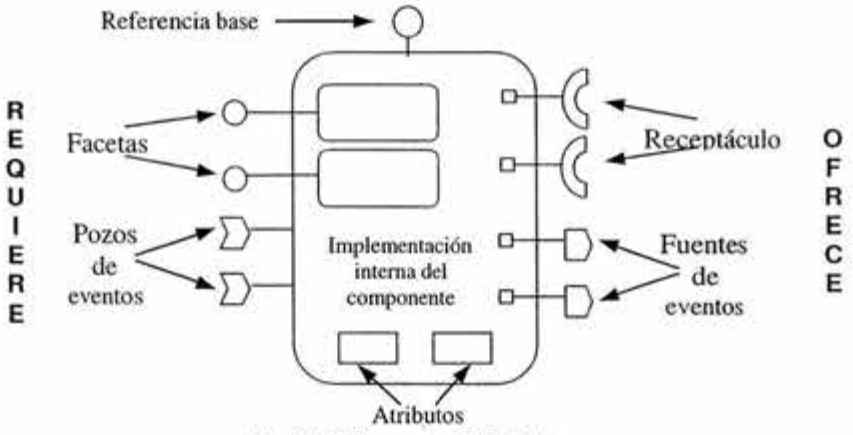


Fig. 3-12. Componente CORBA

Facetas (*facets*). Representan las interfaces de lo que **provee** un componente y son usadas de manera síncrona por los clientes. Es decir, las facetas permiten que el componente muestre a sus clientes su aspecto funcional.
Receptáculos (*receptacles*). Son las interfaces de lo que **requiere** un componente y son usadas de manera síncrona. Es decir, los receptáculos en los componentes son la contraparte de las facetas y sirven para aceptar referencias a estas con el fin de invocar las operaciones que ofrecen.
Puertos de Eventos *(event).* Representan las interfaces para hacer interactuar el componente bajo un esquema (a diferencia de las facetas y los receptáculos) de invocación asíncrona, existen dos tipos de puertos asíncronos:

- **Fuentes de eventos** (*event sources*). Representan las interfaces que **producen** eventos de un tipo específico.

- **Pozos de eventos** (*event sinks*). Permite a las instancias de componentes **consumir** eventos de un tipo específico.

Atributos. Son las **propiedades** configurables de un componente

Referencia base. Esta referencia les permite a los clientes **acceder** a los distintos puertos de la instancia de un componente.

CCM	
Propietario	OMG
Tipo de middleware	A componentes
Lenguaje	Multilenguajes
Plataforma	Multiplataformas
Complejidad para construir grandes aplicaciones	Media
Cambiar, agregar, o eliminar servicios	Es permitido para los servicios CORBA
Forma en que los objetos o componentes obtienen los servicios	Existen servicios que se obtienen de forma automática o a petición del desarrollador, y hay otros que sólo a petición del desarrollador son obtenidos.
Tolerancia a fallas	No contempla tolerancia a fallas y sólo cuenta con excepciones para evitar la propagación de las mismas

Tabla 3-6 CCM

3.5.5 Servicios

Los servicios[1] entregados por CCM de manera automática a los programadores de componentes son: ocultamiento de los detalles de la red, manejo de ciclo de vida del componente, seguridad, persistencia, transacción y notificación de eventos. Además, el programador cuenta con las herramientas para empaquetar y desplegar de manera automática sus componentes, estas herramientas están basadas en los modelos de ensamblado y empaquetado especificado en CCM.

Es importante mencionar que al igual que en EJB, CCM sólo soporta el modelo de transacciones planas y que tiene dos formas de soportar las transacciones: manejadas por componentes o manejadas por el contenedor.

[1] Algunos de estos servicios pueden ser dejados como responsabilidad al programador haciendo uso de las herramientas e interfaces adecuadas.

En cuanto al servicio de eventos. En el contexto de CCM solamente se usa un subconjunto del Servicio de Notificación de CORBA. Este subconjunto tiene las siguientes características [27]:

- Los eventos se representan como **valuetype**s para el implementador y el cliente del componente.
- Solamente se hace uso del modelo push del Servicio de Notificación de CORBA.
- Toda la administración del canal de eventos es implementada por el contenedor.
- La seguridad y transacciones relacionadas con los puertos de eventos del componente son definidas en el descriptor de despliegue.
- Políticas de transacción para los eventos (Normal, Default y transaccional).

3.5.6 Tolerancia a fallas

La especificación CORBA 3 incorpora las especificaciones de CORBA Tolerante a Fallas y CCM a la especificación de CORBA 2.x, pero lo anterior no implica que CCM pueda ser tolerante a fallas. A decir de algunos expertos, existen las bases para que CCM pueda ser tolerante a fallas en un futuro no muy lejano, pero la aparición de estas dos especificaciones en CORBA 3 no implica que se puedan combinar CCM y CORBATaF en la actualidad.

Así que por lo estudiado, podemos decir que CCM no es tolerante a fallas, pero que si podemos decir que cuenta con algunos mecanismos (como son: manejo de transacciones, uso de excepciones) para poder implementarlo. Incluso, en [40] se menciona que CCM puede llegar a manejar réplicas sin hacer cambios a la especificación gracias al modelo de desplegado, pero el programador de la implementación debe de hacerse cargo de las políticas de consistencia así como de su programación.

La tabla 3-1 resume las características generales de CCM.

3.6 Séneca

En este capítulo haremos una revisión al proyecto Séneca, que es un proyecto resiente sobre la especificación de un middleware a componentes. Séneca es un modelo de middleware a componentes [49] que nació en la Universidad de los Andes en el segundo semestre del 2002 durante un curso acerca de aplicaciones basadas en componentes que se imparte a estudiantes de maestría.

La motivación de la definición de Séneca es puramente académica y se debe a la intención de tener una herramienta que describa un modelo a componentes claro que no mezcle o esconda detrás de los GUIs o detrás de los patrones, los conceptos subyacentes al modelo, como se hace en algunos modelos de middleware a componentes publicados.

Séneca es un modelo a componentes que está dividido en cuatro submodelos: modelo abstracto, modelo de programación, modelo de ensamblado y desplegado, y modelo de ejecución.

3.6.1 Modelo a componentes Séneca

Séneca es un modelo a componentes multi-interfaces y multi-instancias, que sigue la arquitectura mostrada en la Fig. 3-13.

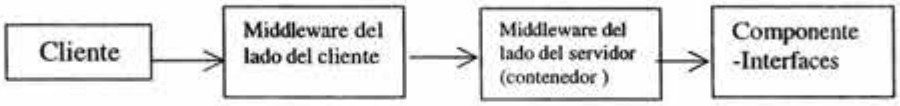


Fig. 3-13 Arquitectura

En donde:

Un **componente** es una entidad que encapsula (esconde) código y datos, y que define un conjunto de interfaces para interactuar con el exterior.

Existen dos tipos de componentes en Séneca. Los cuales son:

- Componentes simples
- Componentes colección. Un conjunto de componentes.

También se pueden realizar una relación de uso entre componentes, esto es, los componentes dentro de su definición hacen referencia a las interfaces de otros componentes para hacer uso de ellos. Además, que los componentes pueden tener instancias que pueden ser usadas por más de un cliente a la vez o no, y tener persistencia o no.

Una **interfaz** es un mecanismo usado para definir la forma exacta en que un componente proveerá su funcionalidad a otros componentes o aplicaciones. Pero también las interfaces podrán servir para definir la forma en que el componente recibirá la funcionalidad de otros componentes o aplicaciones, la cual le permite realizar su trabajo, ver 3.6.2.

Un **contenedor**, como en EJB, es aquella entidad que le permite a un componente manejar, de forma automática, los aspectos no funcionales del sistema, tales como: el manejo de instancias (compartidas o no compartidas), ciclo de vida, conexión entre componentes, persistencia, transacciones, etc.

3.6.2 Modelo Abstracto

En el modelo abstracto de Séneca se selecciona el tipo de componente que será usado en la aplicación, ya sean componentes simples (**component,** tabla 3-7 línea 20) o componentes colección (**component collectionOf,** tabla3-7 línea 33). También en este modelo se debe dar la descripción del componente, que incluye:

1. Las interfaces que provee el componente (**provides,** tabla 3-7 **línea 21**) y las interfaces requeridas o usadas por el componente (**uses,** tabla 3-7 línea 27).
2. Las propiedades (o atributos) del componente que definen su estado (**state,** tabla 3-7 línea 28).
3. Y en el caso de un componente de tipo colección se da la definición del tipo del componente y el nombre de la llave primaria (**key**, tabla 3-7 línea 33) con el cual se buscará un componente específico en la colección.

Como parte de este modelo se dan las siguientes definiciones:

- **Interfaz**. En Séneca una interfaz (**interface,** tabla 3-7 línea 2) es un conjunto de firmas (o signatures) de métodos (al estilo Java). Así que, cada firma consiste de un nombre, una lista de definición de parámetros, un tipo de regreso y una lista de definiciones de excepciones que el método dispara. El tipo de regreso y el tipo de los parámetros pueden ser algunos tipos válidos en Java o en las interfaces definidas en el modelo abstracto.

  En Séneca se necesitan definir si los parámetros pueden ser modificados o no dentro de los métodos. Así los parámetros pueden ser: Entrada de sólo lectura (**in**, tabla 3-7 línea 3), o entradas o salidas modificables (**out**).

- **Estado de un componente**. El estado de un componente es definido por un conjunto de atributos. Un atributo tiene un tipo y un nombre. Los atributos pueden ser de lectura y escritura (por **omisión**) o pueden ser restringidos a sólo lectura a través de la cláusula **readonly** (tabla 3-7 línea 29).

- **Colección de componentes**. Los elementos (que son componentes) de un componente de tipo colección son creados de manera dinámica, o lo que es lo mismo, a tiempo de ejecución.

  Al construir un componente colección se debe de cumplir:

  1. Los elementos de la colección deben de ser componentes con atributos de estado.
  2. El nombre de la llave primaria debe de ser uno de los atributos de estado de los elementos de la colección.

  Para manipular la colección Séneca provee una interfaz básica de manera automática para cada tipo de colección. Esta interfaz contiene dos métodos: create() y find() (tabla 3-7 líneas 17-18).

  Ahora al igual que los componentes de tipo simple, los componentes de colección en su definición pueden proveer más de una interfaz, usar otros componentes y declarar uno o más atributos de estado.

Ejemplo: En la tabla 3-7 se define el modelo abstracto de tres componentes, dos de tipo simple y otro de tipo compuesto, así como sus interfaces.

Línea	Código
1	
2	Interface Idisplay { // Interfaz que define un solo método
3	Void print (in String s); // firma definida en la interfaz con
4	} // parámetros de entrada sólo lectura
5	interface ISuma {
6	int sumar (in int op1, in int op2);
7	}
8	interface IResta {
9	int restar (in int op1, in int op2);
10	}

```
11 interface IstateCAritmetica {    //interfaz que debe de ser creada de manera automática
12   String getId();                 //por el compilador al detectar la cláusula state en
13   Int getCummulator();            //el componente Caritmetica
14   Void setCummulator(int c);
15 }
16 interface IbasicCollectionCAritmetica { //Interfaz creada de manera automática por el compilador
17    IstateCAritmetica find(in String id) throws FinderException;
18    IstateCAritmetica create(in String id) throws CreateException;
19 }
20  component Cdisplay {          //Componente simple que no usa otros componentes y que usa la
21   provides Idisplay;           // interfaz IDisplay para proveer su funcionalidad a
22  }                             //otros componentes o programas
23 component Caritmetica {        //Componentes simple que requiere de otro
24   provides Isuma;              // componente para funcionar
25   provides Iresta;
26   provides IstateCAritmetica;
27   uses IDisplay;               //Especifica la utilización de IDisplay para funcionar
28   state {                      //Estado que define dos atributos o propiedades, uno de sólo lectura
29    readonly String id;         // y otro de lectura escritura (por omisión)
30    int cummulator;
31   }
32  }
33  component CAritmeticas collectionOf(CAritmetica) key id { //Componente de tipo colección
34     provides IBasicCollectionCAritmetica;      //donde los elementos son componentes de tipo
35     state {                                    // CAritmetica y que utiliza como llave el id de su
36       in numElem;                              // estado para crearlos o buscarlos, además de que
37      }                                         //puede tener un estado propio
38 }
```

Tabla 3-7. Ejemplo de Modelo Abstracto

3.6.3 Modelo de programación

En el modelo de programación, se describen las características de los estados de los componentes (si es que lo tienen), la asociación entre cada interfaz definida en el componente y la clase que la implementa, así como, la relación entre los atributos del componente y los valores de los argumentos de los métodos en cada interfaz definida.

En el resto de la sección describiremos cada uno de los anteriores aspectos.

Características de los componentes. En Séneca existen dos características en los componentes que deben de ser definidas en el modelo de programación, estas características son: si los componentes son compartidos (**shared**), entonces todos los clientes usarán sólo una instancia en particular del componente, si no son compartidos (**not shared,** tabla 3-8 línea 2), entonces cada cliente usará una instancia que será independiente (ver sección 3.6.5).

Es importante señalar que si el estado de un componente de tipo colección es definido como compartido, este valor es propagado a los elementos de la colección sobrescribiendo el valor dado en su descripción.

Línea	Código
1	Component CAritmetica { //componente no persistente y no compartido (para cada cliente que
2	not shared //haga una invocación a él se creará una nueva instancia)
3	not persistent
4	ISuma implementedBy SumaImpl
5	IResta implementedBy RestaImpl;
6	IstateCAritmetica implementedBy StateAritmeticaImpl; //interfaz e implementación
7	} //hechas de forma automática por el compilador

Tabla 3-8 ejemplo de modelo de programación

Implementación de interfaces y estado del componente. La asociación entre la interfaz y su implementación se hace en el modelo de programación, y se realiza colocando el nombre de la interfaz seguido de la cláusula **implementedBy** (Tabla 3-8 línea 4) y después el nombre de quien la implementa. Además, en este modelo el estado de un componente puede ser modificado o consultado, sin hacer uso de la interfaz del estado, como se muestra en el siguiente ejemplo.

3.6.4 Modelo de ensamblado y desplegado

En el modelo de ensamblado y desplegado se definen el nombre de instancias y el lugar donde éstas serán creadas a través de la cláusula **Instances** (tabla 3-9 línea 1), además de realizar las conexiones (**connections**, tabla 3-9 línea 9) entre los distintos componentes que se especificaron con la cláusula **uses** (tabla 3-7, línea 27) al momento de definir al componente en el modelo abstracto.

Línea	Código
1	Instances
2	{
3	host localhost contains { // Lugar o sitio donde podrán ser creadas las instancias
4	CDisplay display; // El cliente podrá referenciar a un componente de tipo CDisplay
5	CAritmetica aritmetica; //a través del nombre display
6	}
7	}
8	
9	connections {
10	from aritmetica to display; // se realiza la composición se uso entre componentes, el
11	} //componente CAritmetica utiliza el componente display
13	}

Tabla 3-9. Ejemplo de modelo de desplegado y ensamblado

3.6.5 Modelo de ejecución

El modelo de ejecución de Séneca es basado en objetos contenedor. Un contenedor es aquel que provee un ambiente propicio en tiempo de ejecución para las instancias de los componentes. El

contenedor es responsable de proporcionar el aspecto no funcional de un componente incluyendo instanciación, distribución, ciclo de vida, transacción, persistencia, balanceo de carga, etc.

Instanciación de un componente. Para usar una interfaz provista por un componente, primero se requiere tener una referencia de una instancia de un componente. Hay dos formas de obtenerla:

- Desde una aplicación externa (e.g servlet, aplicaciones stand-alone), en este caso se requiere usar el método lookup();

```
Context c = new InitialContext();
String instanceName ="display";
IDisplay s = (IDisplay)c.lookup(url);
s.print(" Hello World ");
assertTrue(true);
```

- Desde una aplicación que provea una interfaz de un componente, en este caso se requiere usar el método **connect** dando el tipo de componente requerido y la referencia de la instancia que hace la petición:

```
IDisplay id = (IDisplay)SenecaServices.connect("CDisplay", this);
```

La tabla 3-10 resume las características generales de Séneca.

Séneca	
Propietario	Proyecto de investigación Universidad de los Andes
Tipo de middleware	A componentes
Lenguaje	Multilenguajes
Plataforma	Multiplataformas
Complejidad para construir grandes aplicaciones	Media
Cambiar, agregar, o eliminar servicios en el middleware	NO permitido, ya que no existe (aún) una interfaz para cada servicio.
Forma en que los objetos o componentes obtienen los servicios	Existen servicios que se obtienen de forma automática o a petición del desarrollador, y hay otros que sólo son obtenidos a petición del desarrollador.
Tolerancia a fallas	No contempla tolerancia a fallas. Sin embargo, tiene mecanismos como: las excepciones que son lanzadas por los métodos y que, con ellas se podría evitar la propagación de una falla.

Tabla 3-10 Séneca

3.7 Comparación entre los distintos modelos

En esta sección haremos una comparación entre los distintos modelos estudiados CORBA, DCOM, CCM, EJB y Séneca. La comparación la hacemos según el paradigma que es usado en las aplicaciones soportadas en los middlewares: objetos distribuidos o componentes

3.7.1 Comparación entre modelos a objetos distribuidos

En esta sección hemos omitido a CORBATaF (CORBA tolerante a fallas) en las comparaciones, ya que CORBATaF no es precisamente una especificación completa, si no una extensión de CORBA que se hace con el único propósito de agregar tolerancia a fallas al modelo CORBA, y al compararlo con los otros modelos obtendríamos el mismo resultado que con CORBA pero con tolerancia fallas.

	DCOM	CORBA
Propietarios	Microsoft	OMG
Tipo de middleware	A objeto distribuidos	A objeto distribuidos
Lenguaje	Multilenguajes	Multilenguajes
Plataforma	Multiplataformas	Multiplataformas y heterogéneos
Complejidad para construir grandes aplicaciones	Alta	Alta
Cambiar, agregar, o eliminar servicios en el middleware	Es permitido sólo para aquellos servicios que tienen una interfaz bien definida.	Es permitido para los servicios CORBA
Forma en que los objetos o componentes obtienen los servicios del middleware	A través de APIs	A través de APIs
Tolerancia a fallas	No contempla tolerancia a fallas. Sin embargo, tiene mecanismos como: los ping y timeout. Estos mecanismos podrían ayudar al programador a implementar este servicio	No contempla tolerancia a fallas. Sin embargo, tiene mecanismos como: las excepciones y el servicio de colección de objetos, que podrían ayudar al programador a implementar este servicio

Tabla 3-11 DCOM & CORBA

De la tabla 3-11 podemos notar que la diferencia que existe entre las especificaciones de DCOM y CORBA no es grande, aún que en esencia estas especificaciones tienen diferentes filosofías en su arquitectura. DCOM es llamado "la versión CORBA de Microsoft" y quizás la principal diferencia

entre esas dos especificaciones es la escalabilidad, donde CORBA es mucho más escalable que DCOM.

En el caso de tolerancia a fallas, no se puede decir gran cosa para DCOM y CORBA, ya que ambas especificaciones no la contemplan. Aún cuando las dos especificaciones cuentan con algunos mecanismos para poder implementar la tolerancia a fallas, estos mecanismos no son suficientes para poder hacer uso de algunas de las formas de replicación (por ejemplo, la replicación pasiva o primaria) en sistemas ya creados. Además de que el desarrollador de la aplicación será el encargado de diseñar e implementar la tolerancia a fallas.

La tabla 3-11 resume la comparación de las características generales de DCOM y CORBA.

3.7.2 Comparación entre modelos a componentes

	EJB	Séneca	CCM
Propietario	SUN	Proyecto de investigación Universidad de los Andes	OMG
Tipo de middleware	A componentes	A componentes	A componentes
Lenguaje	Java	Multilenguajes	Multilenguaje
Plataforma	Multiplataformas y heterogéneos	Multiplataformas	Multiplataformas y heterogéneos
Complejidad para construir grandes aplicaciones	Media	Media	Media
Cambiar, agregar, o eliminar servicios en el middleware	No permitido	NO permitido, ya que no existe (aún) una interfaz para cada servicio.	Es permitido para los servicios CORBA
Forma en que los objetos o componentes obtienen los servios del middleware	De forma automática y en algunos casos a petición del implementador	Existen servicios que se obtienen de forma automática o a petición del desarrollador, y hay otros que sólo son obtenidos a petición del desarrollador.	Existen servicios que se obtienen de forma automática o a petición del implementador, y hay otros que sólo a petición del implementador son obtenidos.
Tolerancia a fallas	No contempla tolerancia a fallas y sólo cuenta con excepciones para evitar la propagación de las mismas.	No contempla tolerancia a fallas. Sin embargo, tiene mecanismos como: las excepciones que son lanzadas por los métodos y que con ellas se podría evitar la propagación de una falla.	No contempla tolerancia a fallas y sólo cuenta con excepciones para evitar la propagación de las mismas.

Tabla 3-12 EJB y Séneca & CCM

En la tabla 3-12 podemos notar las diferencias entre las especificaciones de EJB, CCM y Séneca. Esas diferencias en lo general[1] no son muy grandes, notando que la mayoría de ellas se encuentran en el lenguaje en que se implementan los componentes en las especificaciones. Esto es, Mientras que en EJB se debe de utilizar sólo Java, en CCM se pueden utilizar varios lenguajes (C++, Java, etc.) y en Séneca no se especifica.

Quizás las diferencias más grandes no se encuentran al comparar EJB, CCM y Séneca entre sí, sino al comparar éstas con las dos anteriores (DCOM y CORBA), notando que al cambiar los modelos a objetos distribuidos a modelos a componentes el trabajo de los desarrolladores de aplicaciones distribuidas es más sencillo, ya que la separación de la parte funcional y la no funcional de un sistema, así como el proporcionar de manera automática los servicios por parte del middleware, disminuyó la complejidad en la implementación de tales aplicaciones.

En cuanto a la tolerancia a fallas, hay un gran avance en los modelos a objetos distribuidos en comparación a modelos a componentes, ya que CORBA cuenta con CORBATaF para el servicio a tolerancia a fallas, mientras que los modelos a componentes sólo cuentan con algunos mecanismos para implementarlo, más no con el servicio, peor aún, en cuanto a tolerancia a fallas CORBA perdió al evolucionar en CCM ya que no se puede utilizar el Servicio de colección de objetos de CORBA [36] de manera natural con los componentes.

La aparición de CCM y CORBATaF en una misma especificación (CORBA 3) hizo pensar a algunas personas que CCM ya era tolerante a fallas, lo cual es un error. Pero deja ver la necesidad de una especificación de un middleware a componentes tolerante a fallas.

La tabla 3-12 resume las diferencias que se encuentran entre las características generales de EJB, Séneca y CCM.

[1] La diferencia entre EJB, CCM y Séneca en lo general no es muy grande, pero si nos fijamos en los detalle existe una gran diferencia entre las arquitecturas propuestas.

4 APROXIMACIÓN A UN MIDDLEWARE TOLERANTE A FALLAS

Como se reporta en el capítulo anterior, los middlewares a objetos distribuidos tienden a evolucionar hacia middlewares a componentes, como lo podemos corroborar con Corba y CCM. Esta evolución permite tener en CCM una mejor separación de los aspectos funcionales y los no funcionales en una aplicación (de la que era posible en Corba), lo cual, se traduce en una disminución de complejidad al construir aplicaciones distribuidas. Por otro lado, también se constata la preocupación de los investigadores y las empresas por dotar a los middlewares de servicios de tolerancia a fallas, dado el uso cada vez más frecuente de este tipo de soluciones en servicios críticos.

La integración de tolerancia a fallas en el middleware no es una tarea sencilla debido a que tradicionalmente un sistema tolerante a fallas se construye integrando elementos de detección, encargados de vigilar los componentes del sistema, elementos de notificación que señalan la presencia de una falla y elementos de recuperación que corrigen los efectos de la falla. El impacto de la falla en el funcionamiento de un sistema está en función de cuántos de estos elementos se integraron. Un sistema que no contiene ninguno de los elementos mencionados no puede reaccionar a la falla y simplemente su funcionamiento se desvía. Un sistema que integra elementos de detección y notificación, podrá al menos señalar la falla al usuario del sistema (otro sistema o un ser humano). Si además se cuenta con elementos de recuperación es posible que el sistema vuelva a su funcionamiento normal en algún momento, o mejor aún que nunca exhiba un comportamiento distinto del especificado. Los mecanismos utilizados en ambos casos son diferentes pues en uno se requiere regresar a un estado coherente previo y en el otro tener listo un elemento de reemplazo que tome el lugar del que falló.

Por otro lado, también se debe de tener en cuenta que los elementos de detección y de recuperación que se quieran integrar al middleware deberán contemplar el tipo de fallas al que se quiere hacer frente. Para cada tipo de falla, las técnicas de detección son diversas y los mecanismos de recuperación también (ver secciones 2.2 y 2.3). Además, se deberá de tener cuidado de que dichos elementos de tolerancia puedan ser integrados a los elementos que ya se encuentran en el middleware sin afectar su correcto funcionamiento.

Finalmente, es importante mencionar que la complejidad del sistema tolerante fallas no sólo depende del número de elementos que lo integren y del número de fallas de las que el sistema se pueda recuperar. Sino también depende de los elementos de la aplicación que se harán tolerantes a fallas y del nivel de transparencia que se les quieran dar a las fallas.

Con el fin de acercarnos a nuestro objetivo de producir un middleware a componentes que ofrezca el servicio de tolerancia a fallas mas adecuado a la aplicación, en este capítulo hacemos una propuesta tomando en cuenta un modelo de tolerancia a fallas preciso y proponemos proporcionar la adaptabilidad tomando uno de tres 3 niveles de niveles de transparencia, que serán definidos mas adelante.

De esta manera, la propuesta queda organizada de la siguiente forma. En la sección 4.1 describimos el modelo de tolerancia a fallas a soportar, en la sección 4.2 se muestra el modelo abstracto de componentes considerado, en la sección 4.3 se enuncian los mecanismos a utilizar para soportar el modelo, en la sección 4.4 se muestran las nuevas responsabilidades que tendrán que asumir los actores del modelo abstracto y por último en la sección 4.5 se muestran los algoritmos propuestos para lograr la integración de todos los elementos en el sistema.

4.1 Modelo de tolerancia a fallas

4.1.1 Definición del modelo

De acuerdo a lo enunciado en la sección anterior, antes de construir un sistema tolerante a fallas es importante definir:

- El tipo de fallas que se quiere tolerar
- El impacto de cada una de ellas en el comportamiento del sistema y
- La cantidad de cada una de ellas que el sistema es capaz de tolerar.

Llamaremos a este conjunto de parámetros, el **modelo de tolerancia a fallas** (MTF) del sistema. El MTF nulo es aquel en donde no se contempla ninguna falla. Esta definición del modelo de tolerancia a fallas aumenta a la definición de Nivel de Tolerancia a Fallas (NTF) presentada en [50] la cantidad de fallas que se desea tolerar.

Una vez definido el MTF, es relativamente simple determinar los elementos de detección, de notificación y de recuperación que deben incluirse en la aplicación. Debe considerarse sin embargo, que el costo de la tolerancia a fallas en un sistema depende directamente de las características del MTF que se quiere integrar en una aplicación. Un MTF determina la infraestructura mínima para ejecutar un sistema que lo ofrezca. Por ejemplo, no se puede tener un sistema que tolere dos fallas de caída de servidor si sólo se cuenta con dos sitios. En otros términos, no todos los MTF se pueden implementar en cualquier infraestructura.

Flexibilidad y transparencia

En nuestro contexto decimos que una tarea se hace de manera transparente cuando el beneficiario de la tarea no participa en el esfuerzo para llevarla a cabo mientras que, la flexibilidad de una solución, está en proporción directa con la cantidad de variaciones que permite.

Con respecto a un MTF este puede ser provisto por el middleware de manera que:

A) Todas las aplicaciones tengan ese mismo NTF
B) Sólo las aplicaciones que se elija, tengan el NTF
C) Sólo algunos componentes de las aplicaciones que se elijan tengan el NTF

Mientras que la primera opción tiene la ventaja de ser absolutamente transparente para el constructor de aplicaciones, puede penalizar a aquellas aplicaciones que no requieren la Tolerancia a fallas. La segunda opción le deja a cargo la responsabilidad de decidir si se integra o no la TaF y la última, de decidir cuales componentes son tolerantes a fallas y cuales no.

Por otro lado, la primera opción carece de flexibilidad mientras que en la segunda permite al menos elegir y en la última muchos diferentes esquemas tienen cabida. De hecho, la idea de hacer convivir el MTF nulo con otro, se puede generalizar y permitir sistemas que integren componentes con distintas garantías de fiabilidad.

Opcionalidad

Tradicionalmente la tolerancia a fallas se incluye como una tarea más a programar por el constructor de la aplicación. Esta manera de atacar el problema no es coherente con la tendencia actual de dejar al programador de aplicaciones a cargo sólo de las tareas aplicativas.

Por otro lado, programar un MTF en el middleware tampoco es una solución adecuada pues no todas las aplicaciones requieren el mismo MTF ni disponen de la misma infraestructura física. Por lo tanto, la inclusión de la tolerancia a fallas en un middleware debe ser opcional.

A largo plazo, dado que a cada MTF corresponden mecanismos distintos, sería deseable poder generar una instancia de middleware para cada MTF. El caso de un middleware sin tolerancia a fallas es aquel que considera el MTF nulo.

Así la adaptabilidad de un middleware tolerante a fallas se puede dar tomando uno de tres 3 niveles de niveles de transparencia definidos: uno en donde todas las aplicaciones tienen el MTF especificado, otro en donde sólo algunas aplicaciones tienen el MTF especificado y finalmente uno en dónde sólo los componentes seleccionados por el constructor de la aplicación tienen el MTF especificado

4.1.2 Modelo Propuesto

Las fallas toleradas en este modelo serán las de tipo de *caída-pausa*, ver sección 2.2.2, donde el sitio al regresar de la falla tendrá el estado en el que se encontraba hasta antes de la caída.

En este modelo se propone que la tolerancia fallas se haga a nivel de sitio y la redundancia se haga a nivel de componente, usando sólo un respaldo por cada componente. Entonces sin contar las fallas del sitio donde se encuentra el cliente final, el sistema tolera como mínimo la falla de un sitio y como máximo (según la distribución de los respaldos de los componentes en los sitios del sistema) podría soportar la falla de hasta el total de los sitios del sistema menos uno.

Es necesario precisar que entre el tiempo en que notifica un componente su *regreso de una falla* y el tiempo en que este componente esta *listo para tomar el lugar* del componente respaldo, existe un tiempo de actualización, en el cual si el componente primario falla, el sistema fallará, ya que esta acción es tomada como si ambos componentes tuvieran una falla al mismo tiempo. De esta forma podemos asegurar, a lo menos, un sistema 1-Tolerante.

Entonces, el modelo de tolerancia a fallas de nuestra propuesta es:

<caída-pausa, recuperación,1>

Niveles de transparencia de TaF

En esta sección describimos los posibles niveles de transparencia de tolerancia a fallas que podrá tener el desarrollador de las aplicaciones distribuidas cuando se implemente esta especificación en un modelo middleware a componentes.

Se visualizan tres posibles niveles de transparencia, estos niveles de transparencia difieren según la percepción de la tolerancia a fallas que tenga el desarrollador de la aplicación a la hora de hacer su implementación. Estos niveles son:

- **N1**, en este nivel, el desarrollador no requiere que agregar algún tipo de código en la implementación de la aplicación, ya que todas las aplicaciones que se ejecuten en ese middleware serán tolerantes a fallas y con la ventaja de que no se requiere modificar el modelo original del middleware. El inconveniente es que existirán aplicaciones que no requerirán el servicio de tolerancia a fallas y de todas maneras lo obtendrán de manera automática incrementando así el tráfico en la red, y consumiendo tiempo de procesamiento en los sitios.
- **N2,** en este nivel, se deja al desarrollador de la aplicación la posibilidad de elegir (según su criterio y experiencia) si su aplicación completa requiere del servicio de tolérente a fallas o no, permitiendo así no incrementar el tráfico en la red con mensajes poco útiles. Al elegir este nivel de tolerancia es necesario hacer cambios en la sintaxis y semántica de los mecanismos de definición de aplicaciones en el modelo de componentes original.
- **N3** en este caso, se deja al desarrollador de la aplicación la posibilidad de elegir qué componentes de la aplicación requieren tolerancia a fallas y cuales no. Es evidente que en este caso la trasparencia a la tolerancia a fallas es menor que en los casos anteriores y se recomienda para gente con mayor experiencia, ya que si bien se tiene la oportunidad de disminuir más (con respecto al caso anterior) el trafico en la red, también una mala elección en alguno de los componentes ocasionará que la aplicación en su totalidad falle. Al elegir este nivel de tolerancia es necesario hacer cambios en la sintaxis y semántica de los mecanismos de definición de los componentes en el modelo original.

La elección del nivel de transparencia del servicio de la tolerancia a fallas, que se le quiera dar al desarrollador de la aplicación distribuida, se deja al implementador o integrador del servicio en el middleware.

4.2 Modelo genérico (cliente-servidor-contenedor)

En esta sección se mencionan las características de los actores y el comportamiento que deben de seguir dentro del sistema distribuido antes de que sean implementados los mecanismos y algoritmos de tolerancia a fallas que se propone en las secciones 4.3, 4.4 y 4.5.

4.2.1 Actores

En los modelos básicos de componentes distribuidos existen tres actores principales que son:

- *Cliente.*- Es aquel que hace la invocación de algún método al componente, el cliente podría ser un programa de aplicación o un componente que invoca el método de otro componente. Además por ser un sistema distribuido el cliente podría o no ejecutarse en el mismo sitio que el componente.
- *La capa de servicios.*- Es aquella que intercepta la invocación que hace el cliente y se encarga de transportarla a través de la red y entregarla al componente, además esta capa se encarga de proporcionar servicios al componente, como son; la seguridad, manejo de transacciones, manejo de ciclo de vida, persistencia, etc. Estos servicios pueden darse al componente de manera automática o a través de una invocación de los mismos. La capa de servicios y el componente corren en el mismo sitio y cada capa de servicios puede manejar más de un componente a la vez.

- *Componente.*- es la lógica de negocios encapsulada en una pieza de software a la cual se puede acceder únicamente a través de sus interfaces, además de tener una buena definición de su comportamiento, así como sus requerimientos de entrada y salida.

4.2.2 Comportamiento de los actores

La invocación de un cliente a un método de algún componente ejecutará los siguientes pasos:

a) El cliente hace una invocación de algún método a un componente a través de sus interfaces.
b) La capa de servicios intercepta la invocación y la transporta hasta donde se encuentra el componente y se la entrega, no sin antes aplicarle algún servicio por ejemplo, la concurrencia en la cual la capa de servicios tienen la facultad de decidir si las invocaciones se ejecutan de manera paralela o serial.
c) El componente ejecuta la invocación y al hacerlo pueden surgir dos casos, que son:
   - Si el componente para ejecutar la invocación requiere hacer una o más invocaciones a otro(s) componente(s), entonces el componente que hace la invocación pasa a ser cliente del otro componente y se comienza en el paso b) de manera recursiva para cada una de las invocaciones.
   - Si el componente puede ejecutar la invocación que le hace su cliente sin la necesidad de hacer una invocación a otro componente, entonces pasamos al paso d).
d) El componente regresa el resultado a la capa de servicios.
e) La capa de servicios lo transporta hasta el cliente y se lo entrega.

Los anteriores pasos se pueden ver gráficamente en la Fig. 4-1.

Fig. 4-1 Comportamiento de los actores en el modelo abstracto

4.3 Mecanismos

Suposiciones del sistema

1. Supondremos un sistema distribuido el cual usa un middleware a componentes que a su vez utiliza llamadas a procedimientos remotos (o mensajes síncronos) para realizar la comunicación entre los sitios.
2. Supondremos un canal de comunicación confiable, unidireccional y *FIFO* (primero que entra primero que sale, por sus siglas en inglés) que une a los sitios.
3. El sistema no cuenta con memoria compartida.
4. La latencia de comunicación, i.e. el tiempo que tarda un mensaje en llegar del emisor al receptor, es finito y acotado.
5. No existe un reloj común y es necesario el uso de mensajes para poder sincronizar los sitios.
6. Cuando un componente falla, este detiene su funcionamiento evitando fallas del tipo bizantinas y no permitiendo se corrompan los datos de la memoria estable.

7. Los procesos locales en los sitios del sistema son determinísticos, esto es, que la ejecución en el mismo orden de las mismas operaciones siempre genera el mismo estado.
8. Supondremos también que los sitios no tienen una carga excesiva y siempre que no hayan sufrido una falla podrán contestar a tiempo.

Los mecanismos utilizados para lograr la semántica de tolerancia a fallas son:

mi Redundancia.- se hace uso de *redundancia* a nivel de componentes, colocando un componente en un sitio y una copia de éste en un sitio distinto. También se usa redundancia a nivel de mensajes, ya que los mensajes de invocación llegarán tanto al componente como a su copia.

mii Técnica de respaldo primario.- en esta versión, utilizaremos sólo una réplica para los componentes (ver sección 2.4.3), esto es con el objetivo de que el modelo resultante sea simple, además de que en la práctica (en la mayoría de las veces) esto es suficiente. Se deja la posibilidad de que en posteriores versiones se incremente la cantidad de réplicas.

miii Bitácoras.- Cada componente tendrá asignada un área en memoria *principal* para guardar el historial de sus invocaciones, resultados y cambios de estado debidos a la ejecución de las invocaciones. También se contará con una bitácora en *memoria persistente* donde serán guardadas las claves de los componentes que se encuentran en uso.

miv Puntos de revisión.- Para evitar el crecimiento desmedido de las bitácoras se efectuarán periódicamente en ellas puntos de revisión (ver sección 2.3.2).

4.4 Nuevas Responsabilidades de los actores

Para lograr la semántica de fallas propuesta en el MTF, ver sección 4.1, los actores del modelo abstracto tendrán nuevas responsabilidades las cuales se mencionan a continuación.

Cliente. No tendrá nuevas responsabilidades.

Capa de servicios (middleware). Seguirá ofreciendo todos los servicios que ya proporcionaba, además de realizar las siguientes tareas:

- En el sitio del cliente

  ci La primera vez que haga una invocación a un componente lo hará al componente primario.

  cii Espera un tiempo determinado (timeout) después de hacer la invocación, si antes del término de este tiempo, el sitio del componente no responde, se tomará esa acción como una falla del sitio.

  ciii Al darse cuenta de que el componente ha fallado, reenviará la invocación pero en esta ocasión lo hará al componente respaldo.

- En el sitio de ambos componentes (primario y respaldo)

  a) Hará la administración de la *bitácora* asignada a cada componente.
  b) Maneja los mensajes para la consistencia entre los sitios de los componentes primario y el sitio del componente respaldo (según el algoritmo de la Fig. 4-5).
  c) Cada vez que un componente sea creado guardará en una área de memoria estable una clave que indique qué componente fue creado.

d) Cuando regrese de una falla buscará en memoria estable qué componentes eran utilizados y los creará nuevamente, después enviará un mensaje al componente principal para que sepa que ya regreso de la falla y sea actualizado para que pueda ser integrado nuevamente al sistema tomando el lugar de componente respaldo.
e) Cada vez que un componente sea eliminado, en la memoria estable se eliminará la clave del componente.
f) Verificará con la ayuda de la bitácora si el componente ya ejecutó cada una de las invocaciones que arriban al sitio. Si una invocación ya fue ejecutada por el componente no será pasada nuevamente a él para que la reejecute, en ese caso, se obtendrá el resultado de la bitácora y se envía al cliente.

- En el sitio del componente primario

a) Cuando haga una invocación al sitio del componente respaldo, esperará la respuesta un tiempo determinado (timeout), si antes del término de este tiempo, el sitio del componente respaldo no responde, se tomará esa acción como una falla del sitio.
b) Cuando detecte que el sitio del componente respaldo ha fallado, dejará de enviarle las invocaciones que usaba para mantener la consistencia entre ambos, y las enviará nuevamente hasta que el sitio del componente respaldo haya regresado de la falla y esté listo.

- En el sitio del componente respaldo

c) Cuando detecte que las invocaciones son hechas por el cliente y no por el componente primario, entonces sabrá que el componente primario ha fallado y manejará la falla (según el algoritmo de la Fig.4-6)

Componente. Seguirá trabajando según sus especificaciones agregando la siguiente funcionalidad:

pi Contará con dos interfaces, una que permita a la capa de servicios obtener su estado interno y otra que permita la capa de servicios cambiar dicho estado.

4.5 Algoritmos para tolerar las fallas

En esta parte mostraremos los algoritmos tolerantes a fallas que se incorporan al modelo de componentes distribuidos con los cuales se obtiene la integración de técnicas de redundancia en la capa de servicios.

Para hacer más comprensible la forma en que fue estructurada esta sección podemos auxiliarnos de la Fig. 4-2.

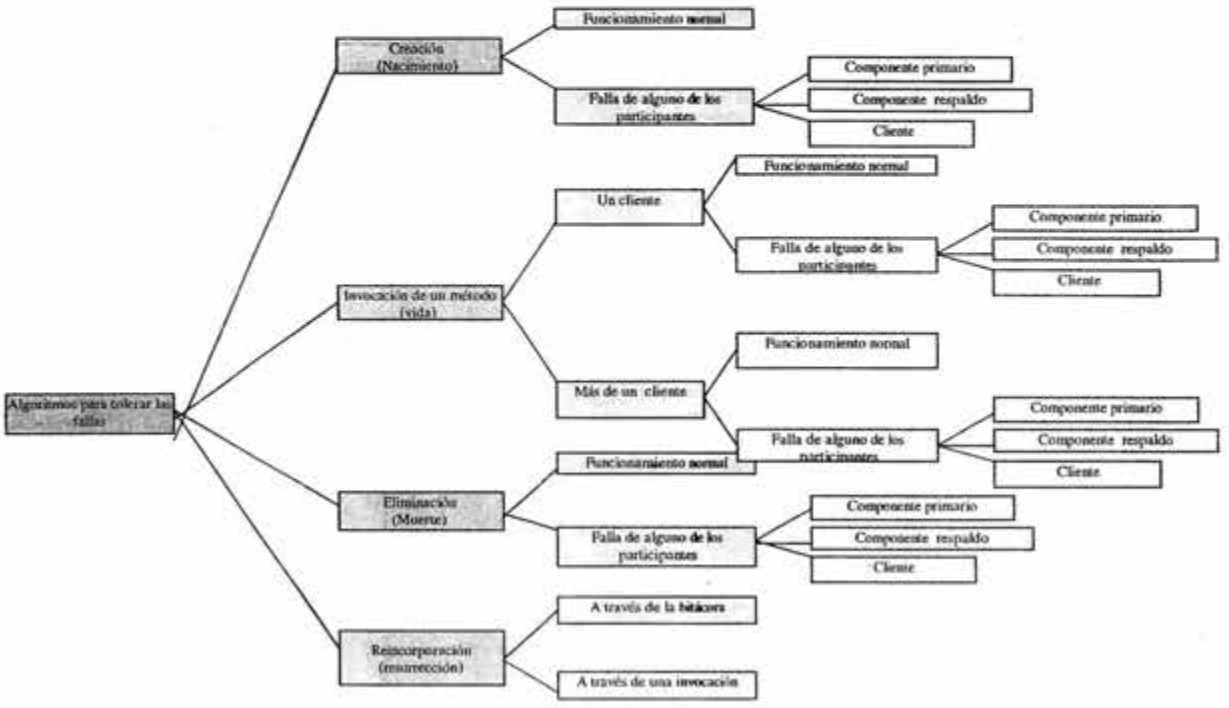


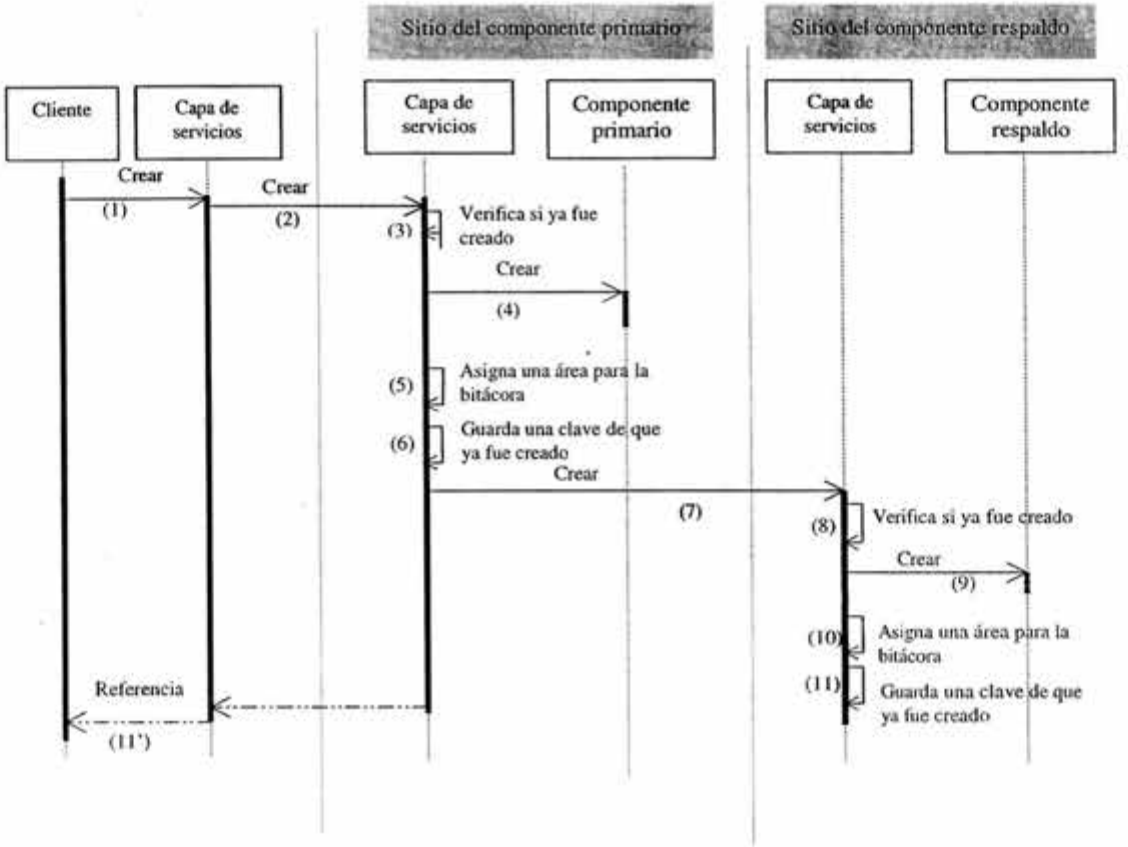
Fig. 4-2 Estructura de la sección 4.5

4.5.1 Algoritmo para la creación de un componente

La invocación del método "crear" es un caso que se debe de estudiar de forma separada de las demás invocaciones ya que hasta antes de que se ejecute dicha invocación los componentes no existen (no existe una instancia de ellos con la cual se pueda trabajar) y no existe su bitácora, por lo que no se les puede hacer un manejo de mensajes como el descrito en el capítulo de Algoritmo de invocación de un método, véase la Fig. 4-5.

Funcionamiento básico normal (libre de fallas)

Cuando un cliente invoca el método para crear una instancia de un componente, suponiendo que no existen fallas, se siguen los pasos de la Fig. 4-3.

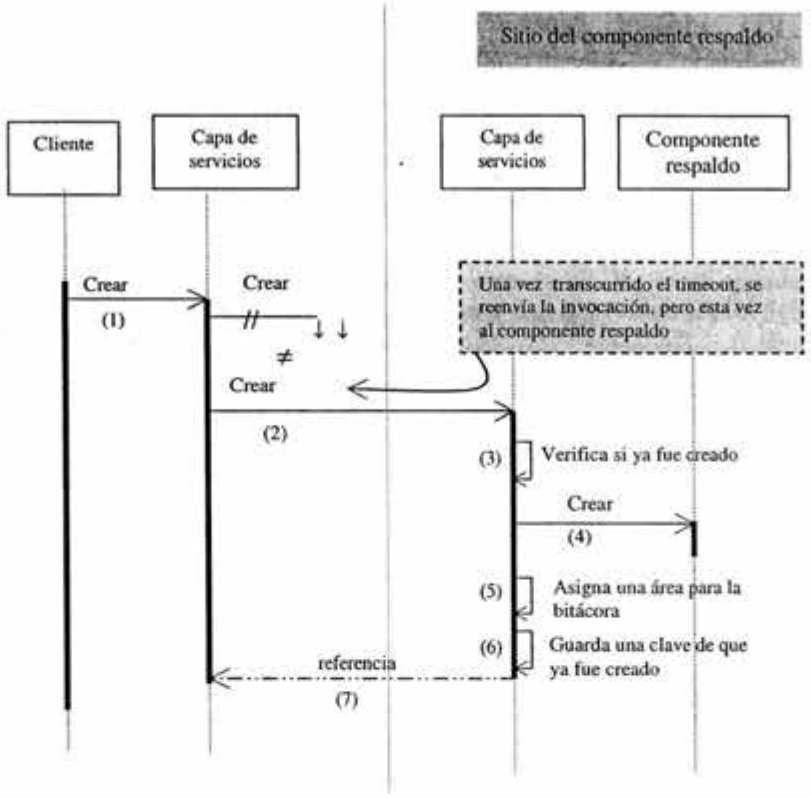


Donde:

(1) El cliente invoca el método crear componente
(2) La capa de servicios envía de forma transparente al cliente la invocación a través de la red al componente primario
(3) La capa de servicios verifica que no se haya hecho la misma solicitud con anterioridad (esto es con la idea de que también el cliente puede ser un componente el cual puede tener componente respaldo)
(4) Se crea la instancia del componente
(5) Se asigna un área para la bitácora del componente primario
(6) Después de crear el componente primario la capa de servicios guarda la clave del componente en memoria persistente
(7) La capa de servicios invoca la creación del componente respaldo
(8) También se verifica si la creación del componente fue hecha con anterioridad
(9) Se crea la instancia del componente respaldo
(10) Se asigna un área para la bitácora del componente respaldo
(11) La capa de servicios guarda en la bitácora de memoria persistente la clave del componente creado
(11') Se envía[1] la referencia del componente al sitio del cliente y se entrega al cliente para que pueda hacer uso de él

Fig. 4.3 Creación de los componente primario y respaldo

[1] Aún que en realidad el mensaje 11' es parte de la invocación (1) por ser un mensaje síncrono, sólo se dibuja para ilustrar la dependencia que tiene el cliente con la capa de servicios



Donde:

(1) El cliente hace la invocación al componente respaldo
(2) La capa de servicios al haber transcurrido el tiempo de espera (timeout) y no recibir contestación a la invocación de creación que se hizo al componente primario, reenvía de forma transparente al cliente la invocación a través de la red al componente respaldo
(3) Verifica si la creación del componente fue hecha con anterioridad, si resulta positiva la verificación, entonces saltamos al paso (7)
(4) Se crea la instancia del componente
(5) Asigna un área de la memoria primaria para guardar la bitácora
(6) La capa de servicios guarda en la bitácora de memoria persistente la clave del componente creado
(7) Se enviar la referencia al cliente

Nota: Si se saltó del paso (3) al paso (7), entonces para la reincorporación al sistema del componente que falló se hace *a través de la búsqueda en la bitácora de la memoria persistente*, véase Fig. 4-10. En otro caso se hace *a través de la petición del sitio del componente primario*, véase la Fig. 4-11

Fig. 4-4 Método Crear con falla del componente

Invocación con falla de alguno de los participantes

Las fallas se pueden presentar en cualquiera de los participantes y podemos distinguir tres casos, que son:

- Falla del sitio del componente primario
- Falla del sitio del componente respaldo
- Falla del sitio del cliente

Falla del sitio del componente primario

La capa de servicios del cliente al no obtener respuesta del componente primario a la invocación de creación, tomará esta acción como una falla del componente y reenvía la invocación pero en esta ocasión lo hace al componente respaldo, quien tomará el lugar del componente primario y comenzará a trabajar sin respaldo siguiendo los pasos de la Fig.4-4.

Falla del sitio del componente respaldo

El sitio del componerte primario al no obtener respuesta a la invocación de crear el componente por parte del sitio respaldo después de un tiempo determinado (timeout), tomará esta acción como una falla de sitio del componente respaldo y comenzará a trabajar sin respaldo, además, la reincorporación del componente cuando regrese de la falla se hará *a través de la petición del sitio del componente primario*, (ver Fig. 4-10)

Falla del sitio del cliente

El cliente podrá ser modelado como un componente, lo cual implica que tendrá un respaldo (réplica) que en caso de fallas tomará el lugar del componente de una forma consistente y sin pérdida de información, para lograrlo, tanto el componente como su respaldo seguirán el algoritmo de la Fig. 4-5. Pero a diferencia de las demás invocaciones, la invocación crear no se puede ser anidada, es decir, no se requiere de la ayuda de otro componente para ejecutar la operación.

4.5.2 Algoritmo en la invocación de un método

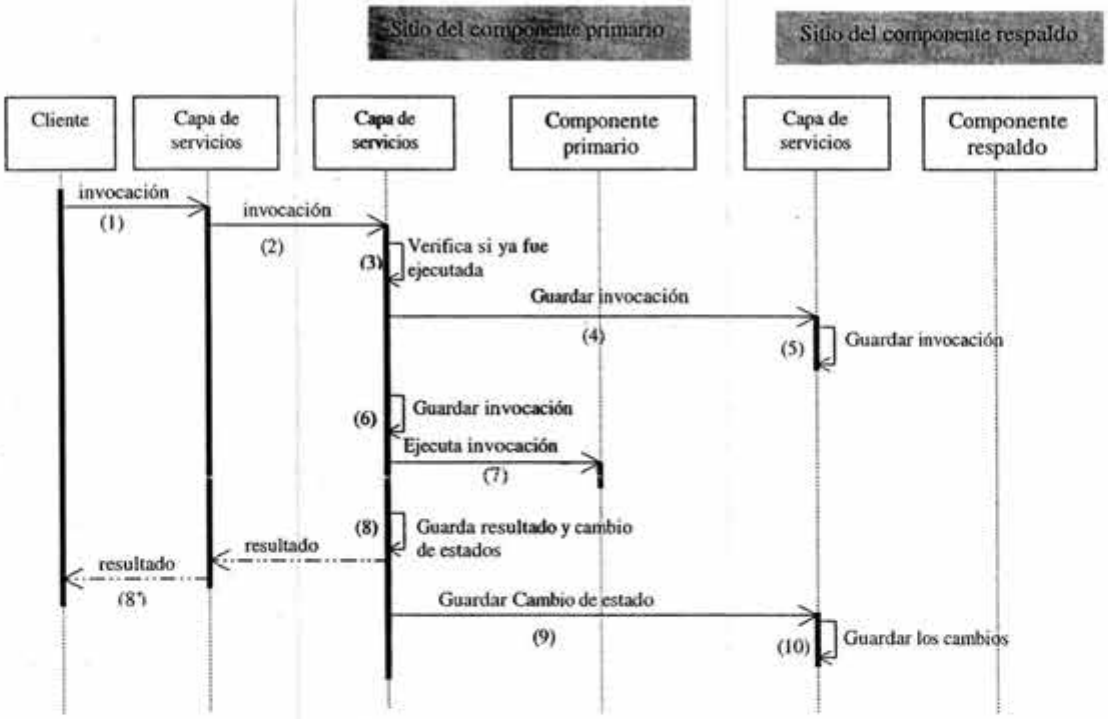
En la invocación de un método por parte de un cliente puede generar dos casos en la capa de servicios del lado del cliente, uno es cuando un solo cliente hace una invocación al componente a la vez y el otro caso es cuando más de un cliente hace la invocación de un método al mismo componente a la vez.

4.5.2.1 Algoritmo para un solo cliente

La invocación de un método por parte de un cliente sobre un componente primario desencadena un par de mensajes entre el sitio de este componente y el sitio de su componente respaldo, estos mensajes sirven para mantener la consistencia entre los dos componentes.

Funcionamiento básico normal (libre de fallas)

La comunicación que hay entre el sitio del componente primario y el sitio de su componente respaldo (cuando ninguno de los dos sufre una falla) se muestra en la Fig. 4-5. Note que suponemos que tanto el componente primario como el componente respaldo ya fueron creados (ver sección 4.5.1).



Donde:

(1) Es la invocación a un método por parte del cliente

(2) la capa de servicios envía de forma transparente al cliente la invocación a través de la red al componente primario

(3) La capa de servicios verifica que la invocación no se haya ejecutado antes, si ya fue ejecutada brincamos al paso (8')

(4) La capa de servicios del componente primario envía una copia de la invocación al componente respaldo

(5) La capa de servicios del sitio del componente respaldo guarda la invocación en la bitácora

(6) Se guarda la invocación en la bitácora del componente primario

(7) Se ejecuta la invocación en el componente primario y regresa a la capa de servicios el resultado y los cambios hechos a su estado

(8) Se guarda el resultado en la bitácora del componente primario

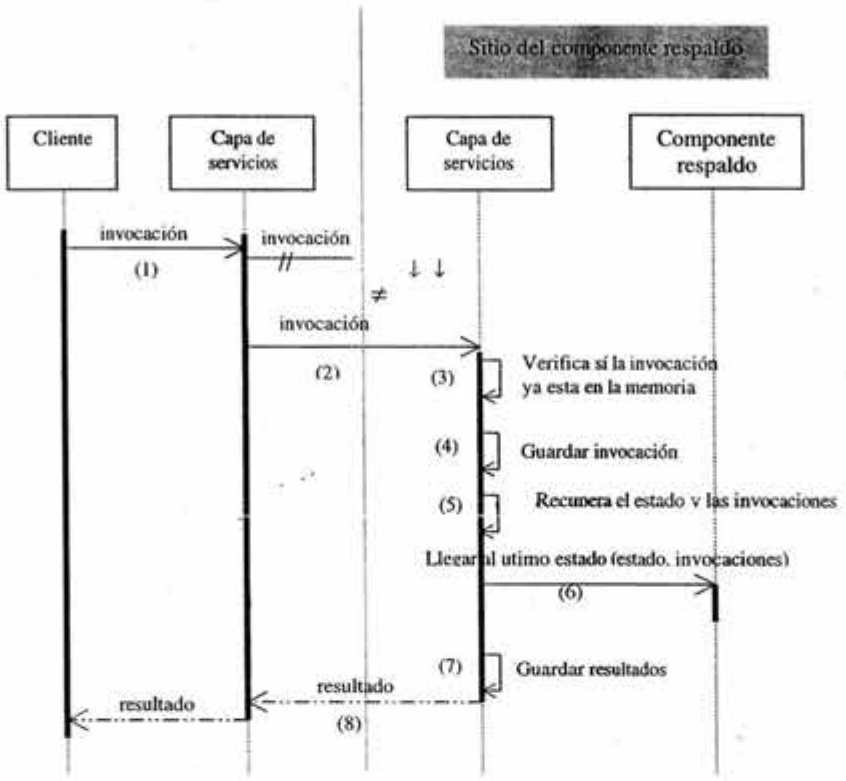
(8') Resultado que es enviado de forma transparente al cliente a la capa de servicios que se encuentra en su sitio y es entregado [1] al cliente

(9) Se envían los cambios del estado al sitio del componente respaldo

(10) El sitio del componente respaldo guarda los cambios en su bitácora

Fig. 4-5 Invocación con un solo cliente

[1] Aún que en realidad este mensaje es parte de la invocación (1) por ser un mensaje síncrono, sólo se dibuja para ilustrar la dependencia que tiene el cliente con la capa de servicios



Donde:

(1) El cliente envía la invocación al componente primario
(2) La capa de servicios al haber transcurrido el tiempo de espera (timeout) y no recibir contestación a la invocación que se hizo al componente primario, reenvía de forma transparente al cliente la invocación a través de la red al componente respaldo
(3) La capa de servicios verifica si la invocación ya está en la bitácora, si se encuentra en la bitácora brincamos al inciso (8), también en ese momento se da cuenta que el componente primario ha fallado si es que la invocación es hecha por el cliente
(4) Guarda la invocación en la bitácora si es que no se encontraba ya en ella
(5) La capa de servicios recupera el estado del componente que hay en la bitácora junto con la invocación no ejecutada
(6) Ambos, el estado y la invocación se entregan al componente respaldo para que actualice su estado
(7) La capa de servicios guarda el resultado de la invocación ejecutada durante la actualización antes de enviarla al cliente (en este caso el componente no entrega el resultado junto con el cambio de estado ya que no hay componente respaldo a quien entregarlo)
(8) Se envía el resultado al cliente

Fig. 4-6 Invocación con falla del componente primario

Invocación con falla de alguno de los participantes

Como las fallas se pueden presentar en cualquiera de los participantes, podemos distinguir tres casos, que son:

- Falla del sitio del componente primario
- Falla del sitio del componente respaldo
- Falla del sitio del cliente

Falla del sitio del componente primario

La capa de servicios que se encuentra en el sitio del cliente al darse cuenta de que el componente primario no responde a la invocación de uno de sus métodos dentro del tiempo estipulado (timeout), tomará esta acción como una falla del sitio del componente primario, y reenviará la invocación pero esta vez lo hará al sitio del componente respaldo.

En el sitio del componente respaldo la capa de servicios al darse cuenta que el mensaje de invocación ha sido enviado por el cliente sabrá que el componente primario ha fallado, y tomará las acciones descritas en la Fig. 4-6.

Falla del sitio del componente respaldo

Por los incisos svii y sviii de las nuevas responsabilidades, ver sección 4.4, de la capa de servicios, la capa de servicios del componente primario es la encargada de detectar que el componente respaldo ha sufrido una falla, y la forma de hacer esto es a través de la no-respuesta a las invocaciones que hace la capa de servicios de componente primario a la capa de servicios del componente respaldo, entonces la capa de servicios del componente primario deja de hacer estas invocaciones y seguirá trabajando sin componente respaldo hasta que éste le envíe un mensaje de que nuevamente está listo.

Falla del sitio del cliente

Un cliente podrá modelarse como un componente, lo cual implica que tendrá un componente respaldo que en caso de fallas tomará el lugar del componente de una forma consistente y sin pérdida de información, para lograrlo tanto el componente como su respaldo seguirán el algoritmo de la Fig. 4-5.

En algunos casos los componentes requieren de otros componentes para ejecutar las invocaciones generando invocaciones anidadas, en tales casos sigue aplicando lo expuesto en el párrafo anterior, en donde cada cliente de los distintos niveles del anidamiento tendrá su componente respaldo.

4.5.2.2 Algoritmos para más de un cliente a la vez

La capa de servicios al recibir más de una invocación a la vez las forma en el orden en que fueron llegando y las pasa al componente una a una, es decir, la serializa.

Funcionamiento básico normal (libre de fallas)

Si no existe alguna falla por alguno de los participantes, entonces, podremos utilizar el algoritmo de la Fig. 4-5 para la ejecución de cada una de las invocaciones sin interferir en el manejo de los mensajes entre el componente principal y el componente respaldo. Ver Fig. 4-7.

Note que también suponemos que tanto el componente primario como el componente réplica ya fueron creados, ver sección 4.5.1.

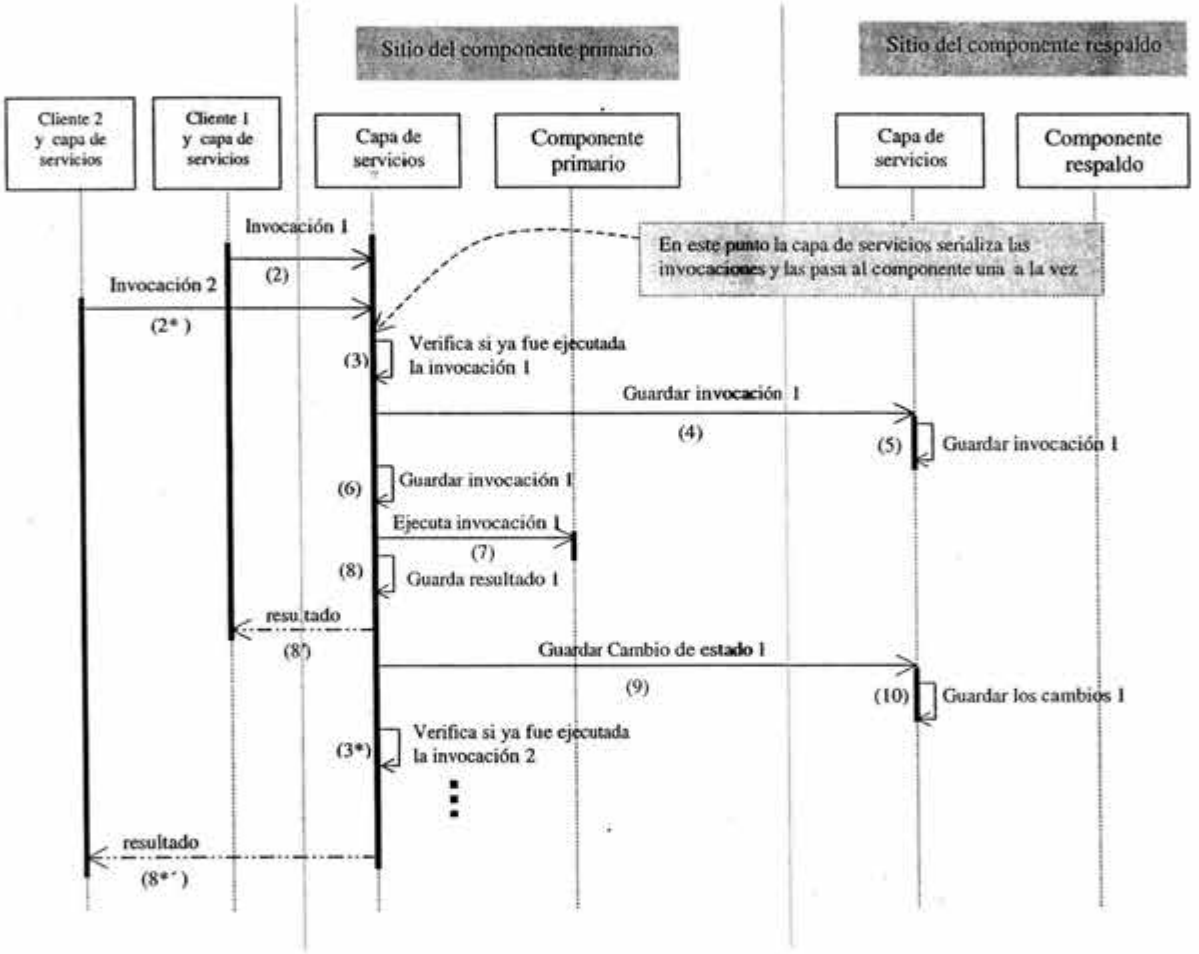


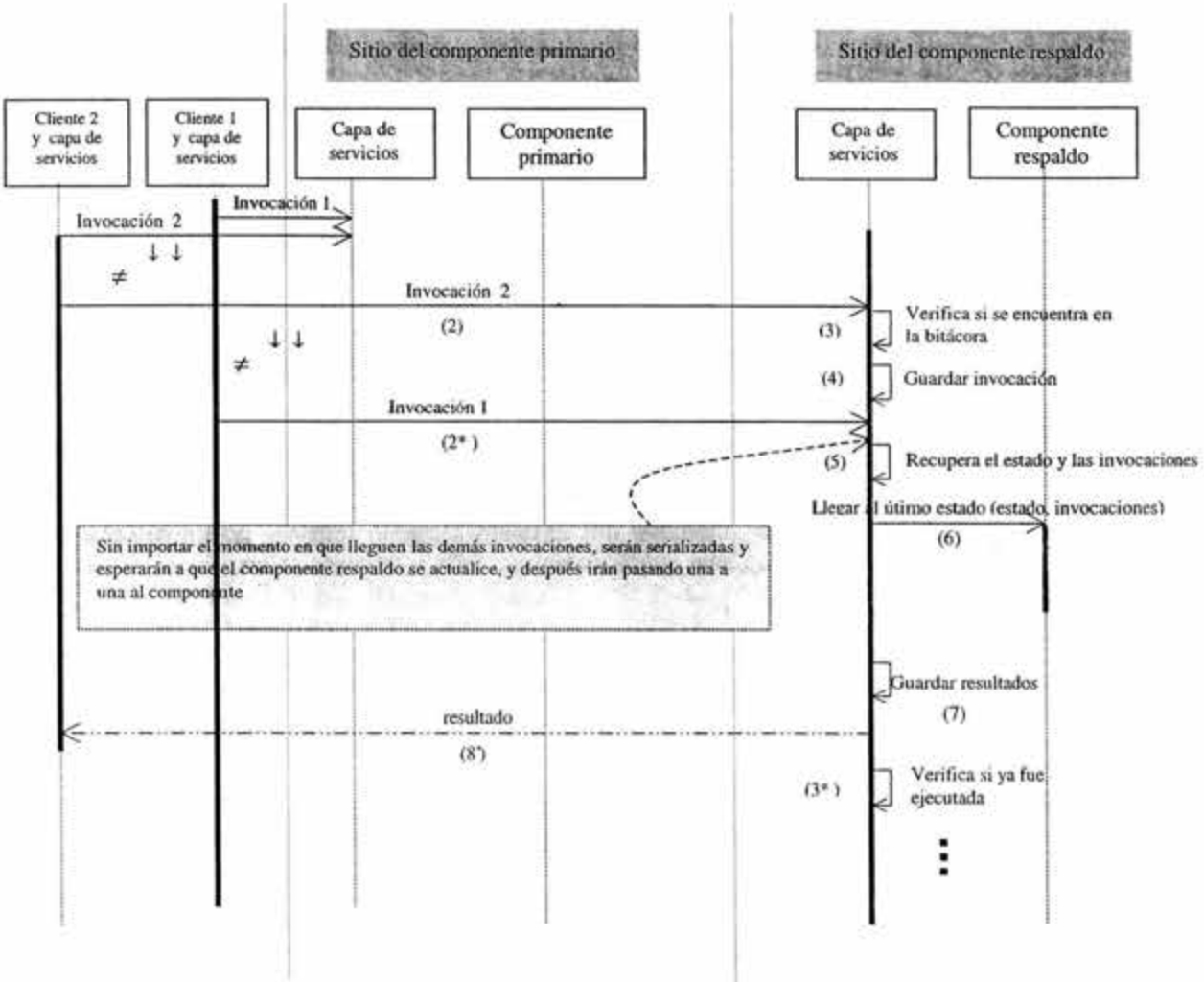
Fig. 4-7 Diagrama de ejecución en serie

Invocación con falla de alguno de los participantes

También como en el caso de un solo cliente, las fallas se pueden presentar en cualquiera de los participantes y podemos distinguir tres casos, que son:

- Falla del sitio del componente primario
- Falla del sitio del componente respaldo
- Falla del sitio del cliente

Falla del sitio del componente primario



Donde:

(2) El primero de los clientes en detecte que el componente primario ha fallado, envía la invocación al componente respaldo

(3) La capa servicios verifica si ya se ejecutó la invocación o sí la invocación ya está en la bitácora, si la invocación ya se encontraba en la bitácora brincamos al paso (5), también en ese momento se da cuenta que el componente primario ha fallado si es que la invocación es hecha por el cliente, si después llegan las demás invocaciones de los otros clientes la capa de servicios las serializará en el orden en que fueron llegando y las detendrá hasta que se termina de actualizar el componente respaldo

(4) Guarda la invocación en la bitácora si es que no se encontraba ya en ella

(5) La capa de servicios recupera el estado del componente que hay en la bitácora junto con las invocaciones no ejecutada

(6) Ambos, el estado y las invocaciones (que podría ser una o dos según el momento en que se presentó la falla en el componente primario) se entregan al componente respaldo para que actualice su estado. El resultado de las invocaciones ejecutadas por el componente se entregan a la capa de servicios (en este caso el componente no entrega el resultado junto con el cambio de estado ya que no hay respaldo a quien entregarlo)
(7) La capa de servicios guarda en la bitácora el resultado por cada una de las invocaciones y el último estado al que llegó
(8) La capa de servicios entrega el resultado al cliente del paso (1) y este componente toma el lugar de componente primario.
(3*) Cada una de las invocaciones que se encuentran serializadas en la capa de servicios son pasadas una a una al componente y se aplicará para su ejecución el algoritmo de la Fig. 4-5 sin hacer las invocaciones al componente respaldo hasta que haya uno

Fig. 4-8 Ejecución en serie con falla del componente primario

Al igual que en el caso en que el componente es usado por un solo cliente a la vez, por los incisos cii y ciii de las nuevas responsabilidades, ver sección 4.3, la capa de servicios que se encuentra en el sitio del cliente es la encargada de detectar que el componente primario ha fallado, en este caso cada una de las capas de servicios al detectar que el componente no responde dentro de un tiempo determinado[1] (timeout), tomará esta acción como una falla y enviarán la invocación al sitio del componente respaldo.

En el sitio del componente respaldo la capa de servicios al recibir la primera invocación de un cliente sabrá que el componente primario ha fallado, y tomará las acciones de la Fig. 4-8.

Falla del sitio del cliente

En el caso en que uno o más de los clientes fallen se seguirá lo descritos en la sección 4.5.1, en la subsección *fallas del sitio del cliente,* para cada uno de los clientes por separados.

Lo que cabe aclarar en esta sección, es que si uno de los clientes falla, su respaldo tomará su lugar y reenviará la invocación la cual, quizás ya se encontraba en la fila esperando su turno. En este caso la invocación será formada nuevamente y se dejará que el algoritmo de la Fig. 4-8 resuelva el problema de la reejecución y el orden.

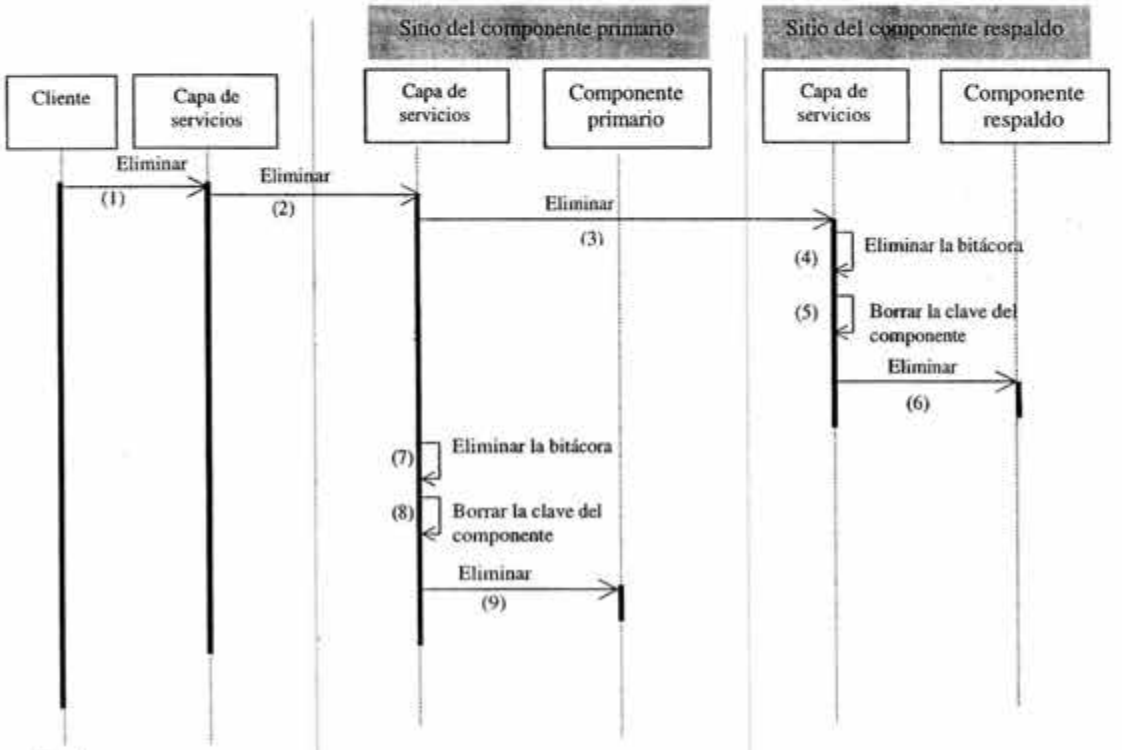
4.5.3 Eliminación del componente

El cliente al no requerir más los servicios de un componente lo puede eliminar y junto con él se eliminará su componente respaldo. También junto con los componentes se eliminarán sus respectivas bitácoras y se borrarán de las bitácoras que se encuentran en memoria persistente la clave del componente, para que en caso de falla durante la reincorporación la capa de servicios no intente crearlo nuevamente.

[1] Los timeout aún cuando fueran definidos con la misma duración en cada sitio, podrían diferir debido a que son medidos en diferentes máquinas con diferentes relojes. En este caso esa diferencia ya es tomada en cuenta en la Fig. 4-8.

Funcionamiento básico normal (libre de fallas)

Los pasos a seguir para la eliminación de un componente cuando no se presentan fallas en los sitios de los involucrados, son los siguientes:



Donde:

(1) El cliente invoca el método Eliminar
(2) La capa de servicios envía de forma transparente al cliente la invocación a través de la red al componente primario
(3) La capa de servicios al recibir la invocación de Eliminar la envía al sitio del componente respaldo
(4) La capa de servicios libera el área de memoria primaria que fue asignada para la bitácora del componente respaldo
(5) La capa de servicios borra de la bitácora que se encuentra en memoria persistente la clave del componente que será eliminado
(6) Se elimina el componente respaldo
(7) La capa de servicios libera el área de memoria primaria que fue asignada para la bitácora del componente primario
(8) La capa de servicios borra de la bitácora que se encuentra en memoria persistente la clave del componente que será eliminado
(9) Se elimina el componente primario

Fig. 4-9 Eliminación del componente

Eliminación con falla de alguno de los participantes

Como ya lo hemos mencionado antes, las fallas durante la invocación de eliminación pueden presentar en cualquiera de los participantes y podemos distinguir tres casos, que son:

- Falla del sitio del componente primario
- Falla del sitio del componente respaldo
- Falla del sitio del cliente

Falla del sitio del componente primario

En el caso en que el componente primario falla la capa de servicios del sitio del cliente al detectar la falla enviará la invocación de eliminación al sitio del componente respaldo para que sea eliminado. El componente primario con la falla de su sitio pudo haber sido eliminado parcialmente, quedando en la bitácora de la memoria persistente la clave del componente, pero la cual será eliminada cuando se intente reincorporar al componente, véase paso (3) de la Fig. 4-11.

Falla del sitio del componente respaldo

En este caso, la capa de servicios del sitio del componente primario detecta la falla del componente respaldo, si lo hace antes de enviar la invocación de eliminación, simplemente no la envía y elimina al componente primario, si lo hace después de enviar la invocación, ya no espera la respuesta y elimina al componente primario. En ambos casos, si no fue eliminado el componente respaldo a petición del primario se eliminará al intentar reincorporarse al sistema, véase paso (3) de la Fig. 4-11.

Falla del sitio del cliente

Como ya lo hemos dicho con anterioridad, al cliente se le puede modelar como un componente el que tendrá un respaldo el cual tomará su lugar en caso de que falle este, y reenviará la invocación de eliminación al componente primario.

4.5.4 Reincorporación al sistema de un componente cuando regresa de una falla

Recordemos que la tolerancia a fallas se hace a nivel de sitio, así que cuando el sitio falla, todas las instancias de los componentes que se encontraban en el sitio que falló, fallan con él, por tanto cuando el sitio regresa de la falla todas las instancias de los componentes que se encontraban en el sitio hasta antes de la falla debe de reincorporarse al sistema como componentes respaldo.

Para poder reincorporar los componentes al sistema es necesario primero crearlos nuevamente y después actualizarlos. Para la reincorporación de los componentes existen dos formas, las cuales están en función del momento en que falló el sitio y el paso en que se encontraba el algoritmo de creación (ver Fig. 4-3).

Las formas mencionadas son:

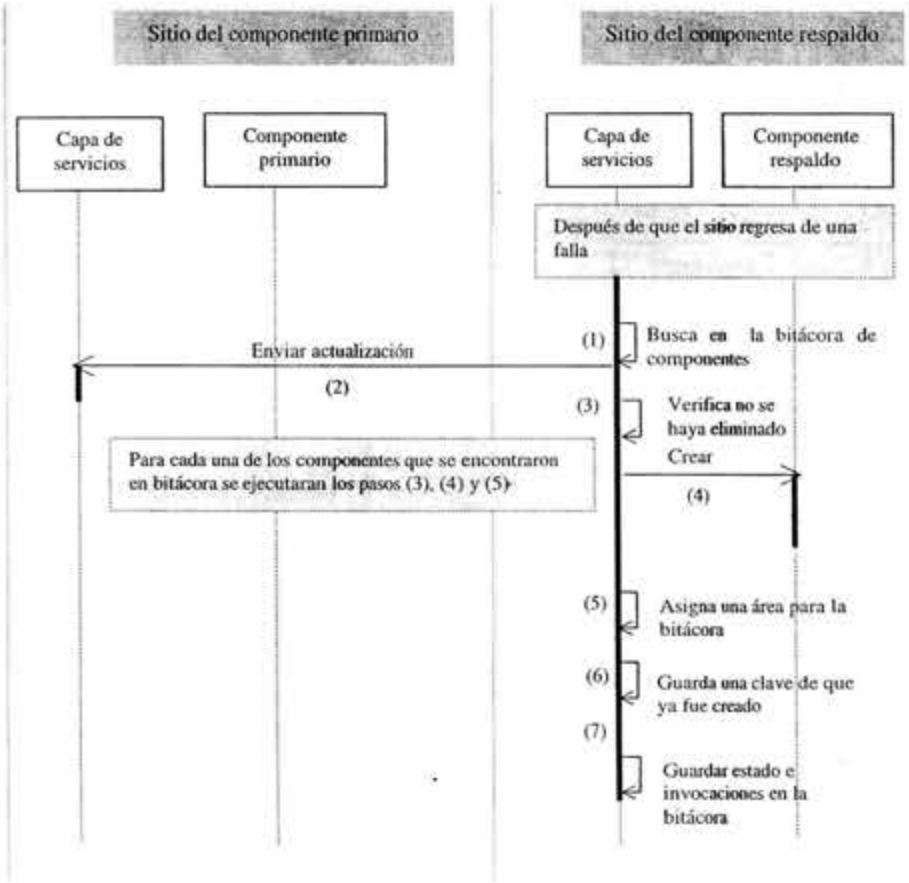
- A través de la búsqueda de sus claves en la memoria persistente que se encuentra en el sitio que regresa de la falla. Esta forma de reincorporación será usada en algunas ocasiones durante

el intento de creación, y siempre será usada cuando la falla se detecte durante el intento de eliminación o invocación. Otra forma es,

- A través de la petición explícita del componente (primario) que se encuentra en funcionamiento. Esta forma de reincorporación sólo será usada cuando la falla se detecte durante el intento de creación.

A través de la búsqueda en la bitácora de la memoria persistente

Este mecanismo de recuperación se usará cuando el sitio ha fallado después de haber guardado la clave del componente creado en la bitácora de la memoria persistente, lo cual siempre ocurre en los algoritmos de invocación y eliminación. La recuperación seguirá los pasos mostrados en la Fig. 4-10.



Donde:

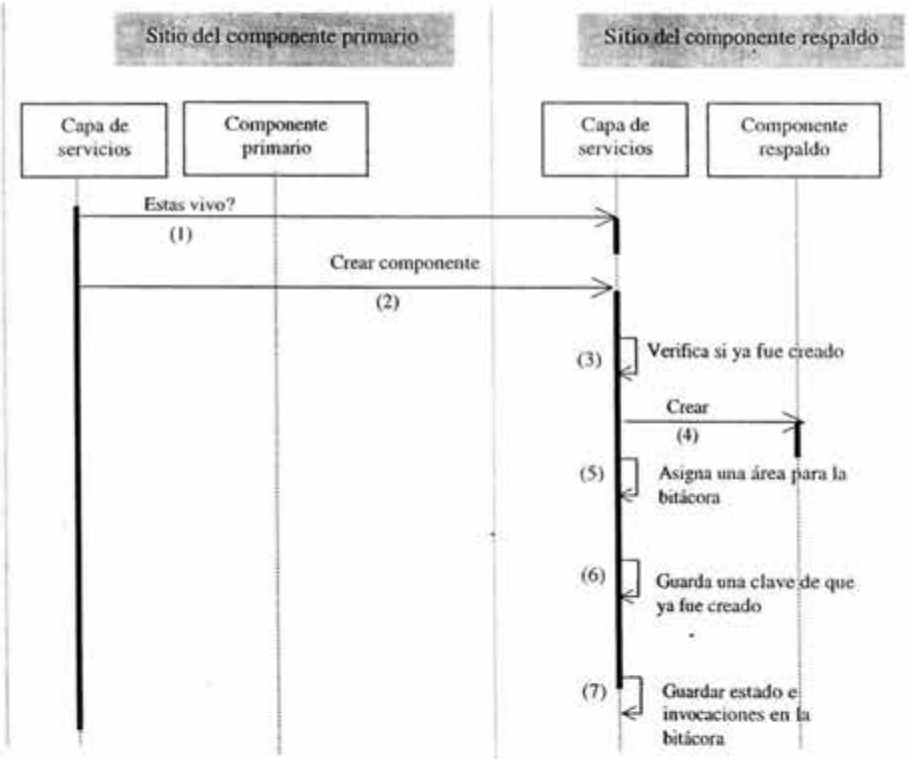
(1) La capa de servicios revisa en al bitácora (que está en memoria persistente) qué componentes estaban en funcionamiento hasta antes de la falla

(2) Hace una invocación al sitio del componente principal para que le envíe el estado que tienen en la bitácora junto con las invocaciones que no han sido ejecutadas

(3) Verifica que el componente no haya sido eliminado. De ser así, se aborta el algoritmo y no se crea el componente, además de que se borra la clave del componente de la bitácora
(4) Crea nuevamente cada uno de los componentes que estaban en funcionamiento
(5) Asigna un área para la bitácora del componente
(6) Guarda la clave de creación del componente en la bitácora que se encuentra en memoria persistente
(7) Guarda en su bitácora el estado y las invocaciones entregadas en (2) por el sitio del componente primario, En el futuro el componente primario comienza a enviarle los cambios de estado y las copia de las invocaciones que reciba de los clientes, según el algoritmo de la Fig. 4-3.

Fig. 4-10 Reincorporación inicializada en el sitio que falló

A través de la petición del sitio del componente primario



Donde:

(1) Después de un tiempo previamente determinado y de estar trabajando sin respaldo, la capa de servicios del sitio el componente primario envía una invocación de prueba sólo para saber si el sitio ya regresó de la falla
(2) Si la prueba anterior resulta exitosa envía una invocación para crear el componente respaldo, esta invocación lleva como parámetros el estado y las invocaciones que no han sido ejecutadas en el sitio del componente primario

(3) La capa de servicios del sitio que regresa de la falla verifica si ya fue ejecutada anteriormente la invocación de creación, si ya fue ejecutada saltamos al paso(7)
(4) La capa de servicios crea el componente
(5) Asigna un área en memoria primaria para la bitácora del componente creado
(6) Guarda la clave del componente en memoria persistente
(7) Guarda el estado y las invocaciones en la bitácora después de esto el componente esta listo para tomar el lugar de componente respaldo y envía su referencia al sitio del componente primario

Fig. 4-11 Reincorporación inicializada en el componente primario

Este mecanismo de recuperación se implementa debido a que existe la posibilidad de que el sitio del componente respaldo haya fallado antes de que la clave de los componentes sea guardada en la bitácora y no sea posible recuperar el componente con el algoritmo de la Fig. 4-10. Por tanto se siguen los pasos de la Fig. 4-11.

Notemos que esta forma de reincorporación sólo se usará cuando el componente falla durante el algoritmo de la invocación del método crear de la Fig. 4-3 y explícitamente la falla debe de ocurrir en el componente primario entre los paso (2) al (7) o que el componente respaldo falle entre los pasos (1) y (11). En otros casos la forma de reincorporación será a través de la búsqueda en la bitácora de la memoria persistente. Puede darse el caso de que las dos formas de recuperación se den de manera simultánea y se presentará cuando en el algoritmo de creación el componente primario falla entre los pasos (6) y (7). En ese caso la primera forma de recuperación que inicie desactiva a la otra.

5 VALIDACION

En este estudio se utilizaron dos especificaciones para hacer la validación de la propuesta de agregar tolerancia a fallas a los middlewares de componentes especificada en el capítulo 4, en primer lugar se usó la especificación de EJB [10] de la cual se obtuvo la especificación EJBTaF que se puede consultar en el anexo de este documento pero de la cual no se hizo alguna implementación por no tener acceso a un código fuente de alguna implementación de EJB. En segundo lugar se usó la especificacion de Séneca de la cual se obtuvo SénecaTaF (sección 5.1), y utilizando su implementación Séneca-j (sección 5.2) se obtiene el prototipo SénecaTaF-j (sección 5.3).

5.1 Modelo SénecaTaF

Séneca al igual que los otros middleware a componentes estudiados en el capítulo 3, no tiene contemplado el servicio de tolerancia a fallas dentro de los servicios middleware, por tal razón, se proponen cambios al modelo de tal forma que Séneca pueda soportar réplicas de componentes con un grado de transparencia N3, ver sección 4.1, donde los desarrolladores de las aplicaciones distribuidas tiene la responsabilidad (y la oportunidad) de elegir qué componentes serán tolerantes a fallas y cuales no. Dando paso así al modelo SénecaTaF.

Los cambios hechos en los modelos de Séneca para dar paso al modelo SénecaTaF son:

1. **Modelo abstracto**. Sin cambios, sigue igual que en modelo original.

2. **Modelo de programación**. Sigue la especificación del modelo original excepto que se restringe a utilizar componentes que sean compartidos (shared) con o sin estado, dado que para este tipo de componentes el contenedor sólo hace una instancia de él y los clientes se turnan su hilo de control para ejecutar sus invocaciones. Lo anterior se debe a que existe la restricción en el modelo abstracto para que en caso de que lleguen dos o mas invocaciones de distintos clientes al mismo componte, éstas deban de ser serializadas, por la capa de servicios, y pasadas una a la vez para que sean ejecutadas por el componente.

   En esta propuesta de modelo, se permite elegir al desarrollador de la aplicación agrega o no, la característica de tolerante a fallas a cada componente[1] (además de las características de compartido o persistente), esto mediante la incorporación de la cláusula:
   - **toleranceFault** si se quiere que el componente tenga tolerancia a fallas (1-tolerante) del tipo:, Caída-pausa y falla de omisión
   - **untoleranceFault** si no se quiere que tenga tolerancia a fallas

   Ejemplo:

```
component CAritmetica {
    unshared
    not persistent
    toleranceFault //componente tolerante a fallas
    IStateAritmetica implementedBy AritmeticaStateImpl; // // Esta interfaz esta
                // dada por omisión ya que el componente está definido con estado
    ISuma implementedBy SumaImpl;
    IResta implementedBy RestaImpl;
    }
```

[1] Se deja la oportunidad al desarrollador del sistema distribuido de decidir que componentes son necesarios en una operación critica según su criterio y experiencia.

Se deja abierta la posibilidad, en futuras versiones, de poder escoger la cantidad de réplicas (n-tolerante) y la forma en que serán manejadas (activa o pasivamente).

3. **Modelo de ensamblado y desplegado**. En este modelo no habrá cambios, y en cuanto a las réplicas, la localización y su conexión se harán de manera automática y transparente para el desarrollador de la aplicación.
4. **Modelo de ejecución**. Este modelo seguirá la definición que se da en el modelo original, excepto que el contenedor deberá de manejar la parte del trabajo de tolerancia a fallas que le corresponda, según las características de los componentes dadas en el modelo de programación e incluso el manejo de tolerancia a fallas.

Además, en este modelo habrá cambio en cuanto a la instanciación de un componente, ya que del lado del cliente debe de haber una clase que haga el trabajo de ocultar la distribución del sistema y la tolerancia fallas que le corresponda (si es que al componente se le agregó esta característica). De esta forma, ahora para obtener una referencia de un componente, se hará de la misma manera que si el componente estuviera en el mismo sitio que el cliente. Ejemplo:

```
CDisplay display = new CDisplay ();
display.print("Hello World");
```

5.2.1 Herencia de SénecaTaF

Después de agregar a Séneca tolerancia a fallas, SénecaTaF es una especificación que agrega tolerancia a fallas al modelo Séneca. El modelo SénecaTaF es un modelo para crear middleware a componentes con las características que se mencionan en el resto de esta sección.

Tipo de modelo

SénecaTaF sigue un modelo a componentes, el cual describe la creación, identidad, búsqueda y operación de un componente en un ambiente distribuido.

Descripción de componentes

Los componentes son software que representa la lógica de negocio que se ejecutan en un entorno llamado "contenedor" el cual los aísla del mundo exterior.

Separación entre interfaces e implementación

Los componentes en SénecaTaF sólo son accedidos por los clientes haciendo uso de sus interfaces y nunca de manera directa.

Servicios

Los servicios[1] entregados por SénecaTaF de manera automática a los programadores de componentes son: ocultamiento de los detalles de la red, ciclo de vida del componente, seguridad, persistencia, transacción y ocultamiento de los detalles tolerancia a fallas. Además, el programador cuenta con las herramientas para empaquetar y desplegar componentes.

[1] Algunos de estos servicios pueden ser dejados como responsabilidad del programador haciendo uso de las herramientas e interfaces adecuadas.

SénecaTAF			
Tipo de middleware			A componentes
Plataforma			Multiplataformas
Complejidad para construir grandes aplicaciones			Media
Optimizar, agregar, o eliminar servicios			No es permitido
Forma en que los objetos o componentes obtienen los servicios			De manera automática
Tolerancia a fallas	Tipo de fallas		Caída-amnésica, Caída-parcial-amnésica, Caída-pausa y falla de omisión
	n-tolerante		n =1
	Mecanismos	monitoreo	Pull y Push
		redundancia	Pasiva
		Recuperación	Cuando un componente falla su estado se recupera de una bitácora

Tabla 5-1 SénecaTaF

Tolerancia a fallas

- **Interfaces**. Para lograr la tolerancia a fallas, se propone dos clases y modificar el comportamiento de otras. En general se propone que en el modelo SénecaTaF existan las interfaces para: el manejo de mensajes, el manejo de bitácora, el manejo de réplicas y mecanismos de recuperación.

- **Mecanismos.** SénecaTaF usa los siguiente mecanismos para lograr la tolerancia a fallas:

  - Replicación pasiva (1-tolerante)
  - Bitácoras y puntos de verificación
  - Detección y notificación de fallas
  - Manejo de consistencia

  SénecaTaF tiene las siguientes características:

  - Transparencia de replicación de nivel N3 (ver sección 4.1)
  - Transparencia de fallas

- **Tipo de fallas soportadas.** SénecaTaF pretende tolerar las fallas (1-tolerante) de tipo Caída-pausa y falla de omisión, y dejando la posibilidad de que la especificación pueda evolucionar sin cambios drásticos para ser n-tolerante, donde $n \geq 2$.

Además, se incorpora el mecanismo (no provisto en las especificaciones estudiadas) de monitoreo Push, con el cual, el componente que sufrió la falla inicia su recuperación desde el mismo sitio.

La tabla 5-1 resume las características generales de SénecaTaF.

5.2 Séneca-j

Séneca-j es un middleware que implementa (parcialmente) al modelo Séneca. Séneca-j versión 1.13 es una implementación basada en la tecnología Java, la cual tiene implementada las siguientes características:

- Un compilador de lenguaje de definición del modelo abstracto.
- Un compilador para el lenguaje de definición del modelo de programación.
- Un compilador para el lenguaje de definición del modelo de ensamblado y desplegado.
- Un generador de código de componentes para la descripción del modelo abstracto.
- Una herramienta de desplegado distribuido que instancia componentes de acuerdo a la descripción dada en el modelo de ensamblado y desplegado.

En Séneca-j cada componente es una clase Java que mantiene la relación entre las clases que implementan las interfaces.

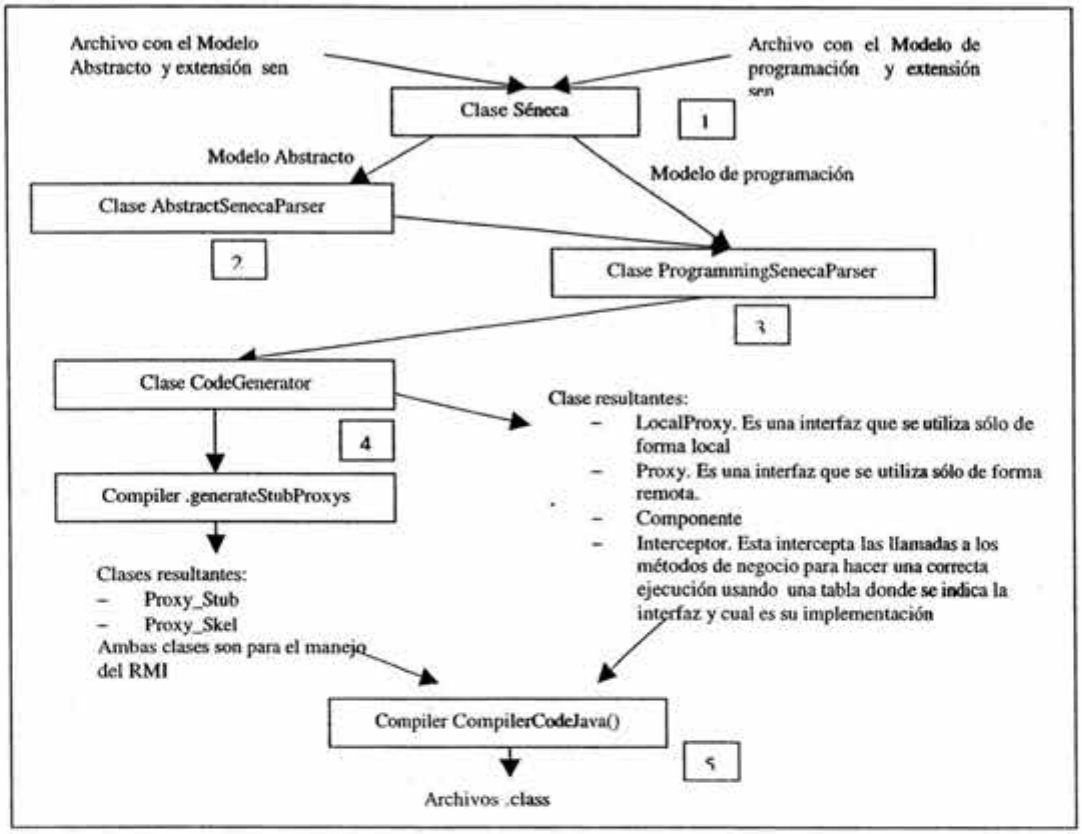
En Séneca-j los submodelos de Séneca[1]; modelo abstracto, modelo de programación y modelo ensamblado y desplegado, son creados en archivos separados con extensión **.sen**, además de que cada una de las interfaces de los componentes deben de ser implementadas en archivo distintos con extensión .java.

En los subsecciones siguientes mostraremos la forma en que Séneca-j crea los componentes y la manera en que los expone a sus clientes a través de sus distintos modelos.

Modelo abstracto y modelo de programación

Para que Séneca-j pueda crear los componentes a través de los archivos de los modelos abstractos y de programación (extensión .sen), implementa la clase Séneca (que administra a los compiladores), la clase AbstractSénecaParser (compilador del lenguaje de definición del modelo abstracto), la clases ProgrammingSenecaParser (compilador del lenguaje de definición del modelo de programación), la clase CodeGenerator (genera archivos con código fuente escritos en lenguaje Java), y la clase Compiler (genera clases java). En la Fig. 5-1 se muestra la forma en que éstas clases están relacionadas, en función de quién produce y de quién necesita lo producido, para lograr la creación del componente:

[1] En esta parte no se menciona, aunque si se trata en este capítulo, al modelo de ejecución ya que este es una característica interna del middleware, y no debe de ser especificado por el desarrollador de la aplicación distribuida.



Donde:

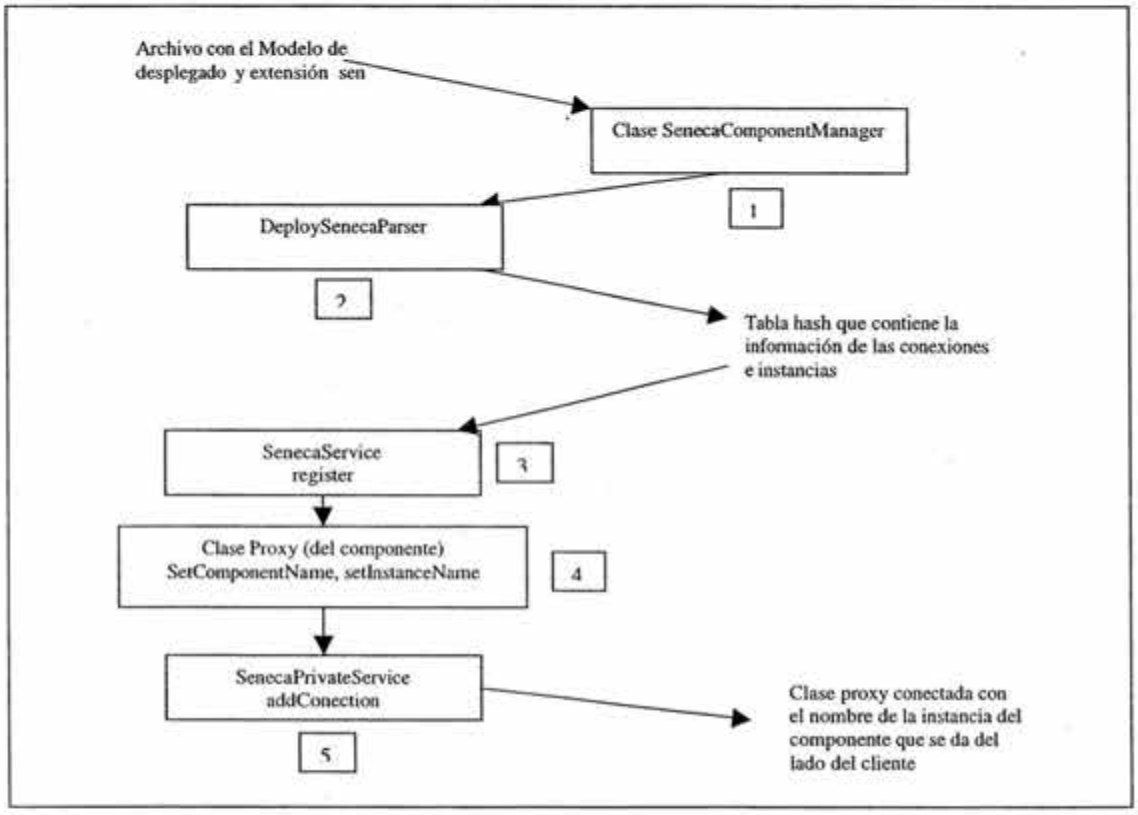
1. La clase Séneca recibe los archivos (con extensión sen) que contienen las especificaciones del modelo abstracto y modelo de programación (archivos separados)
2. Séneca envía al archivo del modelo abstracto al compilador del lenguaje de definición del modelo abstracto (clase AbstractSenecaParser), el cual, crea una estructura de tipo tabla hash que contiene toda la información del modelo.
3. Séneca envía la tabla hash anterior y al archivo del modelo de programación al compilador del lenguaje de definición del modelo de programación (clases ProgrammingSenecaParser), el cual, crea otra tabla hash que combina la información de los dos modelos.
4. Una vez obtenida la tabla anterior, la clase Séneca la envía a la clase generadora de código (clase CodeGenerator), la cual tiene dos tareas; la primera es crear cinco archivos fuentes que permiten (del lado del cliente) dar la impresión de que la entidad componente es sólo un elemento y no la unión de clases separadas que implementan sus interfaces, la segunda tarea es enviar la tabla hash al compilador (clase Compilator) para que genere los archivos fuentes Stub y Skeletor del componente.
5. Séneca envía los archivos fuentes (.java) al compilador para que este cree las clases Java.

Fig. 5-1 Modelos abstracto y de programación en Séneca-j

Modelo de ensamblado y desplegado

En cuanto al ensamblado y desplegado se refiere, Séneca-j implementa la clase SenecaComponentManager (que se encarga de la administración de los servicios de los componentes), la clase DeploySenecaParser (compilador del lenguaje de definición del modelo de

ensamblado y desplegado), la clase SenecaService (que se encarga de los servicios), la clase Proxy creada en el modelo de programación, la clase SenecaPrivateService (que se encarga de la conexión entre el cliente el componente). En la Fig. 5-2 se muestra la forma en que estas clases están relacionadas, en función de quien produce y de quién necesita lo producido, para lograr el ensamblado y la exposición de los componentes a los clientes.



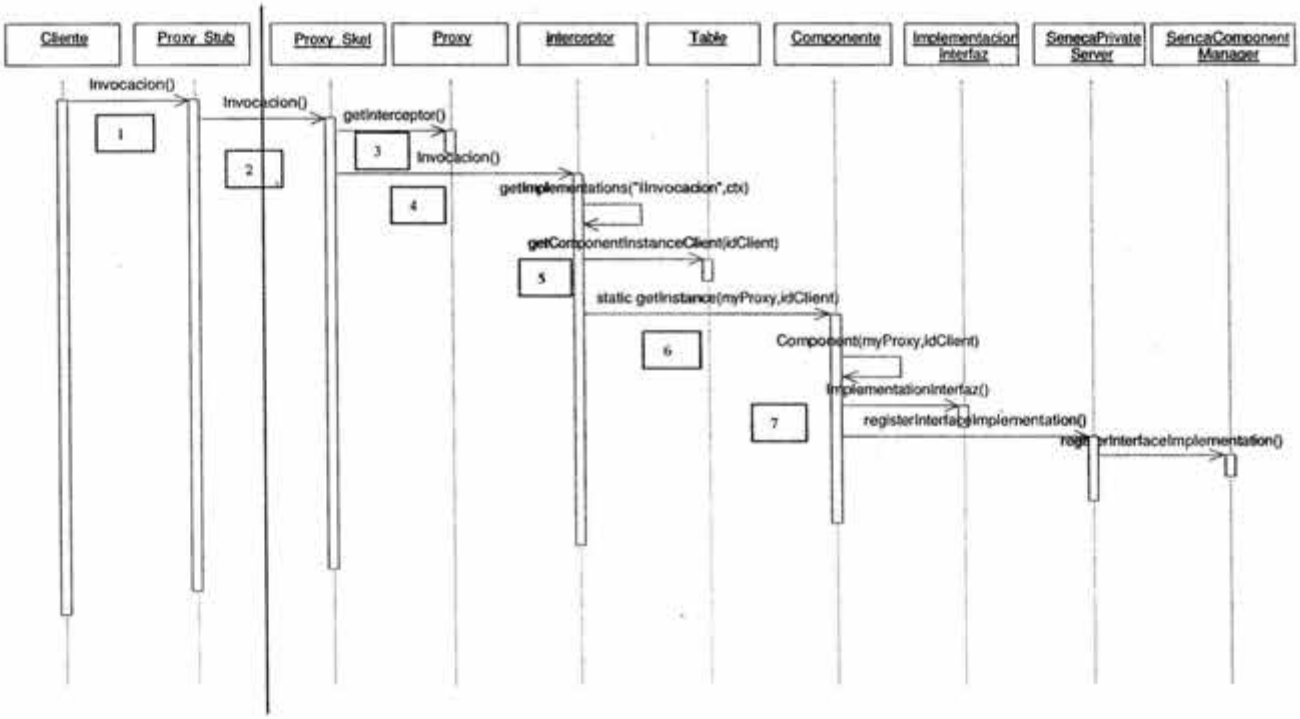
Donde:

1. La clase SenecaComponentManager recibe al archivo que contiene al modelo de ensamblado y desplegado y lo entrega al compilador.
2. El compilador DeploySenecaParser recibe el archivo y crea una tabla hash que contiene la información sobre la instanciación y la conexión de los componentes definidos en los dos anteriores modelos.
3. La tabla hash es recibida por la clase SenecaServices quien hace el registro del lugar dónde se encontrarán las instancias.
4. La clase Proxy muestra el nombre de la clase del componente al que está ligado.
5. La clase SenecaPrivateService hace la conexión entre el componente y el nombre que se le da a la instancia del lado del cliente.

Fig. 5-2 Modelo de desplegado y ensamblado

Modelo de Ejecución

La ejecución de la invocación, que hace un cliente a un componente, se realiza utilizando la interfaz adecuada y su implementación, para asegurar esto Séneca-j hace interactuar sus principales clases como se muestra en la Fig. 5-3.



Donde:

1. Después de encontrar la implementación de la interfaz, a través del método **lookup()** de la clase **Context**, el cliente hace la invocación al método a través del **stub.**
2. La invocación se transmite, de forma transparente, al sitio del componente .
3. Se pide a la clase **Proxy** busque a la clase **Inteceptor** que ha de mantener la asociación entre el cliente y el componente.
4. Una vez que se ha encontrado el **Interceptor** se le entrega la invocación.
5. Quien la registra en **table** donde se guarda el identificador del cliente que será después extraído para regresar el resultado.
6. Después de registrar al cliente, el **Interceptor** pide una instancia del **componente** y envía la invocación.
7. El componente pide una instancia de la **implementación de la interfaz** invocada y la registra en las clases **SenecaPrivateServer** y **SenecaComponentManager** quienes administran los servicios del componente.

Fig. 5-3 Modelo de ejecución (clases principales)

5.3 SénecaTaF-j

Para mostrar como puede ser implementada la especificación SénecaTaF, usaremos la implementación de Séneca llamada Séneca-j y el modelo abstracto (propuesto en capítulo 4), dando paso así al prototipo SénecaTaF-j (Séneca tolerante a fallas -j).

Para lograr SénecaTaF-j la implementación Séneca-j tendrá que ser cambiada, y ese trabajo será dividido en las siguientes dos tareas:

(a) Hacerle cambios para que soporte la nueva cláusula **toleranceFault,** única cláusula que esta en SénecaTaF y que no se encuentra en Séneca
(b) Hacerle cambios para que pueda asegurar la consistencia de las réplicas de los componentes (en el caso de que existan), así como hacer cambios, en las clases existentes, para crear las nuevas clases que ayuden en la transparencia de la replicación de componentes que será dada a los clientes.

5.3.1 Soporte de la cláusula toleranceFault

Para poder soportar la cláusula **toleranceFault** (así como su semántica) se debe cambiar el compilador de lenguaje de definición del modelo de programación, así como el compilador del lenguaje de definición del modelo de ensamblado y desplegado. De tal forma que SénecaTaF-j pueda crear y desplegar de manera transparente y automática las réplicas de los componentes.

5.3.2 Soporte para consistencia de réplicas

Para cumplir con la parte (b) nos auxiliaremos de la especificación de tolerancia a fallas que se definió en el capítulo 4. Para lo cual se requiere que:
- La infraestructura en la que está montada Séneca-j cumpla con algunos prerrequisitos.
- Sean asignadas las nuevas responsabilidades que deben de tener los actores Séneca-j para poder tolerar fallas.
- Agregar en Séneca-j los elementos (o actores) que no se encuentran definidos en esta implementación y que son necesarios para tolerar las fallas.

Prerrequisitos

Los prerrequisitos que debe de tener el modelo Séneca para que pueda ser implementado el modelo abstracto y que no necesariamente deben de estar implícitos en todos los sistemas son:

1. Supondremos un canal de comunicación confiable, unidireccional y *FIFO* (primero que entra primero que sale, por sus siglas en inglés) que une a los sitios.
2. No existe un reloj común y es necesario el uso de mensajes para poder sincronizar los sitios.
3. Cuando un componente falla, éste detiene su funcionamiento evitando fallas del tipo bizantinas y no permitiendo se corrompan los datos de la memoria estable.
4. Supondremos también que los sitios no tienen una carga excesiva y siempre que no hayan sufrido una falla podrán contestar en el tiempo predeterminado para ello.

Nuevas responsabilidades

Las nuevas responsabilidades que deben de tener los actores en la arquitectura de Séneca-j, la cual es heredada de Séneca, que son necesarias para poder tolerar fallas propuestas en la sección 4.1. Así de esta forma, se tiene las siguientes nuevas responsabilidades:

El cliente. no tendrá nuevas responsabilidades

Middleware Séneca. Seguirá ofreciendo todos los servicios que ya proporcionaba, además de realizar las siguientes tareas:

- Stub del componente

  a) La primera vez que haga una invocación a un componente lo hará al componente primario.
  b) Espera un tiempo determinado (timeout) después de hacer la invocación, si antes del término de este tiempo, el sitio del componente no responde, se tomará esa acción como una falla del sitio.
  c) Al darse cuenta de que el componente primario ha fallado, reenviará la invocación pero en esta ocasión lo hará al componente réplica, siguiendo los pasos de la Fig. 4-6.

- El contenedor en el sitio de ambos componentes (primario y réplicas)

  d) Hará la administración de la *bitácora* asignada a cada componente.
  e) Maneja los mensajes para la consistencia entre ambos componentes.
  f) Cada vez que un componente sea creado guardará en un área de memoria estable una clave que indique qué componente fue creado.
  g) Cuando regrese de una falla buscará en memoria estable qué componentes eran utilizados y los creará nuevamente, después enviará un mensaje al componente principal para que sepa que ya regresó de la falla y sea actualizado para que pueda ser integrado nuevamente al sistema tomando el lugar de componente respaldo.
  h) Cada vez que un componente sea eliminado, en la memoria estable se eliminará la clave de este.
  i) Verificará con la ayuda de la bitácora si el componente ya ejecutó cada una de las invocaciones que arriban al sitio. Si una invocación ya fue ejecutada por el componente no será pasada nuevamente a él para que la reejecute, en ese caso, se obtendrá el resultado de la bitácora y se envía al cliente.

- El contenedor en el sitio del componente primario

  j) Cuando haga una invocación al sitio del componente respaldo, esperará la respuesta un tiempo determinado (timeout), si antes del término de este tiempo, el sitio del componente respaldo no responde, se tomará esa acción como una falla del sitio.
  k) Cuando detecte que el sitio del componente respaldo ha fallado, dejará de enviarle las invocaciones que usaba para mantener la consistencia entre ambos, y las enviará nuevamente hasta que el sitio del componente respaldo haya regresado de la falla y esté listo

- El contenedor en el sitio del componente respaldo

l) Cuando detecte que las invocaciones son hechas por el cliente y no por el componente primario, entonces sabrá que el componente primario ha fallado y manejará la falla (según el algoritmo de la Fig.4-6)

componente. Seguirá trabajando según su especificación.

Nuevos actores

Para poder soportar las nuevas responsabilidades de los actores, dadas en la sección anterior, se propone agregar los siguientes actores a Séneca-j:

- **Manejador de Tolerancia.** Este nuevo actor será el encargado de replicar la información en el sitio del componente respaldo. También serán el encargado (en el sitio del servidor) de la detección, manejo y recuperación de los componentes que presenten una falla.
- **Manejador de Bitácora.** Este es un nuevo actor (en el sitio del servidor) es auxiliar de la clase anterior, el cual será encargado de la asignación, creación, eliminación, modificación y administración de la bitácora (tanto en memoria primaria como memoria persistente) que requiere el Manejador de Tolerancia para cumplir su objetivo.
- **Manejador de tolerancia del cliente**. Debido a que las clases stub, que se encuentran en el sitio del cliente, son las encargadas de entregar al cliente la referencia de los componentes que se encuentran en el servidor, se propone la creación de un actor del lado del cliente (con el nombre del componente), que por un lado oculte la red, y que por otro lado, en caso de que el componente sea definido como tolerante a fallas, este actor la haga transparente al cliente. Además de que en él recaerá la responsabilidad agregada a los stub, permitiendo así que los stubs realicen sólo el trabajo para el cual fueron pensados originalmente.

5.3.3 Cambio de Séneca-j a SénecaTaF-j

En esta sección se muestran como los algoritmos especificados en la sección 4.5.1 son usados para hacer cambios en las clases de Séneca-j para dar soporte a la cláusula **tolerenceFault.** Así también se muestran las nuevas clases que se agregaron a SénecaTaF-j para dar soporte a la consistencia de las réplicas, como es especificado en la sección 5.3.2.

Cambios en compiladores

Para implementar SénecaTaF-j se hacen cambios en los compiladores de Séneca-j, según lo especificado en la sección 5.3.1, estos cambios se traducen en cambios en las siguientes clases:

- Clase ProgrammingSénecaParser (compilador de lenguaje de definición del modelo de programación). Esta clase además de hacer el trabajo que hacia en Séneca, ahora también al detectar la característica **toleranceFault** en algún componente será la responsable de crear los elementos para que otro componente idéntico, excepto por el nombre que al final se le agregará el número 2, también pueda ser creado. Ejemplo: CDisplay, CDisplay2 donde los componentes que tienen el número 2 al final serán los componentes respaldo al inicializar el sistema.
- Clase DeploySénecaParser. La clase además de crear los elementos necesarios para que sean desplegados y puedan ser instanciados los componentes en sus sitios correspondientes como es definido en el modelo de desplegado y ensamblado, como lo hacia anteriormente, ahora también será la responsable de crear la información necesaria para que las réplicas de los

componentes puedan ser desplegados e instanciados en sitios distintos a los sitios donde se encuentran los componentes primarios, esto de manera transparente al cliente.

Integración de los nuevos actores

Para hacer la integración de los nuevos actores en Séneca-j, según se especifica en la sección 5.3.2, se requiere que estos sean implementados por clases Java, para esto se propone que:

- El actor Manejador de tolerancia y el actor Manejador de bitácora sean implementados por: la clase **ManejadorDeTolerancia**, la cual implementa la **interfaz ManejadorDeTolerancia** y la clase **ManejadorDeBitacora** la cual implementa la **interfaz ManejadorDeBitacora** respectivamente, y que están definidas en *Especificación de nuevas clases* de esta sección. El lugar que tomarán las clases anteriores entre las clases ya existentes en Séneca, es mostrado en la Fig. 5-4.

Fig. 5-4 Parte del Contenedor que contempla tolerancia a fallas

- Manejador de tolerancia del cliente, este actor será implementado por una clase que tenga el nombre del componente, según como fue definido en al modelo abstracto, y que a través de sus métodos, que tendrán el nombre de las interfaces que provee el componente, el cliente pueda invocar la funcionalidad del componente como si ambos se encontraran en el mismo sitio. Además, esta clase manejará la tolerancia a fallas (si es que el componente es tolerante a fallas) haciendo los cambios de conexión entre el cliente y el componente primario, y el cliente y el componente réplica, según se detecte una falla en uno u otro, lo anterior de manera transparente para el cliente. Su ubicación entre las clases ya existentes en Séneca es mostrada en la Fig. 5-5.

La integración y la forma en que interactúan las clases anteriores con las clases de Séneca-j se muestran en la sección *Cambios a los diagramas del modelo abstracto.*

Simplificación de diagramas

Antes de mostrar como los algoritmos del capítulo 4 son ajustado y con la intención de evitar una saturación de información en los diagramas de SénecaTaF-j, las clases y sus mensajes que no aportan tolerancia a fallas son agrupadas (esto es sólo un efecto visual ya que las clases y los mensajes entre ellas seguirán existiendo) tomando como referencia las clases mostradas en la Fig. 5-3. De esta manera las agrupaciones son las siguientes:

- Las dos clases Stub y Stub2[1] que se encuentran del lado del sitio del cliente y que son parte de Séneca-j, se agrupan con la clase cliente para formar una clase llamada Clase/Stub.
- Las clases Proxy_skel y Proxy que se encuentran del lado del sitio del componente primario formarán una clase llamada Proxy_skel/Proxy y para las que se encuentran del lado del sitio del componente réplica se les agregará un 2 al final, o sea Proxy_Skel/Proxy2.
- Las clases Interceptor y Table que se encuentran del lado del sitio el componente primario formarán la clase llamada Interceptor/Table y para las que se encuentran del lado del sitio del componente réplica se llamará Interceptor/Table2.

Y las clases ImplementacionInterfaz, SénecaPrivateServices y SénecaComponentManager no aparecerán en los subsecuentes diagramas (aún cuando también sean parte de SénecaTaF-j) ya que no toman un papel determinante en el manejo de la tolerancia a fallas, ya que el manejo de las mismas se propone se haga antes de que la invocación llegue al componente.

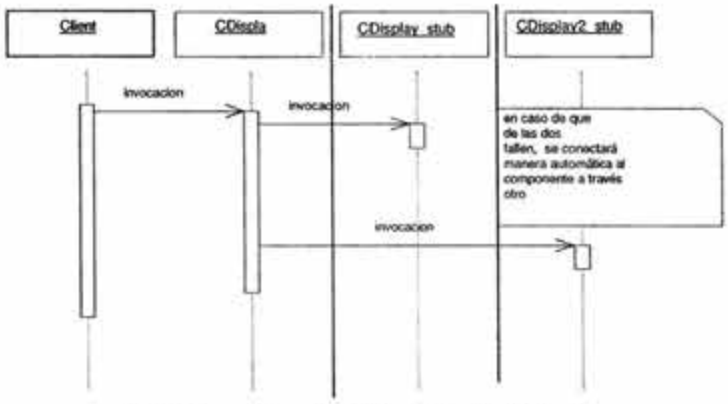


fig. 5-5 clase tolerante a fallas del lado del cliente

Ajuste de algoritmos

Para lograr que las nuevas clases (o nuevos actores) que permiten la tolerancia a fallas puedan interactuar con las clases ya existentes en Séneca-j, los diagramas propuestos (en el capítulo 4) que representan los algoritmos de tolerancia a fallas (solución a las nuevas responsabilidades) son ajustados para que formen parte de SénecaTaF-j. Estos ajustes son mostrados en la Tabla 5-2.

Caso	Acción
Algoritmo en la invocación de un método (vida)	
Algoritmo para un solo cliente	
Funcionamiento básico (libre de fallas)	Los cambios que sufre el diagrama de la Fig. 4-5 son mostrados en la Fig. 5-6.
Invocación con falla de alguno de los Participantes	
Falla del componente primario	Los cambios que sufre el diagrama de la Fig. 4-6 son mostrados en la Fig. 5-7. La reincorporación del componente se hará *a través de la petición del sitio*.

[1] Las clases Stub y Stub2 son las clases, del lado del cliente, que hacen la conexión con el componente primario y el componente respaldo respectivamente al inicial el sistema.

	del componente primario, (véase Fig. 5-12).
Falla del sitio del componente réplica	Cuando se presente este caso de falla en el modelo Séneca, el Manejador de Tolerancia que se encuentra en el componente primario deja de enviar las invocaciones el sitio del componente réplica hasta que el componente réplica regrese de la falla.
Falla del sitio del cliente	Cuando el sitio del cliente falle, el cliente réplica tomara su lugar si es que el cliente tuviera uno, de no ser así, se producirá una falla visible al usuario.
Algoritmo para más de un cliente a la vez	
Funcionamiento básico (libre de fallas)	Los cambios que sufre el diagrama de la Fig. 4-7 son mostrados en la Fig. 5-8.
Invocación con falla de alguno de los Participantes	
Falla del sitio componente primario	Los cambios que sufre el diagrama de la Fig. 4-8 son mostrados en la Fig. 5-9.
Falla del sitio del componente réplica	Las mismas acciones a seguir que en el caso de la invocación para un solo cliente a la vez.
Falla del sitio del cliente	Las mismas acciones a seguir que en el caso de la invocación para un solo cliente a la vez
Eliminación del componente (Muerte)	
Funcionamiento básico normal (libre de fallas)	Los cambios que sufre el diagrama de la Fig. 4-8 son mostrados en la Fig. 5-10.
Invocación con falla de uno de los participantes	
Falla del componente primario	En este caso el Stub al detectar la falla, envía la invocación al sitio del componente réplica para que sea eliminado. El componente primario con la falla de su sitio fue eliminado parcialmente quedando en la bitácora de la memoria persistente la clave del componente, pero la cual será eliminada cuando intente reincorporarse, véase paso (3) de la Fig. 5-10.
Falla del sitio del componente réplica	En este caso el Manejador de tolerancia del sitio del componente primario al detectar la falla la ignorará y sigue la parte el algoritmo que le corresponde en su sitio, ya que igual que en el caso anterior, el componente respaldo con la falla de su sitio fue eliminado parcialmente quedando en la bitácora de la memoria persistente la clave del componente, pero la cual será eliminada cuando intente reincorporarse, véase paso (3) de la Fig. 5-10.
Falla del sitio del cliente	Como ya lo hemos mencionado, al cliente se le puede modelar como un componente, que tendrá una réplica y el cual tomará su lugar en caso de que falle este, y reenviará la invocación de eliminación al componente primario.
Reincorporación de un componente cuando regresa de una falla (Resurrección)	
A través de la búsqueda en la bitácora de la memoria persistente	Los cambios que sufre el diagrama de la Fig. 4-10 son mostrados en la Fig. 5-11.
A través de la petición del sitio del componente primario	Los cambios que sufre el diagrama de la Fig. 4-11 son mostrados en la Fig. 5-12.

Tabla 5-2 Diagramas de algoritmos para la tolerancia a fallas

La invocación del método para la creación no está contemplada en la tabla 5-2 dado que durante el ensamblado y desplegado de una aplicación en Séneca-j se realiza la creación de los componentes siguiendo los pasos de la fig. 5-1. Pues bien, para SenecaTaF-j seguiremos los mismos pasos pero entre los pasos 3 y 4 donde se crea, entre otras cosas, la clase Interceptor del componente es ahí donde pediremos que se creen las clases ManejadorDeTolerancia y ManejadorDeBitacora así como las clases bitácora persistente y de memoria RAM, las cuales serán instanciadas sólo cuando los componentes sean utilizados por primera vez según el algoritmo de la fig. 4-4 omitiendo el paso 4.

Cambios en los diagramas del modelo abstracto

En esta sección son mostrados los cambios que sufrieron los diagramas de los algoritmos que componen al modelo abstracto, los cambios son debidos a la incorporación de las dos nuevas clase, Manejador de Tolerancia y Manejador de Bitácora, y al ajustar estos diagramas a SénecaTaF-j.

En los siguientes diagramas los mensajes que no aparecen en el modelo abstracto, pero que son necesarios para ajustarlos a SénecaTaF, son etiquetados con una letra de en vez de un número como en el modelo abstracto.

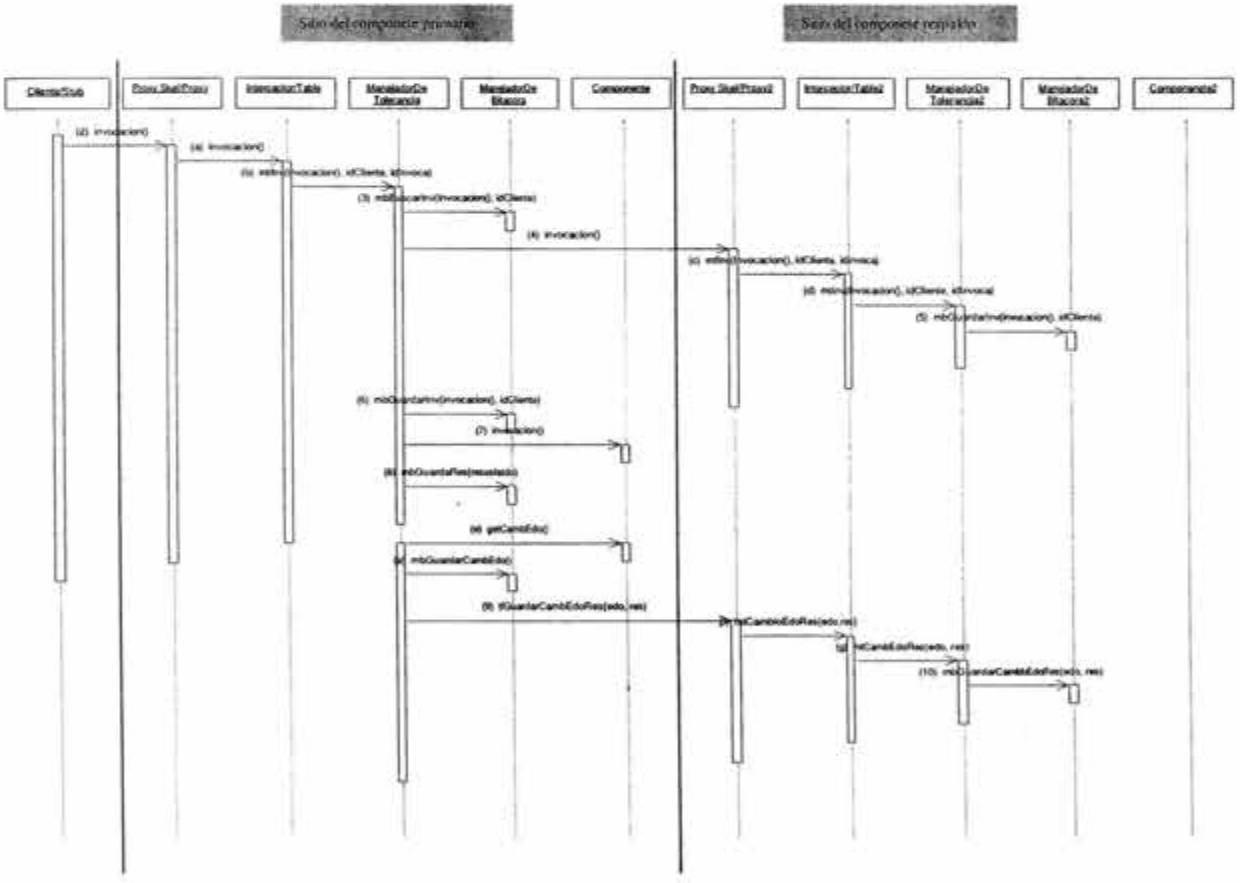


Fig. 5-6 Invocación

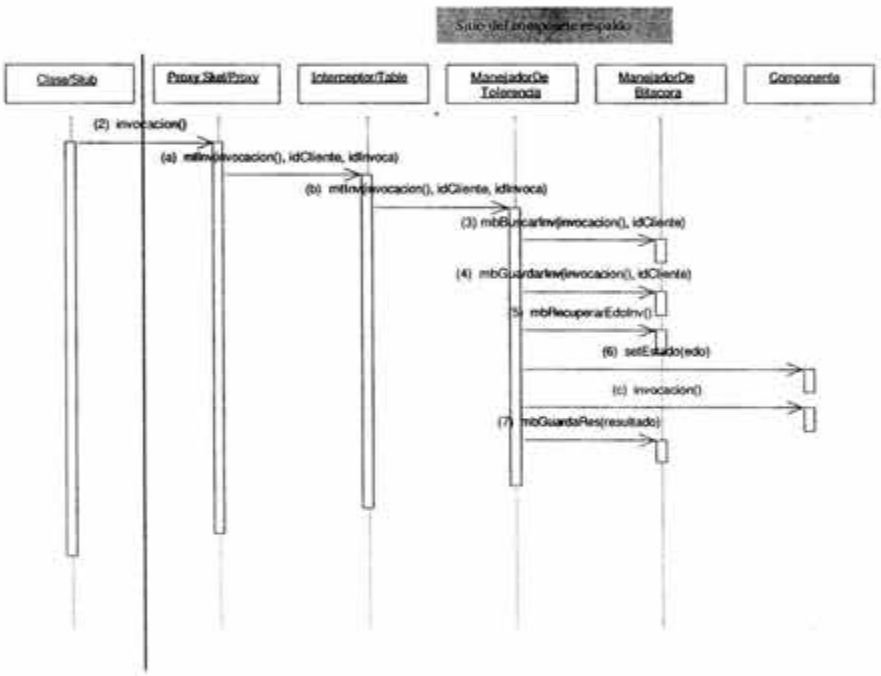


Fig. 5-7 Invocación con falla del componente primario

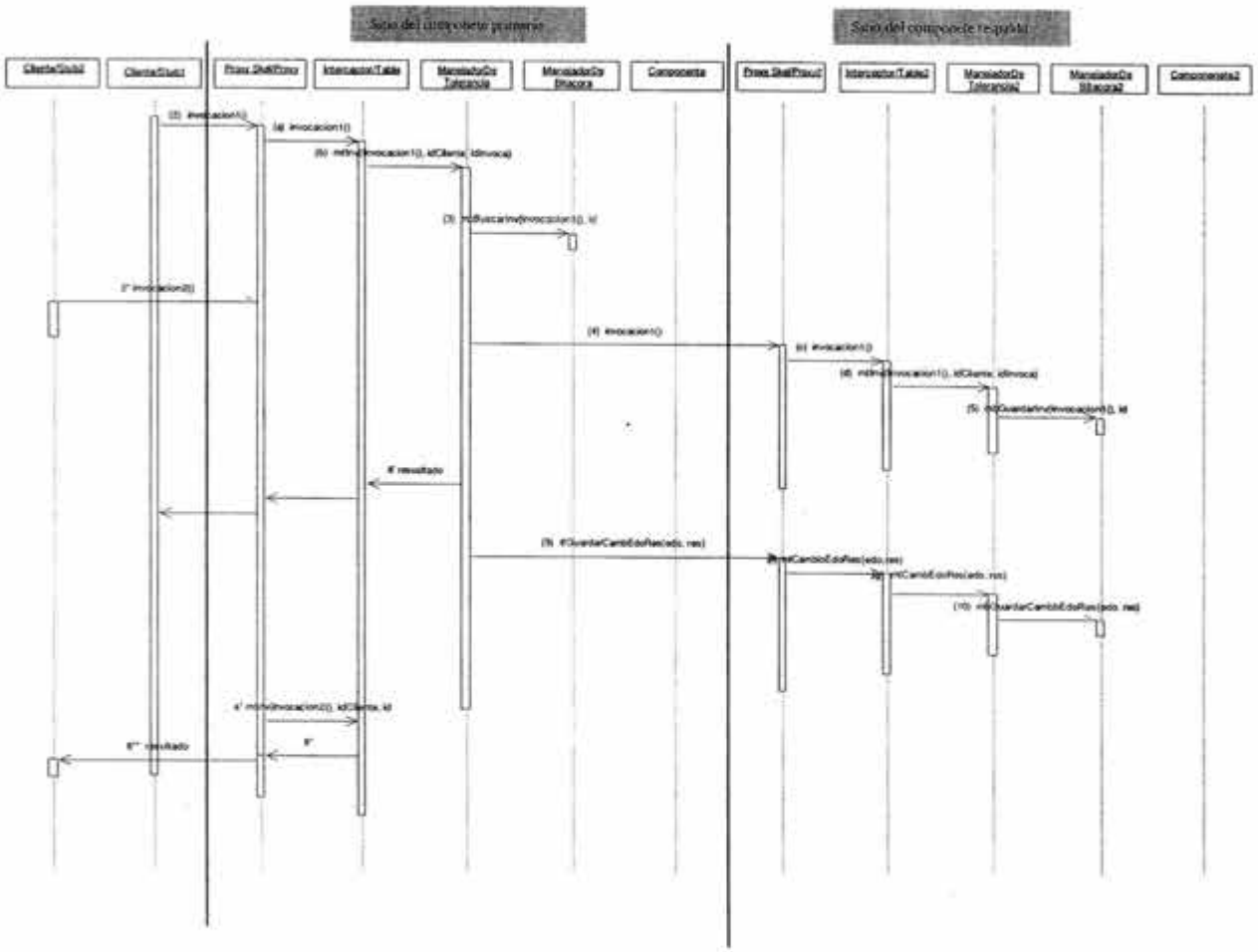


Fig. 5-8 Invocación con más de un cliente a la vez

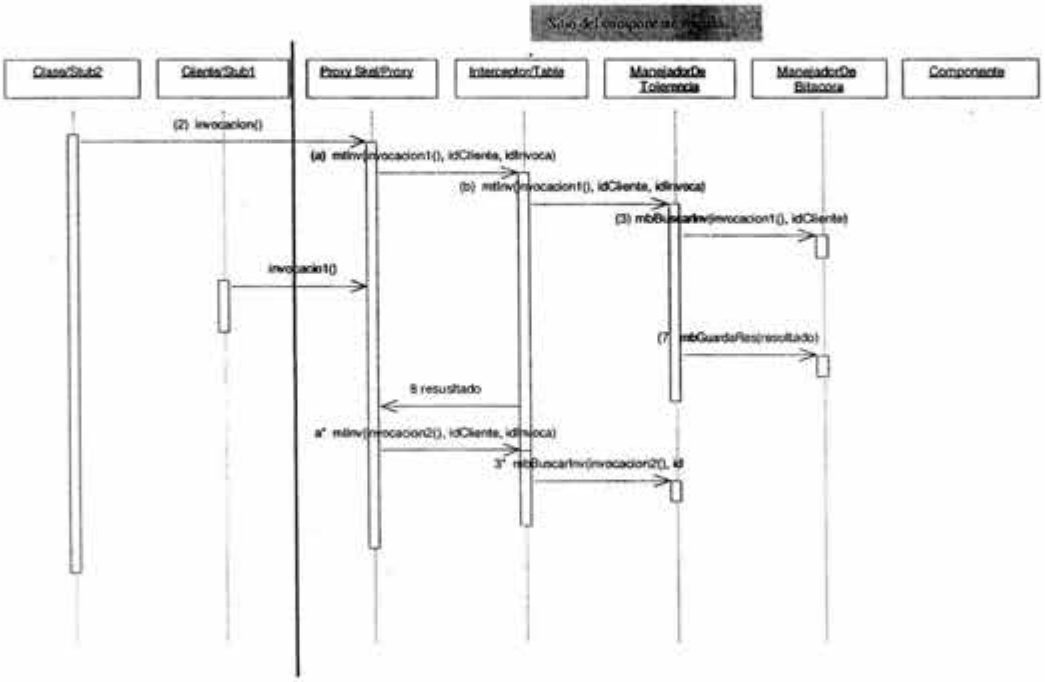


Fig. 5-9 Invocación con más de un cliente y con falla

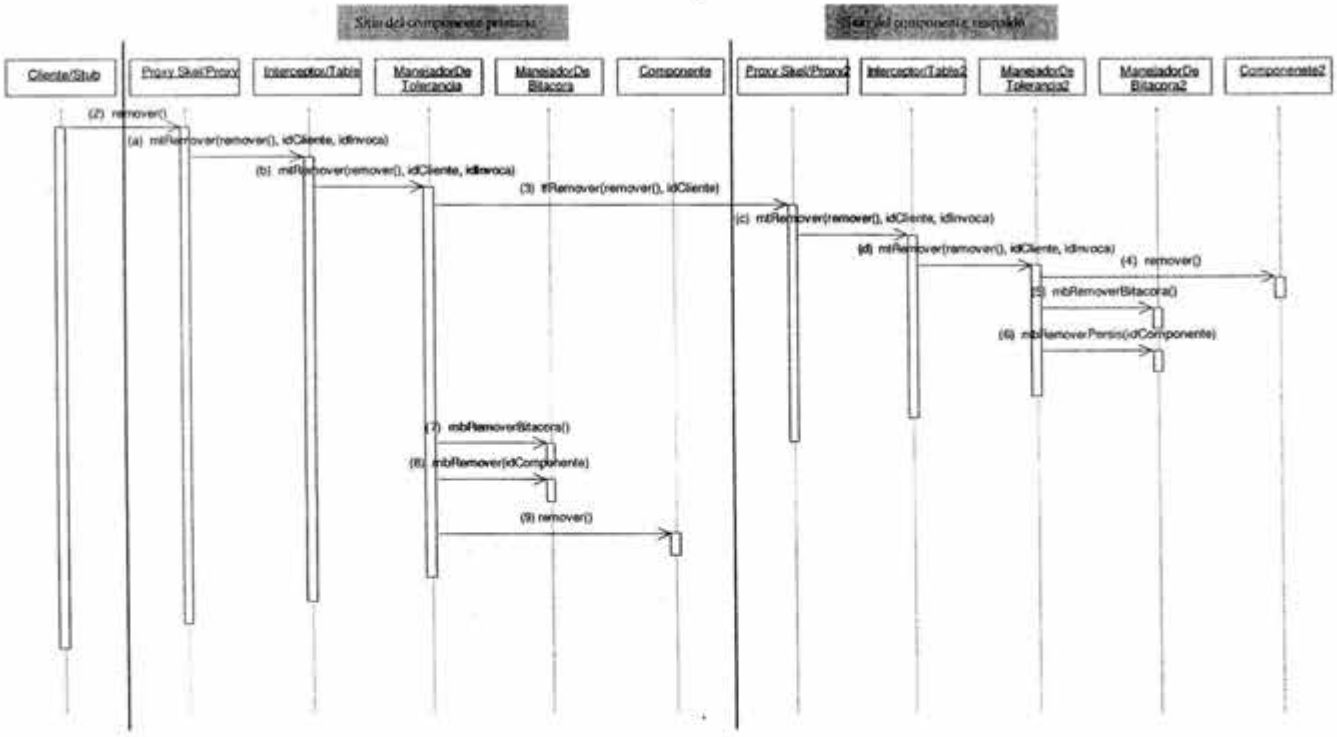


Fig. 5-10 Eliminar (remover) un componente

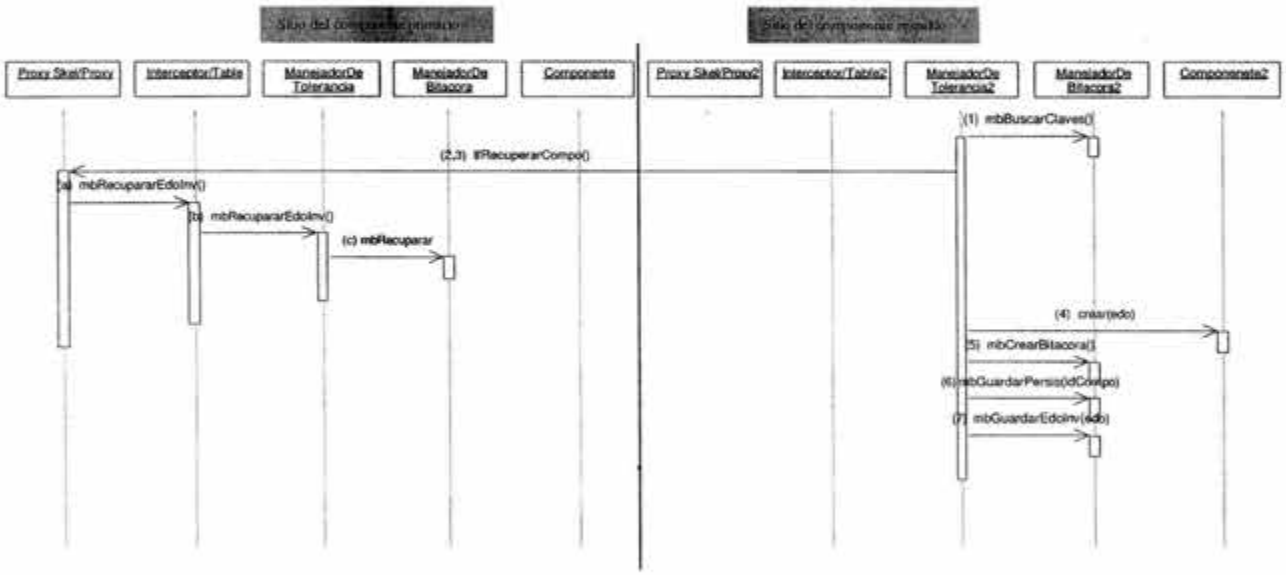


Fig. 5-11 Recuperación a través de la bitácora

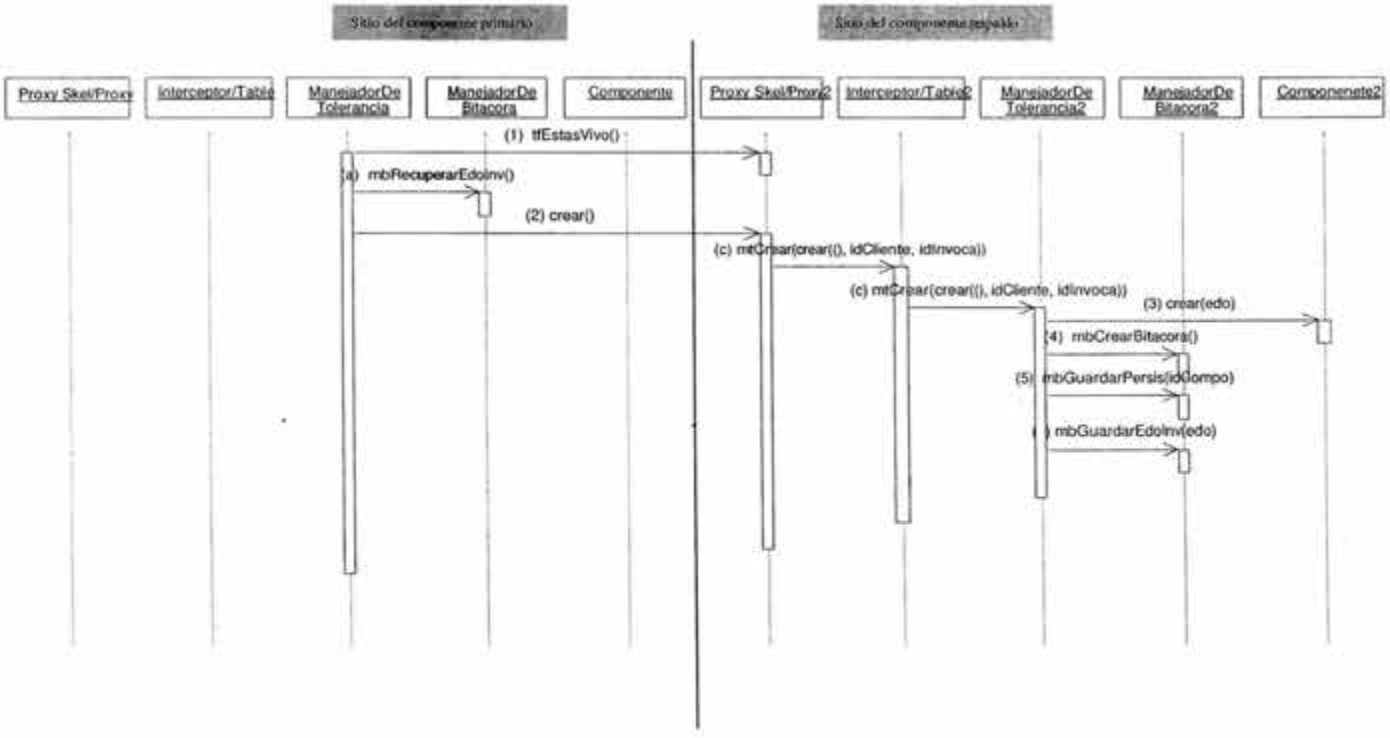


Fig. 5-12 Recuperación a través del componente primario

Especificación de las nuevas clases

En esta sección mostraremos la especificación las clases ManejadorDeTolerencia y ManjadorDeBitacora. Dicha especificación es la siguiente:

a) Interfaz ManejadorDeTolerancia

public abstract interface ManejadorDeTolerancia
que extiende de `java.rmi.Remote`
paquete co.udu.uniandes.séneca.componentManager

El proveedor de Contenedor es el responsable de implementar los métodos de esta interfaz en la clase ManejadorDeTolerancia que será parte de este contenedor Séneca.

La clase ManejadorDeTolerancia (auxiliada por la clase ManejadorDeBitácora) es la encargada de:

- Mantener la consistencia de los estados del componente primario y el componente respaldo a través de la replicación de las invocaciones.
- Es la encargada de detectar la falla y tomar medidas para que no se interrumpa el servicio aún cuando uno de los componentes (primario o respaldo) falle.
- Es la encargada de reincorporar al sistema un componente que regresa de una falla. Esta interfaz muestra los métodos que deben de ser usados para soportar la tolerancia a fallas.

La clase ManejadorDeTolerancia cuenta con los siguientes métodos:

mtInv	Void **mtInv**(string *invocación* , string *idCliente*, string *idInvoca*) Este método es el encargado de replicar la *invocación* al componente respaldo que es hecha por un cliente al componente primario siguiendo los pasos descritos en la Fig. 5-6. El mecanismo con el que cuenta este método para saber si el componente respaldo ha fallado o no, consta de comparar la variable que indican la identidad de quien hace la invocación (*idInvoca*) y la variable que indica la identidad del cliente (*idCliente*). Si estas variables son iguales, significa que ha ocurrido una falla en el componente primario y la clase ManejadorDeTolerancia tomará las acciones descritas en la Fig. 5-7 si la invocación es hecha por un solo cliente, en caso de que la invocación sea parte una serie de invocaciones que son hechas por más de un cliente a la vez las acciones a tomar son descritas en la Fig. 5-8.
mtCambioEdoRes	void **mtCambioEdoRes**(object *cambEdo*, string *res*) A través de este método la clase ManejadorDeTolerancia que se encuentra en el sitio del componente respaldo es provista de los cambios que sufrió el estado (*cambEdo*) del componente primario al ejecutar la última invocación, este cambio de estado es guardado en la bitácora junto con el resultado que haya arrojado el componente después de la ejecución de dicha invocación.

	El cambio de estado es pasado al ManejadorDeTolerancia a través de la variable *cambEdo* que es un vector de strings donde se guardan los nombres y los valores de las variables que sufrieron cambios durante la ejecución de la invocación.

Tabla 5-3

El protocolo que seguirán los métodos para comunicarse con las demás clases del Contenedor es el mostrado en el capítulo 4.

Interfaz ManejadorDeBitácora

public abstract interface ManejadorDeBitácora
que extiende de `java.rmi.Remote`
paquete co.udu.uniandes.séneca.componentManager

El proveedor de Contenedor es el responsable de implementar los métodos de esta interfaz en la clase ManejadorDeBitácora que será parte de este contenedor.

La clase ManejadorDeBitácora es la encargada de asignar, dar formato, administrar y liberar el área de memoria primaria que es utilizada para la bitácora de cada uno de los componentes que se ejecuten en el mismo sitio que ella. Además, también es la encargada de crear y administrar una área en memoria persistente donde se guardarán las claves de todas las instancias de los componentes que se encuentren en ejecución en el sitio.

La clase ManejadorDeBitacora es una clase auxiliar de la clase ManejadorDeTolerancia para soportar la tolerancia a fallas. Esta clase cuenta con los siguientes métodos:

mbBuscarInv	Boolean **mbBuscarInv**(string *invocación*, string idCliente) La invocación de este método indica, a esta clase, que debe de buscar en la bitácora del componente un registro que coincida con la variable *invocación* y la variable *cliente*. De encontrar la clase dicho registro, el método regresará un valor de verdadero, de lo contrario regresará un valor de falso.
mbGuardarInv	void **mbGuardarInv**(string *invocación*, string *idCliente*) Este método indica, a esta clase, guarde en la bitácora del componente las variables *invocación* y *idCliente*
mbGuardaRes	void **mbGuardaRes**(string *res*) Este método indica, a esta clase, guarde la variable *res* que corresponde al resultado de la última invocación que se guardó en la bitácora.

mbGuardarCambEdoRes	void **mbGuardarCambEdoRes**(object *cambEdo*, string *res*) Este método indica, a esta clase, guarde las variables *cambEdo* y *res* que corresponden al cambio de estado y el resultado del componente primario después de la ejecución de la última invocación que fue guardada en la bitácora.
mbCrearBitacora	void **mbCrearBitacora()** Asigna y da formato al área que será utilizada como bitácora del componente. El formato de la bitácora deberá de contar con registro que a su vez tendrán campos donde se guardaran variables de tipo string. En los cuales se guarda la invocación, la identidad de cliente, la identidad de que hace la invocación, los cambios de estado que sufra el componente y el resultado de la invocación.
mbGuardarPersis	void **mbGuardarPersis** (string *idComponente*) Guarda en la bitácora que se encuentra en memoria persistente la variable *idComponente*, en la cual se encuentra guardado el tipo de componente que ha sido creado. Para que en caso de fallas se cree nuevamente.
mbRemoverPersis	void **mbRemoverPersis** (string *idComponente*) Borra de la bitácora que se encuentra en memoria persistente la variable *idComponente*, en la cual se encuentra el tipo de componente que ha sido removido. Para que en caso de fallas no se intente crear nuevamente.
MbRecuparar EdoInv	object **mbRecuparar EdoInv** () Al invocar este método se le indica, a esta clase, que busque y entregue el último estado del componente que se encuentre en la bitácora junto con las invocaciones que no se hayan aún ejecutado. El método regresará el estado y las invocaciones en una variable de tipo vector, donde el primer elemento será otro vector donde se encontraran todas las variables del estado, seguido de este vector se encontrarán las invocaciones en el orden en que fueron guardadas en la bitácora.

Tabla 5-4

El protocolo que seguirán estos métodos para comunicarse con las demás clases del Contenedor es el mostrado en el capítulo 4

5.4 Evaluación

En esta sección presentaremos la evaluación cualitativa que abarca la integración de tolerancia a fallas a través del modelo abstracto y sus algoritmos, definido en el capítulo 4, sobre la especificación de Séneca que da paso a SénecaTaF. Así también, la evaluación de la implementación de un prototipo que toma como base a Séneca-j.

5.4.1 Evaluación del modelo abstracto sobre Séneca

La evaluación consiste principalmente en mostrar la dificultad para lograr las características que fueron agregadas en SénecaTaF en términos de la especificación del modelo abstracto, la cual a continuación se describe:

- **Actores.** La forma en que se definen los actores en el modelo abstracto (como son cliente, capa de servicios y componente) coinciden con la forma en que fueron definidos en Séneca. Esto implica que no se tuvieron que hacer cambios en este sentido para incorporar los actores del modelo abstracto en SénecaTaF.

- **Nuevas responsabilidades de los actores.** En el caso de Séneca no se especifica de manera explícita a los elementos que realizan las tareas de los actores en el modelo a componentes. En otras palabras, en la especificación de Séneca no se especifica el nombre, funcionalidad, ni la forma en que se relacionan las clases que realizan las tareas que los actores tienen asignadas. Por esta razón, la integración de tolerancia a fallas en Séneca resulta simple, ya que sólo se debe de definir la semántica de fallas y la forma en que se debe de pedir el servicio al middleware. Pero lo anterior no siempre es así, en el caso de EJB integración del modelo abstracto (tolerancia a fallas) requiere de un mayor análisis como es mostrado en el anexo de este trabajo.

- **Algoritmos y nivel de tolerancia a fallas.** En cuanto a los algoritmos, al no estar especificadas las clases y sus funcionalidades en Séneca, no es necesario definir en SénecaTaF las clases que realizaran la tolerancia a fallas ni los algoritmos de sus comportamientos aún cuando su especificación señala que la semántica y mecanismos de tolerancia a fallas, deberán de ser tomados del modelo abstracto. Por otro lado, el nivel de tolerancia usado en SénecaTaF, que especifica que la tolerancia a fallas puede ser agregada o no a cada componente según el criterio del implementador (N3), resulta ser fácilmente agregada a SenecaTaF mediante la especificación de las cláusulas toleranceFault o unToleranceFault en el modelo de programación que es donde se describen tanto las características del estados de los componentes, así como la asociación entre cada interfaz definida en él y la clase que lo implementa.

5.4.2 Evaluación del prototipo

La evaluación consiste en mostrar las dificultades encontradas para implementar la especificación de SénecaTaF tomando como base un prototipo existente de Séneca llamado Séneca-j. Es importante resaltar que la versión obtenida del prototipo de Séneca-j implementa parcialmente a Séneca y que entre sus principales carencia se encuentra la falta de integración de la distribución así como su poca (casi nula) documentación de lo implementado en él, lo anterior debido que Séneca es concebido como material didáctico y hasta la fecha se sigue trabajando en su especificación.

A continuación se describe la forma en que fue lograda la implementación de las características definidas en SénecaTaF sobre Séneca-j dando paso así a SenecaTaF-j.

Documentación

Como se ha mencionado con anterioridad, la documentación obtenida al inicio de Séneca-j fue relativamente nula, así que el primer paso para la implementación de SénecaTaF-j fue hacer una documentación que nos permitiera identificar a cada uno de los elementos que conformaban a Séneca-j así como su funcionalidad y la relación que guardan entre ellos. Lo anterior se logró estudiando el código de las clases y agregándoles mensajes en lugares estratégicos a cada una de éstas de tal forma que ejecutando varias veces un ejemplo que acompañaba al código fuente se obtuviera, como principal resultado, la sección 5.2 donde se menciona el funcionamiento de Séneca-j.

Como parte de la documentación, también se estudió el metalenguaje XML, las especificaciones y uso de Ant ver. 1.5, velocity-1.3.1-rc2, junit3.8.1, javacc2.1, jdk1.3 y Java-RMI.

Distribución

Después de la falta de documentación, la falta de implementación de la distribución era el problema más grande que nos encontramos en Séneca-j, el cual fue salvado simulando varias máquinas levantando más de un servidor y más de un cliente de RMI en la misma máquina, lo cual nos permitió experimentar el comportamiento del sistema al fallar un sitio.

Bitácoras y reenvío de mensajes

Se hicieron pruebas donde se encontró que era factible agregar en Séneca-j una área de memoria en RAM (bitácora) donde se guardaran todas las invocaciones que se hicieran al componente, también se encontró que era factible reenviar la invocación a otro componentes que se encontrara en otro servidor y que esta invocación se ejecute en este servidor antes de que se ejecute en el componente al que fue enviado originalmente.

Clases Interceptor y ManejadorDeTolerancia

La clase *ManejadorDeTolerancia* que es la encarga de mantener la consistencia en réplicas de los componentes, según la especificación de SénecaTaF-j, será llamada desde la clase *Interceptor* que es la clase encargada de mantener la interacción del cliente con el componente y todos sus elementos[1], lo anterior se logra haciendo pequeñas modificando a la plantilla *ComponentInterceptorTemplate*.

Es importante mencionar que la clase ManejadorDeTolerancia y ManejadorDeBitacora aún no ha sido terminada su implementación, pero que las pruebas que se han hecho sobre los avances de las mismas han sido en su mayoría favorables.

Nivel de tolerancia

La integración de las clausuras *toleranceFault* y *unTolerenceFault* que agregan o no la tolerancia a fallas de nivel N3 y su implementantación, se logró agregando las anteriores cláusulas como

[1] El componente se puede ver como una colección de distintas clases que implementa cada uno de sus métodos y estado.

Faltan páginas

N° 97 - 98

disponibilidad al contar con un componente respaldo que aumenta la probabilidad de que el servicio este disponible.

6.2 Conclusiones

Con base en el estudio realizado y sus resultados se llegó a las siguientes conclusiones:

- Es posible diseñar un servicio de tolerancia a fallas específico el cual puede ser incluido opcionalmente en un middleware a componentes que cumpla con el modelo genérico cliente-servidor-contenedor.
- La tolerancia a fallas puede ser agregada a una aplicación distribuida a granularidades diferentes en un middleware a componentes (adaptabilidad).
- Es factible, agregar el servicio de tolerancia a fallas sin alterar la funcionalidad y semántica del middleware.

6.3 Perspectivas

Antes de enunciar las perspectivas y trabajos futuros, una pregunta válida sería ¿si comenzáramos de nuevo cómo lo haríamos? y la respuesta sería: de la misma manera. Ya que desde mi actual punto vista el camino recorrido, aún cuando fue duro y largo, fue necesario ya que el conocimiento que tenía sobre los sistemas distribuidos y tolerancia a fallas era poco y equivocado.

Para continuar la exploración iniciada en este trabajo se recomienda:

- Extender la implementación del prototipo SenecaTaF-j para poder hacer un estudio de rendimientos.

- Estudiar cómo hacer tolerantes a fallas a los elementos que proveen la tolerancia a fallas en el modelo genérico.

- Explorar el enriquecimiento de los mecanismos integrados. Por ejemplo, integrar la replicación con lo cual se podría lograr que la falla no se perciba haciendo que existan en todo momento al menos 2 réplicas activas.

- Estudiar y proponer la especificación de una interfaz genérica que pueda ser integrada a las especificaciones de middlewares a componentes que permita agregarles el servicio de tolerancia a fallas sin necesidad de cambiar su especificación.

- Estudiar la interacción con otros servicios no funcionales ya que en este trabajo no se ha hecho. Por ejemplo, la interacción entre la tolerancia a fallas y el balance de carga para saber donde desplegar los componentes ya que hasta el momento se hace arbitrariamente.

ANEXO

A INTEGRACION DE TaF EN EL MODELO EJB

Como ya se ha mencionado, el modelo EJB permite a los desarrollador de beans concentrarse en la parte funcional de una aplicación distribuida sin distraerse en los detalles del soporte del sistema, ya que los beans pueden usar los servicios middleware del servidor y contenedor EJB de manera implícita y transparente, esto es, los beans pueden usar los servicios middleware sin necesidad de usar código adicional. Pero para aquellas aplicaciones que requieren seguir con el correcto funcionamiento de su sistema aún en presencia de fallas, no encuentran soporte en el modelo EJB, ya que en la actual especificación no se incorpora, como servicio middleware, el manejo de tolerancia a fallas[1]. Por lo que se tomó la decisión de integrar el modelo abstracto de tolerancia a fallas propuesto en del capítulo 4, en el modelo EJB usando el nivel transparencia N3 (ver sección 4.1) dando origen al modelo EJB Tolerante a fallas (EJBTaF). Dicha integración se describe a continuación.

A.1 Prerrequisitos

Para lograr el comportamiento de tolerancia a fallas, tal como se proponemos en el capítulo 4, de las aplicaciones que se implementan siguiendo el modelo EJB, deben de tener en cuenta los siguientes prerrequisitos que no necesariamente deben de estar contemplados en dichos sistemas:

- Supondremos un canal de comunicación confiable, unidireccional y *FIFO* (primero que entra primero que sale, por sus siglas en inglés).
- La latencia de comunicación, i.e. el tiempo que tarda un mensaje en llegar del emisor al receptor, es finito y acotado.
- Los procesos locales en los sitios del sistema son determinísticos, esto es, que la ejecución en el mismo orden de las mismas operaciones siempre generan el mismo estado.
- Supondremos también que los sitios no tienen una carga excesiva y siempre que no hayan sufrido una falla podrán contestar en tiempo.
- Supondremos que cuando más de un cliente invoca un método de un mismo bean/componente a la vez el contenedor sólo efectúa una ejecución serial[2] de las invocaciones.

A.2 Equivalencias entre el modelo abstracto y el Modelo EJB

Como primer paso en la implementación del modelo abstracto en el modelo EJB, es necesario hacer coincidir los actores de ambos modelos, para ello se ha creado la siguiente tabla:

---

[1] Cabe aclarar que en especificación del modelo EJB [10] si existen algunos mecanismos como son; el manejo de transacciones y la persistencia de datos que permiten tolerar falla. Pero los cuales por si solos no los consideramos como un servicio de tolerancia fallas.

[2] En realidad el contenedor EJB también puede efectuar ejecuciones en paralelo, pero nuestra propuesta se limita a ejecuciones seriales dejando el punto del paralelismo para trabajos posteriores.

Modelo Abstracto		Modelo EJB
Cliente		Cliente
Capa de servicios	Sitio del cliente	EJB home stub, EJB object stub (y las clases de la interfaz JNDI que son parte de la interfaz EJB home)
	sitio del bean	EJB home skeleton, EJB object skeleton. Además del ContenedorTF y Servidor EJB con los que sólo interactúan directamente las mencionadas interfaces y no los bean.
Componente		JavaBean (bean)

Tabla. A-1 Equivalencias entre los modelos

A.3 Nuevas Responsabilidades en el Modelo EJB

Las nuevas responsabilidades de los actores del modelo EJB se obtienen a través de las equivalencias que se muestran en la Tabla A-1 y el empate que tienen con los actores y sus responsabilidades mostradas en la sección 4.4. Así de esta forma se obtiene la siguiente responsabilidades:

Cliente. No tendrá nuevas responsabilidades.

Middleware EJB. Seguirá ofreciendo todos los servicios que ya proporcionaba, además de realizar las siguientes tareas:

- Las interfaces EJBHome stub y EJBObject stub funcionarán de la forma en que son especificadas en [10]. Además se agregan las interfaces EJBHomeTF (Tolerante a fallas) stub y EJBObject stub que funcionarán de la misma forma que las anteriores interfaces excepto que:

    a) La primera vez que haga una invocación a un bean lo hará al bean primario.
    b) Espera un tiempo determinado (timeout) después de hacer la invocación, si antes del término de este tiempo, el sitio del bean no responde, se tomará esa acción como una falla del sitio.
    c) Al darse cuenta de que el bean ha fallado, reenviará la invocación pero en esta ocasión lo hará al bean respaldo.

- Las interfaces EJBHome skeleton y EJBObject skeleton en los sitios de ambos beans (primario y respaldo) funcionarán de la forma en que son especificadas en [10]. Además se agregan las interfaces EJBHomeTF skeleton y EJBObject skeleton que funcionarán de la misma forma que las anteriores interfaces excepto que:

    d) Hará la administración de la *bitácora* asignada a cada componente.

e) Maneja los mensajes para la consistencia entre los sitios del bean primario y el sitio del bean respaldo (según el algoritmo de la Fig.4-5)
f) Cada vez que un bean sea creado guardará en una área de memoria estable una clave que indique qué bean fue creado.
g) Cuando regrese de una falla buscará en memoria estable qué beans eran utilizados y los creará nuevamente, después enviará un mensaje al bean principal para que sepa que ya regreso de la falla y sea actualizado para que pueda ser integrado nuevamente al sistema tomando el lugar de bean respaldo.
h) Cada vez que un bean sea eliminado, en la memoria estable se eliminará la clave de este.
i) Verificará con la ayuda de la bitácora si el bean ya ejecutó cada una de las invocaciones que arriban al sitio. Si una invocación ya fue ejecutada por el bean no será pasada nuevamente a él para que la reejecute, en ese caso, se obtendrá el resultado de la bitácora y se envía al cliente.

- Las interfaces EJBHome skeleton y EJBObject skeleton en el sitio del bean primario funcionarán de la forma en que son especificadas en [10]. Además se agregan las interfaces EJBHomeTF skeleton y EJBObject skeleton que funcionarán de la misma forma que las anteriores interfaces excepto que:

  j) Cuando haga una invocación al sitio del bean respaldo, esperará la respuesta un tiempo determinado (timeout), si antes del término de este tiempo, el sitio del bean respaldo no responde, se tomará esa acción como una falla del sitio.
  k) Cuando detecte que el sitio del bean respaldo ha fallado, dejará de enviarle las invocaciones que usaba para mantener la consistencia entre ambos, y las enviará nuevamente hasta que el sitio del bean respaldo haya regreso de la falla y este listo

- Las interfaces EJBHome skeleton y EJBObject skeleton en el sitio del bean respaldo funcionarán de la forma en que son especificadas en [10]. Además se agregan las interfaces EJBHomeTF skeleton y EJBObject skeleton que funcionarán de la misma forma que las anteriores interfaces excepto que:

  l) Cuando detecte que las invocaciones son hechas por el cliente y no por el bean primario, entonces sabrá que el componente primario ha fallado y manejará la falla (según el algoritmo de la Fig.4-5)

bean. Seguirá trabajando según sus especificaciones agregando la siguiente funcionalidad:

  m) Contará con dos interfaces, una que permita a la capa de servicios obtener su estado interno y otra que permita la capa de servicios cambias dicho estado.

Debido a que con la actual especificación no se podría lograr que los actores (con sus nuevas interfaces) asumirán las nuevas responsabilidades, se propone agregar las siguientes clase al contenedor:

- Manejador de Tolerancia. Esta clase será la encarga de replicar la información en el sitio del bean respaldo. También serán la encargada de la detección, manejo y recuperación de los bean que presentan una falla.

- Manejador de Bitácora. Ésta es una clase auxiliar de la clase anterior, la cual será la encarga de la asignación, creación, eliminación, modificación y administración del área de memoria primaria y persistente que requiere el Manejador de Tolerancia para cumplir su objetivo.

El soporte a las nuevas interfaces y la integración de las nuevas clases dan como resultado el ContenedorTF (contenedor tolerante a fallas).

A.5 Modelo abstracto y modelo EJBTaF

En el modelo abstracto, del capítulo 4, se muestran los algoritmos (genéricos) propuestos que se deben de seguir para prevenir que en caso de presentarse una falla, en alguno de los sitios, el sistema deje de funcionar correctamente. También se muestran los algoritmos que se deben de seguir durante y después de la falla. Todos estos algoritmos deben de ser ajustados para que puedan ser agregados en EJB.

Las dos nuevas clase, Manejador de Tolerancia y Manejador de Bitácora, que son agregadas al contenedor EJB, proporcionan los métodos necesarios para lograr implementar los algoritmos propuestos en el modelo abstracto sobre el modelo EJB. Pero como es de esperarse, la integración de dichas clases requieren de rescribir e integrar nuevas especificaciones al modelo EJB, lo anterior con la intención de mostrar las formas de uso de las clases nuevas, así como los protocolos de comunicación que serán usados entre ellas, y las clases ya existentes en el contenedor. Dando origen el modelo EJBTaF (EJB tolerante a fallas)

A.5.1 Nivel de transparencia

El nivel de transparencia N3 buscado para EJBTaF se logra haciendo que el ContenedorTaF sea capaz de manejar los dos tipos de beans, aquellos que son tolerantes a fallas y que implementan de javax.ejb.SessionBeanTF o javax.ejb.EntityBeanTF y aquellos que no los son e implementan de javax.ejb.SessionBean o javax.ejb.EntityBean.

Fig. A-1 Parte del Contenedor que contempla tolerancia a fallas

Cuando el ContenedorTaF detecta que una de sus herramientas hará la compilación de un bean tolerante a fallas, crean dos beans idénticos (bean primario y bean réplica) que sólo cambiarán por el nombre, ambos tendrán casi el mismo nombre excepto que uno de ellos (la réplica) al final le agregará el número 2. Al momento de desplegado el bean réplica será colocado en un sitio distinto al del bean primario y esto se hará de manera transparente para el desarrollador de la aplicación, y cuando el ContenedorTaF detecta que una de sus herramientas hará la compilación de un bean que no es tolerante a fallas lo tratará como es definido en [10].

A.6 Algoritmos para tolerar las fallas

La forma en que interactúan las dos nuevas clases para lograr la tolerancia a fallas[1], Manejador de Tolerancia y Manejador de Bitácora, son mostradas en las figuras de la sección A.6.3, las cuales están basadas en los casos de fallas previstos en el modelo abstracto. Mientras que los detallas de las interfaces y métodos son mostrados en las secciones A.9 y A.10.

A.6.1 Simplificación de diagramas

Como ya lo hemos mencionado, se ha supuesto que las interfaces EJBObject y EJBHome se pueden dividir en dos partes, una que contenga las clases que manejen la concurrencia y seguridad, y otra que contenga las clases que manejan los servicios restantes (como transacciones, persistencia, ciclo de vida, etc.).

Así que, para simplificar los diagramas mostrados en este capítulo, se harán las siguientes agrupaciones en las figuras de este capítulo:

- Para la Fig. A-1 se agrupan las clases que manejan la seguridad, concurrencia y las clases del Skeletor en una sola clase que se llamará **EJB Object 1**, y las restantes clases del contenedor las agruparemos en una sola clase que se llamará **EJB Object 2**.
- Para las figuras subsecuentes, además del lado del cliente se agruparán las clases del cliente y el Stub. A esta agrupación la llamaremos **Cliente/Stub**.

A.6.2 Ajuste de algoritmos

Los casos de fallas previstos en el modelo abstracto han sido colocados en una Tabla A-2 junto con las acciones que fueron tomadas para ajustarlos al modelo EJB.

Caso	Acción
Creación del bean (Nacimiento)	
Funcionamiento básico normal (libre de fallas)	Los cambios que sufre el diagrama de la Fig. 4-2 son mostrados en la Fig. A-1.
Invocación con falla de alguno de los participantes	
Falla del bean primario	Los cambios que sufre el diagrama de la Fig. 4-4 son mostrados en la Fig. A-2.
Falla del sitio del bean respaldo	Cuando se presente este caso de falla en el modelo EJB, el Manejador de Tolerancia que se encuentra en el bean primario deja de enviar las invocaciones al sitio del bean respaldo hasta que el bean respaldo regrese de la falla.
Falla del sitio del cliente	Cuando el sitio del cliente falle, el cliente respaldo tomará su lugar si es que el cliente tuviera uno, de no ser así, se producirá una falla visible al usuario.
Algoritmo en la invocación de un método (vida)	
Algoritmo para un solo cliente	
Funcionamiento básico (libre de fallas)	Los cambios que sufre el diagrama de la Fig. 4-5 son

[1] La forma en que se deben de comunicar las clases ya existentes en el contenedor no es mostrada en este trabajo y se deja a consideración del proveedor del Contenedor TF, siempre y cuando no este contemplada en la especificación de EJB.

	mostrados en la Fig. A-3.
Invocación con falla de alguno de los Participantes	
Falla del bean primario	Los cambios que sufre el diagrama de la Fig. 4-6 son mostrados en la Fig. A-4. La reincorporación del bean se hará a través de la petición del sitio del bean primario, (véase Fig.A-9)
Falla del sitio del bean respaldo	Se sigue la misma acción que en el caso de la creación que tiene falla de un bean respaldo
Falla del sitio del cliente	Se sigue la misma acción que en el caso de la creación que tiene falla del cliente
Algoritmo para más de un cliente a la vez	
Funcionamiento básico (libre de fallas)	Los cambios que sufre el diagrama de la Fig. 4-7 son mostrados en la Fig. A-5.
Invocación con falla de alguno de los Participantes	
Falla del sitio bean primario	Los cambios que sufre el diagrama de la Fig. 4-8 son mostrados en la Fig. A-6.
Falla del sitio del bean respaldo	Se sigue la misma acción que en el caso de la creación que tiene falla de un bean respaldo
Falla del sitio del cliente	Se sigue la misma acción que en el caso de la creación que tiene falla del cliente
Eliminación del bean (Muerte)	
Funcionamiento básico normal (libre de fallas)	Los cambios que sufre el diagrama de la Fig. 4-9 son mostrados en la Fig. A-7.
Invocación con falla de uno de los participantes	
Falla del bean primario	En este caso el Stub al detectar la falla, envía la invocación al sitio del bean respaldo para que sea eliminado. El bean primario con la falla de su sitio fue eliminado parcialmente quedando en la bitácora de la memoria persistente la clave del bean, pero la cual será eliminada cuando intente reincorporarse, véase paso (3) de la Fig. A-8.
Falla del sitio del bean respaldo	En este caso el Manejador de tolerancia del sitio del bean primario al detectar la falla la ignorará y sigue la parte del algoritmo que le corresponde en su sitio, ya que igual que en el caso anterior, el bean respaldo con la falla de su sitio fue eliminado parcialmente quedando en la bitácora de la memoria persistente la clave del bean, pero la cual será eliminada cuando intente reincorporarse, véase paso (3) de la Fig. A-8.
Falla del sitio del cliente	Como ya lo hemos mencionado, al cliente se le puede modelar como un bean, que tendrá un respaldo y el cual tomará su lugar en caso de que falle este, y reenviará la invocación de eliminación al componente primario.
Reincorporación de un bean cuando regresa de una falla (Resurrección)	
A través de la búsqueda en la bitácora de la memoria persistente	Los cambios que sufre el diagrama de la Fig. 4-10 son mostrados en la Fig. A-8.
A través de la petición del sitio del componente primario	Los cambios que sufre el diagrama de la Fig. 4-11 son mostrados en la Fig. A-9.

Fig. A-2 Solución en la implementación

A.6.3 Cambios en los diagramas del modelo abstracto al modelo EJBTaF

En esta sección son mostrados los cambios que sufrieron los diagramas de los algoritmos que componen al modelo abstracto, los cambios son debidos a la incorporación de las dos nuevas clase, Manejador de Tolerancia y Manejador de Bitácora, y al ajustar estos diagramas al modelo EJB.

En los siguientes diagramas los mensajes que no aparecen en el modelo abstracto, pero que son necesarios para ajustarlos al modelo EJB, son etiquetados con una letra de en vez de un número como en el modelo abstracto.

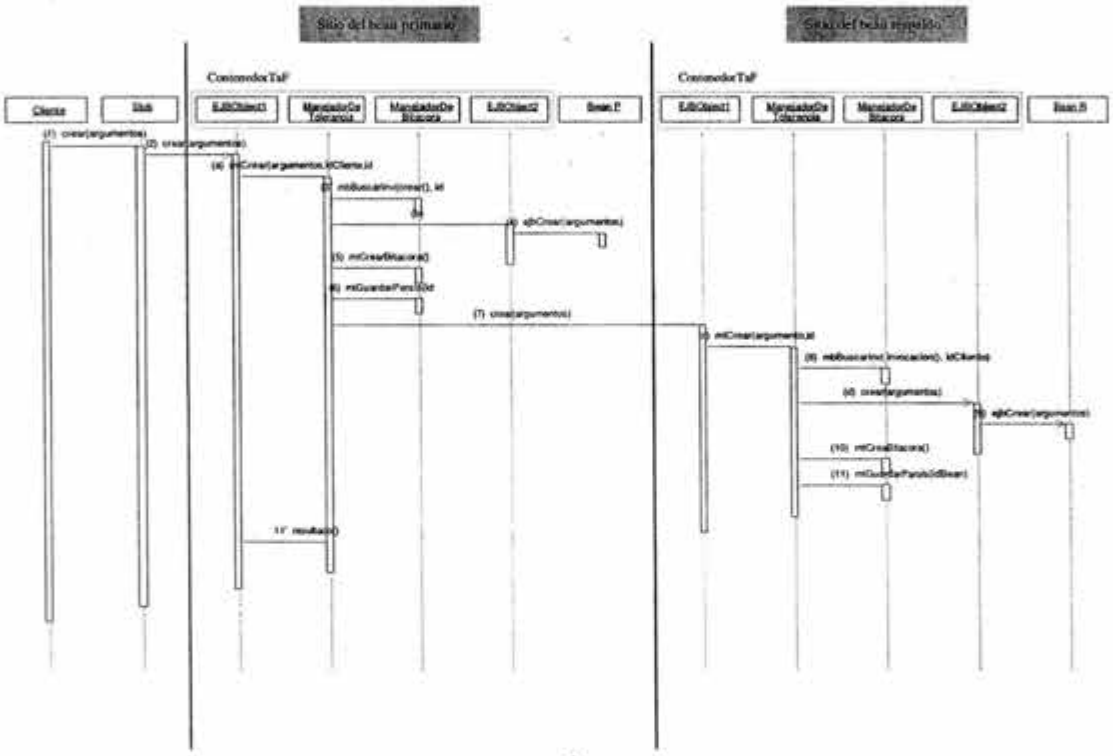


Fig. A-1 Creación

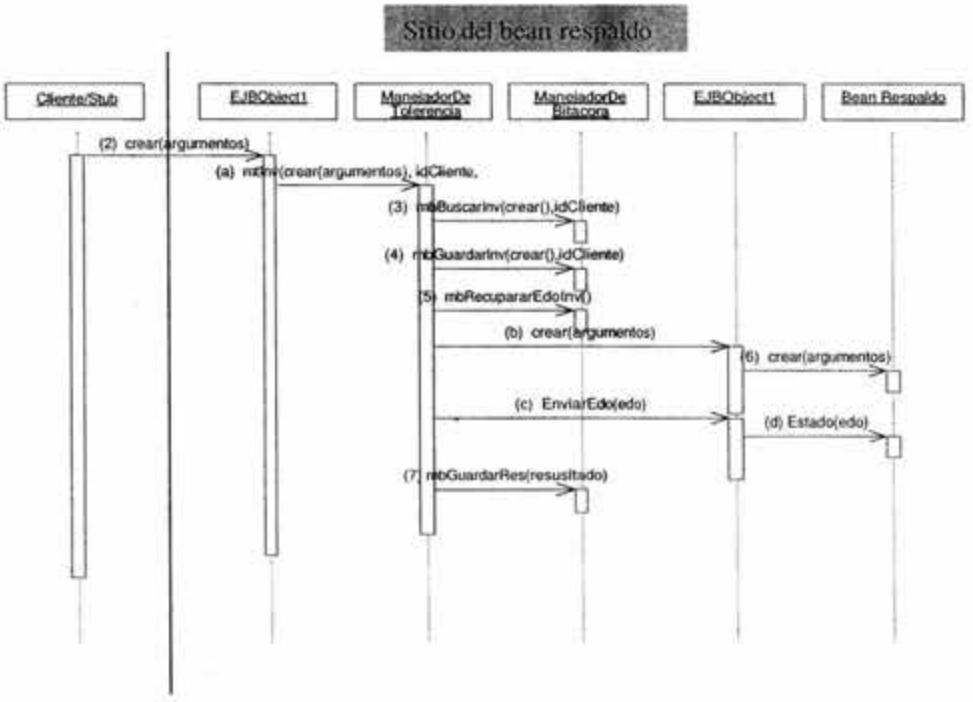


Fig A-2 Creación con falla del bean primario

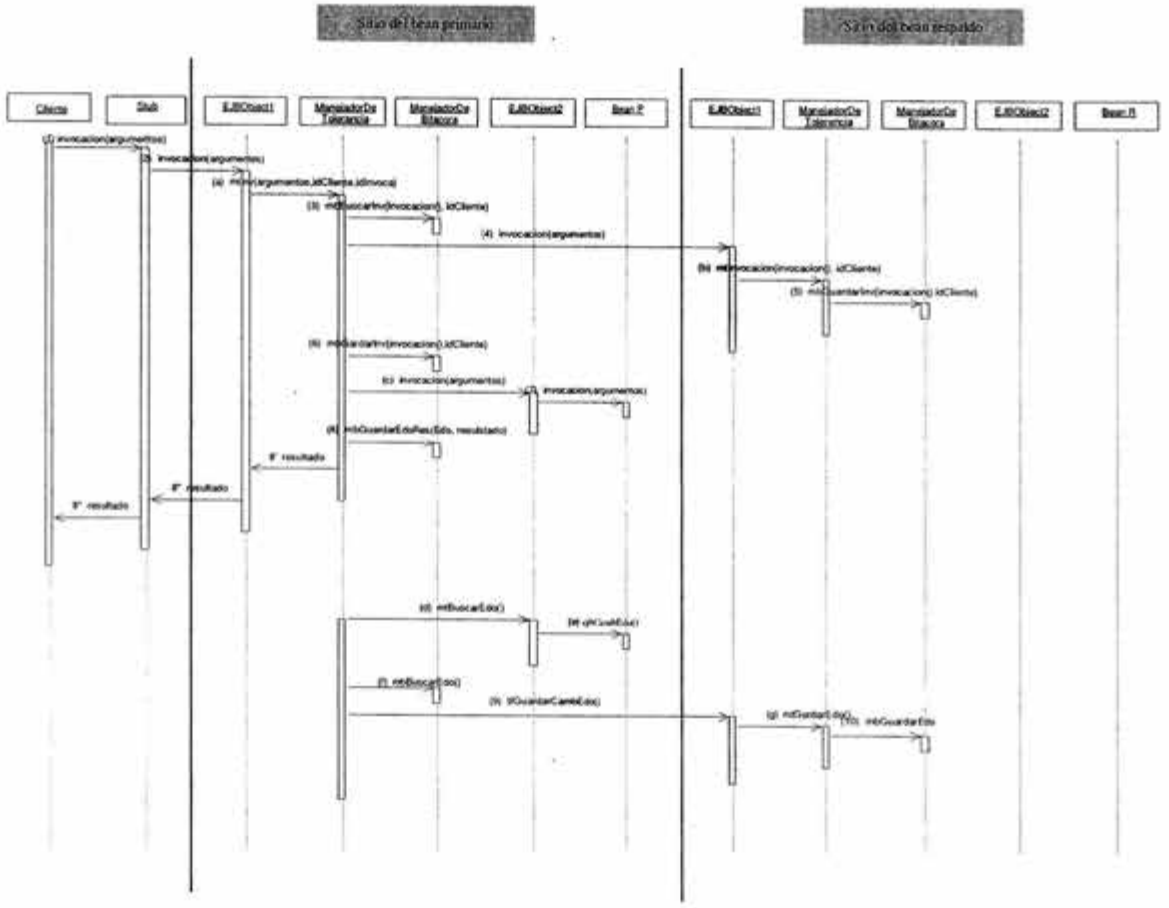


Fig. A-3 invocación

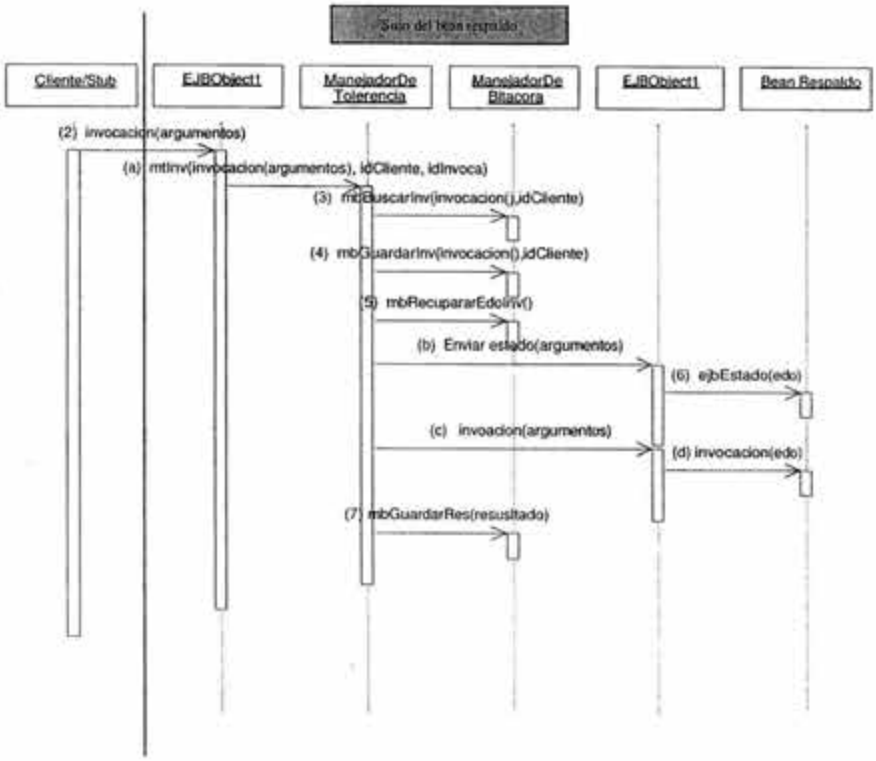


Fig. A-4 invocación con fallas del bean primario

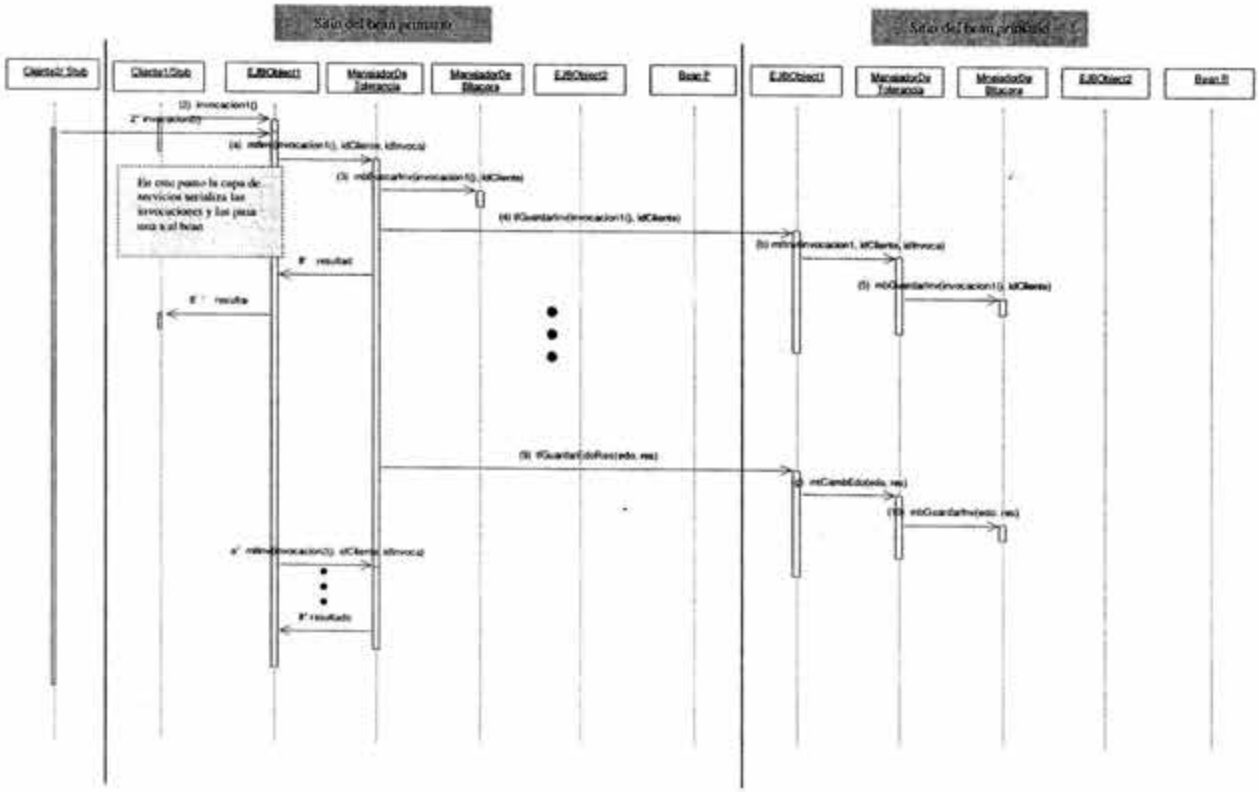


Fig. A-5 ejecucion serial

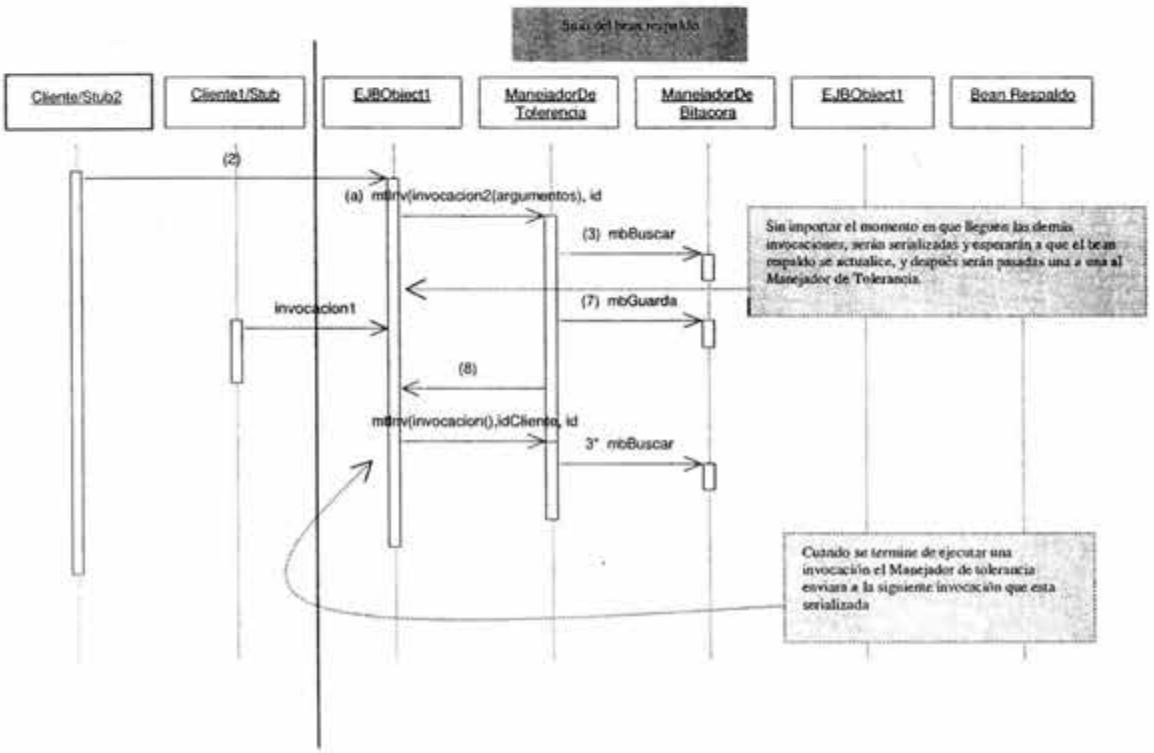


Fig. A-6 Ejecucion serial con fallas

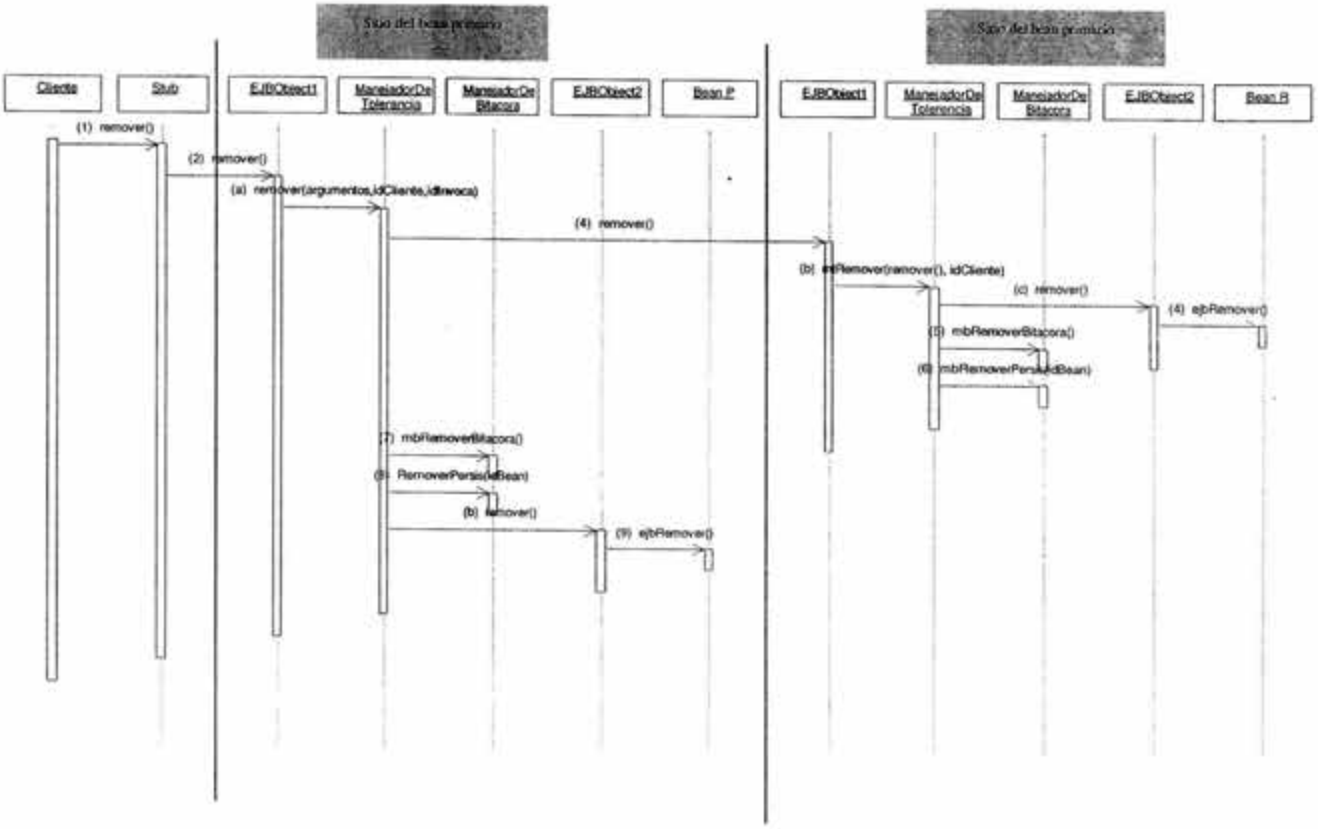


Fig. A-7 Eliminar (remover) un bean

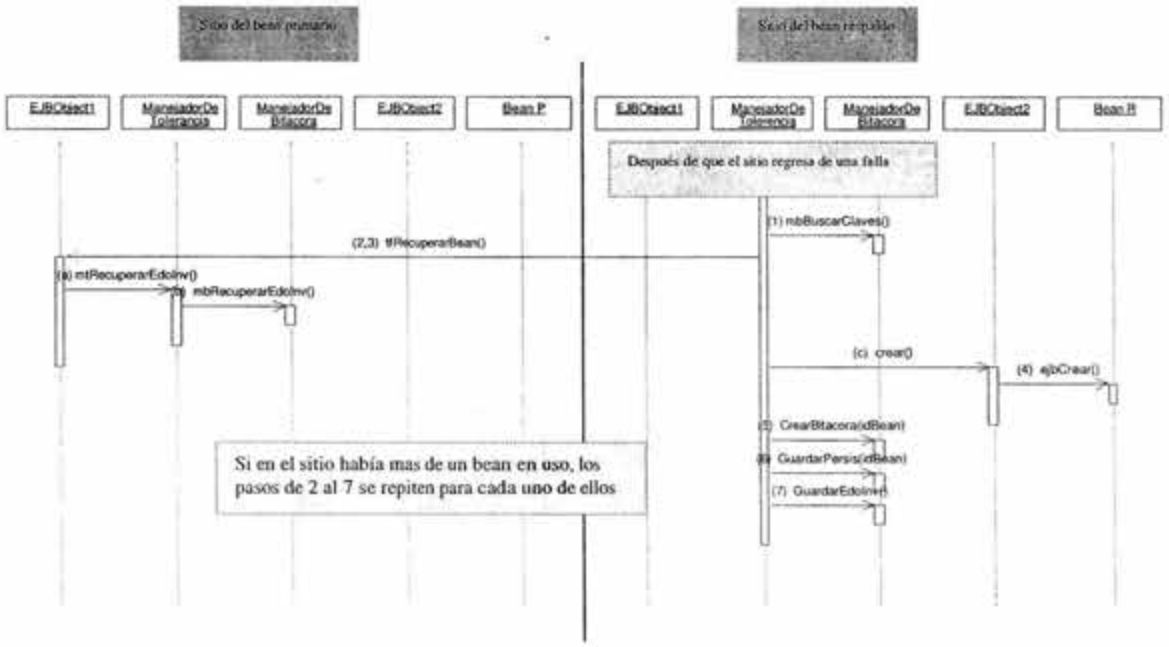


Fig. A-8 Recuparación a través de la bitácora

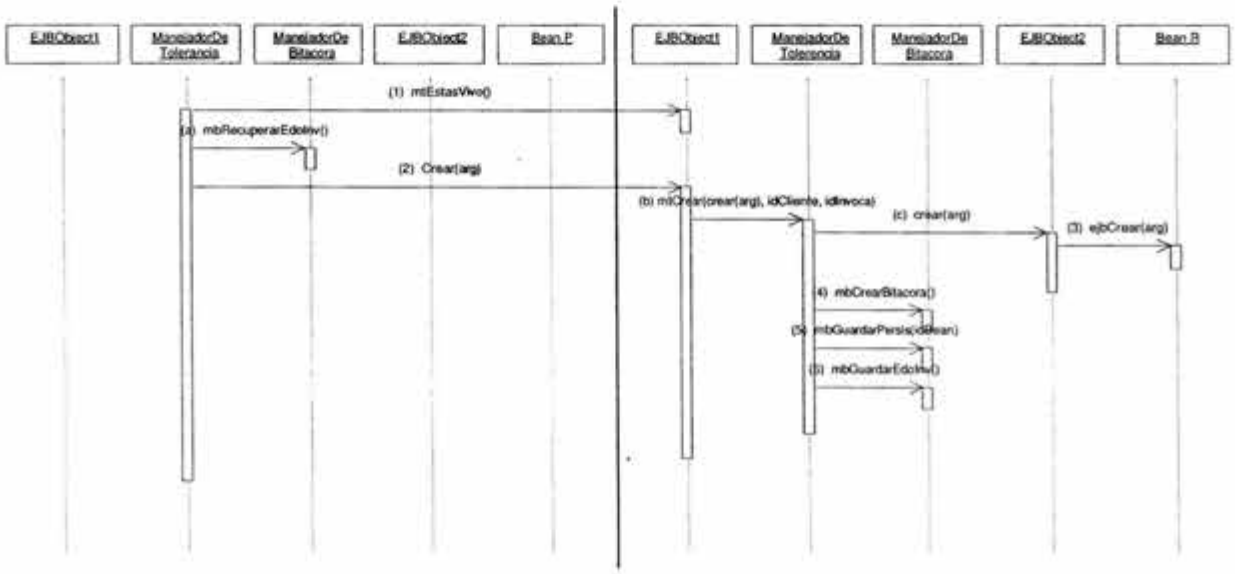


Fig. A-9 Recuperación a través del bean perimario.

A.7 Responsabilidades de proveedor del bean en EJBTaF

En esta sección describiremos las responsabilidades, adicionales a las que se especifican en [10], que tendrá el proveedor del bean para asegurar que los bean tolerantes a fallas puedan ser desplegados en un EJB ContenedorTF.

Contratos, clases e interfaces

El proveedor del bean es responsable de proveer las siguientes clases e interfaces:

- Las clases del bean primario y bean respaldo
- Las interfaces remote interface y remote home interface, si el cliente es un cliente remoto o un cliente local.

Las clases beans

Las siguientes son requerimientos (adicionales a los requerimientos que se marcan en [10] capítulos 7,10 y 12) para las clases de los bean primario y respaldo.

- La clase debe de implementar directamente o indirectamente, la interfaz según sea el caso.

```
javax.ejb.SessionBeanTF
javax.ejb.EntityBeanTF
```

- Se implementarán dos clases idénticas que sólo cambiarán por el nombre, ambas tendrán casi el mismo nombre excepto por que una de ellas (la réplica) al final le agregará el número 2, ejemplo; banco y banco2, donde banco2 es la réplica de banco.

ejbCambioEst< METODO>

La clase del bean implementará el método `ejbCambioEst()` que regresará los cambios sufridos en el estado de un bean después de haber ejecutado una invocación. El método sigue las siguientes reglas:

- Se debe de declarar `public`
- El método no es declarado como `final` o `static`
- El método regresará un *vector* el cual contendrá las variables que cambiaron en el estado de bean, así como su valor.

ejbCreate()<METODO>

Su implementación será según la especificación de los capítulos 7.10.3 y 10.6.3 de [10]

Métodos de negocio

Los métodos de negocios seguirán las reglas que se siguen en [10] para cada uno de los casos (sea un bean de sesión o un bean de entidad).

Remote interface

El proveedor de beans debe de proveer las interfaces remotas de los métodos de negocios (remote interface). Los siguientes son requisitos para la interfaz remota de los métodos de negocio.

- La interfaz debe de extender la interfaz
  `javax.ejb.EJBObjectTF`
- Los métodos definidos en esta interfaz deben de seguir las reglas para RMI/IIOP. Esto significa que sus argumentos y valores de retorno son tipos validos para RMI/IIOP, y su cláusula `throws` deben de incluir java.rmi.RemoteException.
- Para cada método definido en la interfaz remota, debe de haber un método que coincida con el método en la clase del bean. La coincidencia debe de ser:
  - El mismo nombre
  - El mismo número y tipo de argumentos, y el mismo tipo de valor de retorno.
  - Todas las excepciones definidas en la cláusula `throws` del método en la interfaz deben de coincidir con las que se encuentran en la clase de bean.
- Además de agregar los requisitos que son previstos en [10] según sea el caso de bean de sesión o bean de entidad.

Remote home interface

El proveedor de beans debe de proveer las interfaces remotas (remote home interfece) con la cual se pueda crear y borrar instancias de bean. Además, esta interfaz contará con métodos que hagan la búsqueda de otros métodos en el bean.

Los siguientes son requisitos para la interfaz:

- La interfaz debe de extender la interfaz javax.ejb.EJBHomeTF
- Los métodos definidos en esta interfaz deben de seguir las reglas para RMI/IIOP. Esto significa que sus argumentos y valores de retorno son tipos validos para RMI/IIOP, y su cláusula `throws` deben de incluir java.rmi.RemoteException.
- Una remote home interface debe de definir uno o más métodos create().
- Cada método de creación debe de ser llamado " create<METODO>",que debe de coincidir con el método "ejbCreate<Metodo>" definido en la clase del bean, con el mismo número y tipo de argumentos.
- Además de agregar los requisitos que son previstos en [10] según el caso en que use bean de sesión o bean de entidad.

A.8 Responsabilidades del proveedor del Contenedor en EJBTaF

En ésta sección se describe las responsabilidades del proveedor del ContenedorTF para soporta los beans tolerantes a fallas.

El ContenedorTF es responsable de:

- Los beans que no implementen las interfaces tolerantes a fallas deberán de ser manejador con es especificado en [10.]
- Proveer las herramientas de desplegado que permitan colocar las réplicas de los beans tolerantes a fallas, de manera transparente, en un lugar adecuado.

- Proveer de manera transparente el manejo del estado, en tiempo de ejecución, de las instancias de los bean tolerantes a fallas.
- Proveer la implementación de las clases Manejador de Tolerancia y Manejador de Bitácora que serán parte del ContenedorTF.
- Por último, proveerá la implementación de los diagramas mostrados en la sección 5.6, las cuales utilizan las interfaces provistas por el proveedor del bean (EJBObject TF y EJBHomeTF) para soportar fallas.

Recordemos que el ContenedorTF se propone como una extensión del Contenedor y seguirá teniendo las mismas responsabilidades en cuanto a manejo de persistencia, manejo automático de los estados del bean, manejo de transacciones y seguridad, mostradas en la referencia [10] más las responsabilidades arriba mencionadas.

A.9 Implementación de clases

Las clases ManejadorDeTolerancia y ManejadorDeBitácora por ser parte del ContenedorTF, serán implementadas por el proveedor de contenedor.

Las herramientas para desplegar que son provistas por el ContenedorTF son responsables de la generación de las clases adicionales. Las herramientas obtienen la información necesaria para generar las clases adicionales del descriptor desplegador y de las clases e interfaces provistas por el proveedor de beans.

El desplegador deberá crear las siguientes clases:

a) Una clase que implemente la interfaz Manejador de Tolerancia (la clase ManejadorDeTolerancia)
b) Una clase que implemente la interfaz Manejador de Bitácora (la clase ManejadorDeBitácora)
c) Una clase que implemente la interfaz *remote home interface* tolerante a fallas (clase EJBHomeTF)
d) Una clase que implemente la interfaz *remota interface* tolerante a fallas (clase EJBObjectTF)

A.9.1 Clase ManejadorDeTolerancia

La clase ManejadorDeTolerancia, es generada por el proveedor del ContenedorTF, quien implementa la interfaz

```
javax.ejb.ManejadorDeTolerancia
```

A.9.2 Clase ManejadorDeBitácora

La clase ManejadorDeBitacora, es generada por el proveedor del ContenedorTF, quien implementa la interfaz

```
javax.ejb.ManejadorDeBitácora
```

A.9.3 Clase EJBHomeTF

La clase EJBHomeTF, es generada por las herramientas del desplegador (deployment), quien implementa los métodos de la interfaz

`javax.ejb.EJBhomeTF`

A.9.4 Clase EJBObjectTF

La clase EJBObjectTF, es generada por las herramientas del desplegador, quien implementa los métodos de la interfaz

`javax.ejb.EJBObjectTF`

A.9.5 Clase EJBMetadata

Las herramientas del desplegador son responsables de la implementación de la clase que provee meta-datos a la vista del cliente remoto. La clase debe de usar valores validos para el RMI y debe de implementar la interfaz `javax.ejb.EJBMetaData`

A.9.6 Manejo de instancias

El contenedor debe de asegurar que sólo un hilo de control pueda manipular a la vez a las instancias de los beans de sesión, mientras que las instancias de los beans de entidad pueden ser manipuladas por más de un hilo de control pero de forma serial.

A.9.7 Transacciones, seguridad y excepciones

El ContenedorTF debe de seguir las reglas que se tienen con respecto a las transacciones, seguridad, y manejo de excepciones, como se describen en el capítulo 17, 21 y 18, de [10] respectivamente.

A.10 Interfaces

A.10.1 Interfaz EJBHomeTF

public abstract Interfece EJBHomeTF
que extiende de `java.rmi.Remote`
paquete javax.ejb

La interfaz EJBHomeTF es una interfaz remota que es extendida de las interfaces Home de los enterprise bean que serán replicados para tolerar las fallas. El home interface es definido por el proveedor de los enterprise beans e implementados por el ContenedorTF.

El ContenedorTF al implementar la interfaz EJBHomeTF invocará métodos de las clases; Manejador de tolerancia y Manejador de Bitácora, para soportar la tolerancia a fallas. Además los métodos que debe de contener la interfaz y los detalles de los mismos son mostrados en [13].

El método "crear ()" que esta previsto en [10] y que es definido por el usuario, será implementado por el Contenedor TF de tal forma que la vista que tenga el cliente sobre el bean con respecto a la tolerancia a fallas será transparente.

A.10.2 Interfaz EJBObjectTF

public abstract interface EJBObjectTF
paquete javax.ejb

Para lograr una localización transparente, los beans que sean replicados para tolerar las fallas sólo soportarán interfaces remotas. La interfaz remota (remote interface) define los métodos de negocio que son invocados por el cliente que no se encuentra en el mismo sitio que el bean.

La interfaz remota es definida por el proveedor de los enterprise beans e implementada por el ContenedorTF. El ContenedorTF al implementar la interfaz EJBHome invocará métodos de las clases Manejador de tolerancia y Manejador de Bitácora, para soportar la tolerancia a fallas.

Los métodos que debe de contener la interfaz y los detalles de los mismos son mostrados en [14]. Además, también deberán de ser implementados los siguiente métodos:

TfGuardarInv	public void **tfGuardarInv**(string *invocación* , string *idCliente*) throws java.rmi.RemoteException Este método utiliza la clase Manejador de Tolerancia para guardar en la bitácora del bean respaldo la *invocación* que será ejecutada en el bean primario, junto con ella se guarda *idCliente*, que es una `string` en el cual se guarda la identidad del cliente. Sólo el Manejador de Tolerancia que se encuentra en el sitio del bean primario podrá hacer uso de este método.
TfGuardarCambEdoRes	void **tfGuardarCambEdoRes**(string *cambEdo*, string *res*) throws java.rmi.RemoteException Este método utiliza la clase Manejador de Tolerancia para guardar en la bitácora del bean respaldo los cambios en el estado (*cambEdo*) que se generaron en el bean primario después de haber ejecutado una invocación y el resultado dado de la invocación (*res*). Sólo el Manejador de Tolerancia que se encuentra en el sitio del bean primario podrá hacer uso de este método.

Tabla A-3

A.10.3 Interfaz ManejadorDeTolerancia

public abstract interface ManejadorDeTolerancia
que extiende de `java.rmi.Remote`
paquete javax.ejb

	cliente (*idCliente*). Si éstas son iguales, significa que ha ocurrido una falla en el bean primario y el Manejador de Tolerancia tomará las medidas descritas en la Fig. A-2.

Tabla 5-4

El protocolo que seguirán los métodos para comunicarse con las demás clases del ContenedorTF es el mostrado en el capítulo 4.

A.10.4 Interfaz ManejadorDeBitácora

public abstract interface ManejadorDeBitácora
que extiende de `java.rmi.Remote`
paquete javax.ejb

El proveedor de **ContenedorTF** es el responsable de implementar los métodos de esta interfaz en la clase **ManejadorDeBitácora** que será parte de este contenedor.

La clase **ManejadorDeBitácora** es la encargada de asignar, dar formato, administrar y liberar el área de memoria primaria que es utilizada para la bitácora de cada uno de los beans que se ejecuten en el mismo sitio que ella. Además, también es la encargada de crear y administrar una área en memoria persistente donde se guardarán las claves de todas las instancias de los bean que se encuentren en ejecución en el sitio.

La clase **ManejadorDeBitacora** es una clase auxiliar de la clase **ManejadorDeTolerancia** para soportar la tolerancia a fallas, que cuenta con los siguientes métodos:

MbBuscarInv	boolean **mbBuscarInv**(string *invocación*, string idCliente) La invocación de este método indica a la clase Manejador de Bitácora que debe de buscar en la bitácora del bean un registro que coincida con la variable *invocación* y la variable *cliente*. De encontrar la clase dicho registro, el método regresará un valor de verdadero, de lo contrario regresará un valor de falso.
mbGuardarInv	void **mbGuardarInv**(string *invocación*, string *idCliente*) Este método indica al Manejador de Bitácora que guarde en la bitácora del bean las variables *invocación* y *idCliente*.
mbGuardaRes	void **mbGuardaRes**(string *res*) Este método indica al Manejador de Bitácora que guarde la variable *res* que corresponde al resultado a la última invocación que se guardó en la bitácora.
mbGuardarCambEdoRes	void **mbGuardarCambEdoRes**(object *cambEdo*, string *res*)

	Este método indica al Manejador de Bitácora que guarde las variables *cambEdo* y *res* que corresponden al cambio de estado y el resultado del bean primario después de la ejecución de la última invocación que fue guardada en la bitácora.
mbCrearBitacora	void **mbCrearBitacora**() Asigna y da formato al área que será utilizada como bitácora del bean. El formato de la bitácora deberá de contar con registro que a su vez tendrán campos donde se guardarán variables de tipo string. En los cuales se guarda la invocación, la identidad de cliente, la identidad de que hace la invocación, los cambios de estado que sufra el bean y el resultado de la invocación.
mbGuardarPersis	void **mbGuardarPersis** (string *idBean*) Guarda en la bitácora que se encuentra en memoria persistente la variable *idBean*, en la cual se encuentra guardado el tipo de bean que ha sido creado. Para que en caso de fallas se cree nuevamente.
mbRemoverPersis	void **mbRemoverPersis** (string *idBean*) Borra de la bitácora que se encuentra en memoria persistente la variable *idBean*, en la cual se encuentra el tipo de bean que ha sido removido. Para que en caso de fallas no se intente crear nuevamente.
mbRecuparar EdoInv	object **mbRecuparar EdoInv** () Al invocar este método se le indica al Manejador de Bitácora busque y entregue el ultimo estado del bean que se encuentre en la bitácora junto con las invocaciones que no se hayan aun ejecutado. El método regresará el estado y las invocaciones en una variable de tipo vector, donde el primer elemento será otro vector donde se encontrarán todas las variables del estado, seguido de este vector se encontrarán las invocaciones en el orden en que fueron guardadas en la bitácora.

Tabla A-5

El protocolo que seguirán estos métodos para comunicarse con las demás clases del ContenedorTF es el mostrado en el capítulo 4.

A.10.5 Interfaz SessionBeanTF

public abstract interface SessionBeanTF
que extiende de java.EnterpriseBean
paquete javax.ejb

El proveedor de bean es el responsable de implementar los métodos de esta interfaz en todos los beans de sesión que sean creados para tolerar fallas.

Los métodos y los detalles que contiene esta interfaz son los mismos mostrados en [16], agregando el siguiente método:

ejbCambEdo()	object **ejbCambEdo**() Este método compara los valores de variables de estado que tiene el bean antes de la ejecución de la invocación, con los que los valores de las mismas variables que tienen después de la ejecución, y regresa un vector que contiene el nombre y valores de todos aquellas variables que hayan sufrido un cambio.

Tabla 5-6

A.10.6 Interfaz EntityBeanTF

public abstract interface EntityBeanTF
que extiende de java.EnterpriseBean
paquete javax.ejb

El proveedor de bean es el responsable de implementar los métodos de esta interfaz en todos los beans de Entidad que sean creados para tolerar fallas.

Los métodos y los detalles que contiene esta interfaz son los mismos mostrados en [15], agregando el siguiente método:

ejbCambEdo()	object **ejbCambEdo**() Este método compara los valores de variables de estado que tiene el bean antes de la ejecución de la invocación, con los que los valores de las mismas variables que tienen después de la ejecución, y regresa un vector que contiene el nombre y valores de todos aquellas variables que hayan sufrido un cambio.

Tabla 5-6

A.11 Sumario

Recordemos que EJBTaF es una especificación que agrega tolerancia a fallas al modelo Enterprise Java Beans (EJB). El modelo EJB es un modelo para crear middleware a componentes multiplataformas, lo cual, es heredado por EJBTaF.

Tipo de modelo

EJBTaF sigue un modelo a componentes, el cual describe la creación, identidad, búsqueda y operación de un componente en un ambiente distribuido.

Descripción de componentes

Los Enterprise JavaBeans son componentes de software que representa la parte funcional (o lógica de negocio) que se ejecutan en un entorno llamado "contenedor" el cual los aísla del mundo exterior. Los beans son accedidos por las aplicaciones cliente haciendo uso de las interfaces EJBHome y EJBObject.

En EJB existen tres tipos de componentes: de identidad, de sesión sin estado y de sesión con estado. Esta división se hace tomando en cuenta los servicios que los componentes que prestan a los clientes y los servicios automáticos que requieren del contenedor.

Separación entre interfaces e implementación

Los componentes en EJBTaF sólo son accedidos por los clientes haciendo uso de sus interfaces y nunca de manera directa.

En EJBTAF para obtener los componentes las clases deben de heredar directa o indirectamente de javax.ejb.SessionBeanTF, javax.ejb.EntityBeanTF, javax.ejb.EJBObjectTF, javax.ejb.EJBHomeTF según sea el caso, ver sección 4.5. La obtención de estas herramientas es debe a la clara división que existe entre los tipos de componentes.

Servicios

Los servicios[1] entregados EJBTaF de manera automática a los programadores de componentes son: ocultamiento de los detalles de la red, ciclo de vida del componente, seguridad, persistencia, transacción, notificación de eventos (estos heredados de EJB) y ocultamiento de los detalles tolerancia a fallas. Además, el programador cuenta con las herramientas para empaquetar y desplegar componentes.

Tolerancia a fallas

Interfaces. Para lograr la tolerancia a fallas, se propone modificar algunas interfaces y se agregan algunas otras. En general se propone que en el modelo EJB existan las interfaces para el: Manejo de mensajes, Manejo de bitácora, Manejo de réplicas y mecanismos de recuperación. Además, se propone una interfaz para obtener el estado del objeto, el cual será posible guardar en la bitácora.

[1] Algunos de estos servicios pueden ser dejados como responsabilidad del programador haciendo uso de las herramientas e interfaces adecuadas.

Mecanismos. EJBTaF usa los siguiente mecanismos para lograr la tolerancia a fallas:

- Replicación pasiva (1-tolerante)
- Bitácoras y puntos de verificación
- Detección y notificación de fallas
- Manejo de consistencia

Transparencia de fallas

nivel N3 (ver sección 4.2.1)

Tipo de fallas soportadas. EJBTaF pretende tolerar las fallas (1-tolerante) de tipo Caída-amnésica, Caída-parcial-amnésica, Caída-pausa y falla de omisión, y dejando la posibilidad de que la especificación pueda evolucionar sin cambios drásticos para ser n-tolerante, donde $n \geq 2$.

En esta especificación se incorpora el mecanismo (no provisto en las especificaciones estudiadas) de monitoreo Push, con el cual, el componente que sufrió la falla inicia su recuperación.

EJBTAF			
Tipo de middleware			A componentes
Plataforma			Multiplataformas
Complejidad para construir grandes aplicaciones			Media
Optimizar, agregar, o eliminar servicios			No es permitido
Forma en que los objetos o componentes obtienen los servicios			De manera automática
Tolerancia a fallas	Tipo de fallas		Caída-amnésica, Caída-parcial-amnésica, Caída-pausa y falla de omisión
	n-tolerante		n = 1
	Mecanismos	monitoreo	Pull y Push
		redundancia	Pasiva
		Recuperación	Cuando un componente falla su estado se recupera de una bitácora

Tabla A-7 EJB Tolerante a Fallas

REFERENCIAS

[1] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, "Fundamentos de Bases de Datos", Mc Graw Hill, Tercera edición, 1998

[2] Andrew S. Tanenmbaum, "Sistemas Operativos Distribuidos", Prentice Hall, Primera edición, 1996.

[3] Arun K. Somani, Nitin H. Vaidya, "Understanding Fault Tolerance and Reliability", IEEE Computer, Abril 1997.

[4] Componet Object Model (CCM) especitication 0.9. http://msdn.microsoft.com/library/default.asp?URL=/library/specs/S1cf80.htm.

[5] Christian F., "Understanding fault-tolerant distributed systems", Communication, ACM, 1991.

[6] Chris Crenshaw, "The Deleveloper`s Guide to Understanding Enterprise JavaBeans Applications", NOVA Laboratories, www.nova-labs.com, 1999

[7] D.E. Comer, D. Stevens, "Internetworksing with TCP/IP", VOL. III. Prentice Hall, 1999

[8] DCOM/1.0 : http://www.globecom.net/ietf/draft-brown-dcom-v1-spec-03.html

[9] Elizabeth Pérez Cortés, "Tolerancia a Fallas en los Sistemas Distribuidos", Conferencia, Cartagena de Indias, Colombia, Mayo 2001.

[10] Enterprise JavaBeans 2.0 Proposed Final Draf 2, 4-24-01: http://java.sun.com/products/ejb/docs.html

[11] Fault-Tolerance CORBA Specification, V1.0, Abril 2000, OMG Documento ptc/2000-04-04.

[12] Felix c. Gärtner, "Fundamentals of Fault-Tolerant Distributed Computing in Asynchonous Enviroments", ACM Computing Surveys, Vol.31, No. 1, Marzo 1999.

[13] Interface EJBHome: http://java.sun.com/products/ejb/javadoc1.1/javax/ejb/EJBHome.html

[14] Interface EJBObject: http://java.sun.com/products/ejb/javadoc1.1/javax/ejb/EJBObject.html

[15] Interface EntityBean: http://java.sun.com/products/ejb/javadoc1.1/javax/ejb/EntityBean.html

[16] Interface SesionBean: http://java.sun.com/products/ejb/javadoc1.1/javax/ejb/SesionBean.html

[17] Java Naming and Directory Interface: http://java.sun.com/products/jndi

[18] Java Transaction API: http://java.sun.com/products/jta

[19] Juan Hernández, José M. Troya, Antonio Vallecillo, "Lesson 3: Component Interoperability", Dep. Informática Universidad de Extremadura, España 2001.

[20] K. Robbin, "Practical Unix Programming", Prentice Hall, 1996

[21] Kurt Geihs, "Middleware Challenges Ahead", Goethe University, IEEE Computer, Junio 2001.

[22] Lamport L., "Time, clock, and the ordering of event in a distributed system", Communications of the ACM, 1978.

[23] Lidia Fuentes, José M. Troya, Antonio Valecillo, "Lección 1: Desarrollo de Software Basado en componentes", Dept. Lenguajes Y Ciencias de la Computación, Universidad de Málaga, España 2001.

[24] M. Wiesmann, F. Pedone, A. Achiper, B. Kemme, G. Alonso, "Understanding Replication in Database and Distributed Systems", IEEE 1063-6927/00,2000

[25] Nell K. Jain, "Group Fromation Mechanisms for Transaction in Isis", ACM, 1994

[26] OMG. CORBA 2.5 Specification. Object Management Group, Septiembre 2001, OMG TC Document formal/2001-09-34.

[27] OMG. CORBA Component Model Tutorial. Objet Management Group, Junio 2002, OMG TC Document ccm/2002-11-03.

[28] OMG Updated CCM specifacation. Objet Management Group, Noviembre 2001, OMG TC Document pct/2001-11-03

[29] Ofarli, Robert Harkey, Dan. Edwards, Jeri., "The Essential Distributed Objetcs Survival Guide", ISBN 0-471-12993-3, the United States of American: John Wiley & Sins, Inc, 1996.

[30] Peter Michael Melliar-Smith, Louise Elizabeth Moser, "Surviving Network Partitioning", IEEE Computer, Marzo 1998.

[31] R. Marvie y P. Merle, CORBA Component Model: Discussion and Use whith OpenCCM. Special Issue of the Informatica- An International Journal of Coputing and Informatics Dedicated to "Component Based Software Development", Junio 2001.

[32] R. Marvie, P. Merle y J. M. Geib, CORBA Component Model: Towards a Dynamic CORBA Component Platform In Proceedings of the 2nd International Symposium on Distributed Object Aplications (DOC '2000), Anvers, Bélgica, Septiembre 2000, IEEE.

[33] Shiper A., Guerraoui R., "Software-based replication for fault tolerance", IEEE Computer, 1997.

[34] The COM Client/Server Model: http://msdn.microsoft.com/library/en-us/com/comext_8p2r.asp.

[35] Edward Cobb, Dave Frankel, Dave Curtis, Patrick Thompson, "Abstract Component Model" March 23, 1999, orbos/99-03-22

[36] Object Collection Specification, V1.0.1, Agosto 2002, OMG documento formal/02-08-03

[37] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments", Cambridge.

[38] Christian F. "Exception handing and software fault tolerance". Transactions on Computers, 1982.

[39] Weihai Yu, "Introduction to CORBA", Univ. Of Troms ∅, Lecture Handouts for D441S, Primavera 2002.

[40] Vania Marangozova, Daniel Hagimont, "Non-functional replication management in the CORBA Component Model", INRIA Rh ône-Alpes, Francia, 2002.

[41] Schneider F., "Implementing fault-tolerant services using the state machine approach", in ACM Computer Survive.

[42] Cristian F., "Exception handling and software fault tolerance", Transactions on computers, 1982.

[43] OMG. CORBAServices, Diciembre del 1998, Document formal/98-12-19.

[44] OMG. CORBA 2.4.1, "The Common Object Request Broker: Architecture and Specification", Noviembre 2000

[45] OMG. CORBA 3, "The Common Object Request Broker: Architecture and Specification", Noviembre 2002. Docment formal/02-11-03

[46] Vanneschi M., Baiardi F., "Desing of highly decentralized operanting systems", In Distributed Operating Systems.

[47] BEA **WebLogic** Platform Documents: www.edocs.bea.com/

[48] Eternal System: www.eternal-systems.com/

[49] R. Casallas, J. Arias y G. Vega, "Modelo Séneca": http://xue.uniendes.edu.co/~rcasalla/SénecaProject/Index.html . Bogotá, Colombia, 2002

[50] Duong Phuong-Quynh,"La tolérance aux fautes adaptable pour les sytèmes à composants", tesis de doctorado de l' NP Grenoble, spécialité : "Systèmes et Logiciels", Francia, 15 diciembre 2003