# Capítulo 9 - METODOLOGÍA DE DESARROLLO

La replicación de una aplicación es esencial para hacerla tolerante a fallos, pero esa replicación resulta cara de realizar. Además de la duplicación de recursos hardware que precisa, hay un impacto negativo en el rendimiento, ya sea en la carga de trabajo soportada, en el tiempo de respuesta de servicio o incluso en ambos parámetros.

Empleando replicación activa, las réplicas deben mantener su consistencia mutua continuamente, por lo que toda actualización en una réplica debe propagarse a las demás. El impacto de esta propagación depende del sistema de comunicaciones en grupo y su algoritmo empleado. El protocolo optimizado de *Totem* [Totem] requiere, por ejemplo, dos vueltas y media del testigo para entregar fiablemente un mensaje en orden total. SenseiGMS requiere para el mismo cometido media vuelta del testigo más el envío de tres mensajes al miembro más lento del grupo, aunque la eficiencia se incrementa cuando un miembro envía varios mensajes en la misma vuelta.

Y esta propagación sobre el grupo es requerida para cada actualización en la aplicación en condiciones normales. Si el grupo formado por las réplicas cambia su composición, pierde su disponibilidad mientras se negocian los cambios, y es difícil calcular el impacto o duración de este bloqueo. Por ejemplo, en SenseiGMS, un cambio debido a la inclusión o exclusión voluntaria de un miembro se realiza en dos vueltas de testigo, más el envío de un mensaje al miembro más lento del grupo. Pero

si hay una partición del grupo o se cae el miembro que posee el testigo, debe verificarse la regeneración del testigo, comprobarse el consenso, enviarse nuevas vistas, etc.

El coste de la replicación activa puede reducirse empleando replicaciones pasivas, donde la consistencia sólo debe mantenerse a intervalos dados de tiempo, o incluso nunca, en el caso más simple. Cuanto menor sea la consistencia, mayor es el tiempo requerido para recuperar el servicio en caso de que la réplica primaria falle. Y además, sigue existiendo un impacto continuo de rendimiento, pues esa réplica primaria debe aún guardar su estado en dispositivos persistentes. Sin embargo, la especificación del servicio de tolerancia a fallos de CORBA da un fuerte soporte al empleo de replicaciones pasivas y, aunque cubre el uso de grupos activos, su soporte puede resultar insuficiente para aplicaciones complejas, como se detalla en este capítulo.

Si el empleo de replicaciones pasivas supone mejores tiempos de respuesta en la mayoría de los casos, ofrece por otro lado una menor disponibilidad que puede ser prioritaria para determinadas aplicaciones como, por ejemplo, controladores de dispositivos esenciales en un avión. Y si las aplicaciones pueden asegurar una buena calidad de las comunicaciones entre sus réplicas, el coste de la replicación activa puede reducirse a márgenes perfectamente aceptables para tales aplicaciones.

Por ejemplo, un servidor conectado a Internet, dando un determinado servicio mediante *Web*, puede disponer de réplicas situadas geográficamente de tal forma que ningún cliente deba acceder a servidores físicamente muy distantes, lo que supondría peores tiempos de respuesta. Y si la conexión entre las réplicas no se hace mediante Internet, sino en una red con una calidad de servicio fija y elevada, los tiempos requeridos para actualizar todas las réplicas pueden ser adecuados, además de minimizar las probabilidades de indisponibilidad del grupo por cambios en su composición.

Para estos casos donde la red soporta unos parámetros de calidad de servicio adecuados, es también posible emplear protocolos de comunicaciones optimizados, como el desarrollado en *Spinglass* [Spinglass], redundando en un mejor servicio del grupo.

Entendidas las limitaciones asociadas al modelo de replicación activa, este capítulo se centra en la programación de aplicaciones tolerantes a fallos bajo ese modelo. Empleando replicación pasiva, es preciso modificar la aplicación para que almacene su estado de forma periódica, pero este cambio resulta mínimo al compararlo con los cambios necesarios para emplear replicación activa. Es posible emplear diversas técnicas que simplifiquen el desarrollo de aplicaciones replicadas activamente, por lo que nos centramos en el soporte necesario para implementar esas técnicas; en particular, estudiamos el soporte ofrecido en CORBA, detallando sus insuficiencias para aplicaciones no básicas.

Los problemas de rendimiento en grupos activos pueden precisar de algoritmos fuertemente optimizados y específicos para una aplicación concreta. Sin embargo, el enfoque seguido en este capítulo es generalista, por lo que nos basamos en el empleo de orden total causal en las comunicaciones. Es decir, todas las réplicas reciben los mismos eventos en el mismo orden, lo que permite diseñar algoritmos más generales y simples que empleando órdenes menos estrictos.

## 9.1. Sincronización de la respuesta

En una aplicación cliente/servidor, el servidor define una interfaz que el cliente emplea para solicitar sus servicios. Generalmente el cliente queda bloqueado mientras el servidor procesa su solicitud, aunque esto no es siempre cierto, como puede[5] ser el caso de las operaciones *oneway* en CORBA. En una aplicación fiable, este servidor debe contactar a las demás réplicas y obtener así una respuesta consensuada, que es entonces entregada al cliente; este proceso implica en general la necesidad de sincronizar la solicitud del servicio con la elaboración de la respuesta.

Este problema fue brevemente esbozado en el capítulo anterior, al desarrollar un ejemplo de uso con SenseiGMS, y mostrado gráficamente en la figura 8.2. Ahora emplearemos una interfaz muy sencilla, con una única operación que muestra el patrón general de diseño a emplear; esta interfaz es la de un servidor que genera números únicos de forma secuencial, especificado en OMG/IDL como:

```
interface NumberGenerator
{
   long getNumber();
};
```

Una posible implementación de este servidor replicado se basa en que cada réplica mantenga una variable con el último número generado por el grupo. A esta variable la denominamos *lastGeneratedNumber*. Cuando una de estas réplicas recibe una petición *getNumber*, realiza los siguientes pasos:

- Envía un mensaje al grupo, de tal forma que todas las réplicas tengan conocimiento de que está en curso una petición.
- Las réplicas, al recibir el mensaje previo, incrementan automáticamente la variable que almacena el último número generado. Este paso también lo realiza la réplica que envió el mensaje.

---

[5] El estándar CORBA [OMG98] no especifica que la invocación de una operación *oneway* deba impedir el bloqueo del cliente en tanto el servidor la procesa. Sin embargo, un comportamiento no bloqueante resulta una implementación lógica y generalizada.

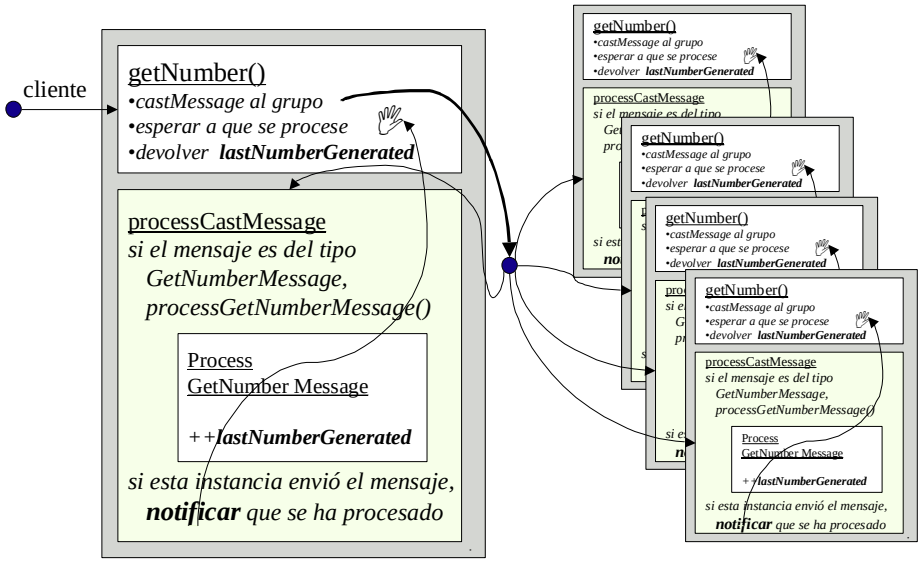


Figura 9.1. Sincronización de la respuesta

- Cuando la réplica que envió el mensaje lo ha procesado, puede ya devolver una respuesta al cliente, el valor contenido en *lastGeneratedNumber*.

Luego la réplica que envía el mensaje debe esperar la notificación de que ha sido procesado; la figura 9.1, similar a la figura 8.2, la extiende mostrando la sincronización requerida e incluye además en pseudo-código el escenario descrito.

Es importante destacar que la secuencia previa de pasos no es correcta. Bajo una implementación general de *threads* o hilos de control, no es posible asumir que un determinado *thread* en estado de espera que pasa a estado activo reciba inmediatamente tiempo de proceso. Por ello, es perfectamente posible que desde el momento en que se realiza la notificación de que el mensaje ha sido procesado hasta que se lee el valor de la variable *lastGeneratedNumber*, la réplica procese otro mensaje que modificaría de nuevo esta variable, devolviendo consecuentemente un valor erróneo.

La figura 9.2 muestra una alternativa a la anterior sincronización, más complicada pero necesaria en todas las operaciones que devuelven algún valor, sea como parámetro de salida o como valor de retorno:

- La réplica recibe en un *thread* la petición *getNumber*. Envía un mensaje al grupo, de tal forma que todas las réplicas tengan conocimiento de que una petición está en curso.
- Espera a que el mensaje se reciba, con lo que ese *thread* (*thread* de envío) queda suspendido.

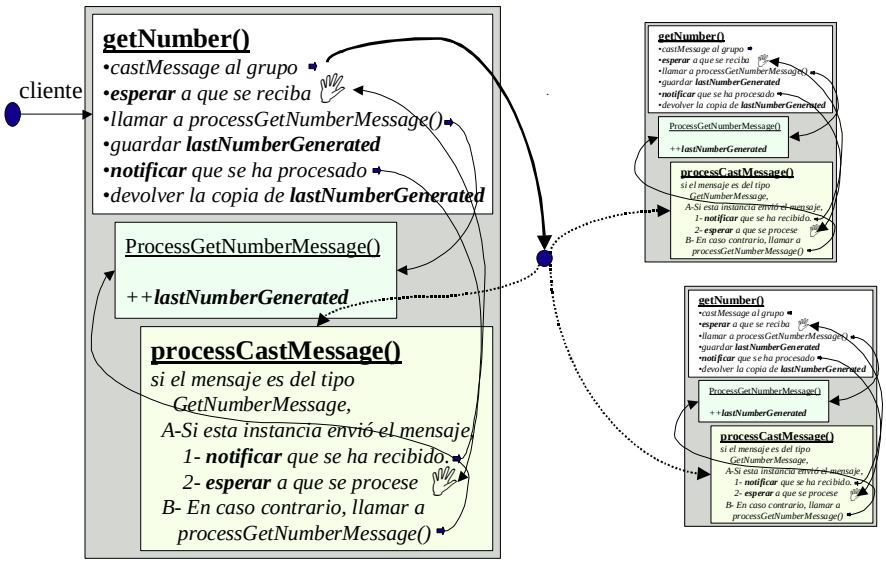


Figura 9.2. Alternativa para la sincronización de la respuesta

- El mensaje se recibe en un *thread* independiente (*thread* de recibo); en la réplica que envió el mensaje, este mensaje debe procesarse en el *thread* de envío. Por ello, se despierta a ese *thread*, y este *thread* de recibo pasa a estado suspendido. Las demás réplicas procesan el mensaje directamente en el *thread* de recibo.
- Cuando el *thread* de envío despierta, procesa el código asociado al mensaje, incrementando el valor de la variable *lastGeneratedNumber*, que será devuelta al cliente. A continuación, despierta al *thread* de recibo, que no debe hacer ya nada más. El *thread* de envío termina también al completarse la petición *getNumber*.

Es necesario en este escenario que el *thread* de recibo no finalice hasta que se haya procesado el mensaje en el *thread* de envío; en caso contrario, la réplica podría recibir nuevos mensajes, con lo que estaría procesando operaciones en paralelo, con resultados imprevistos salvo que incluyera sus propias opciones de sincronización, lo que complica de nuevo el escenario.

En este segundo caso, cuando una réplica recibe un mensaje que no envió, lo procesa como en el primer caso, sin requerir ninguna sincronización. En los dos escenarios previos, hay que considerar la situación en que una réplica es expulsada del grupo mientras está procesando una petición de servicio. En ese caso, los *threads* que estén suspendidos deben desbloquearse, devolviendo alguna excepción al cliente.

## 9.2. Transformación de operaciones en mensajes

En el ejemplo del anterior apartado se envía un mensaje en respuesta a la petición de operación del cliente. Como ya se vio en el anterior capítulo, ésa es la estrategia general: cada mensaje incluye la información necesaria para poder ejecutar en cada réplica el mismo código. Esta aproximación implica definir un mensaje para cada operación, que define como atributos cada uno de los parámetros de entrada definidos en la operación asociada.

Es necesaria una extensión a lo descrito en el anterior capítulo para permitir que una misma réplica pueda procesar varias operaciones concurrentemente. Para cada una de esas operaciones se envía un mensaje y su *thread* asociado se bloquea hasta que el mensaje se reciba o se procese. Por lo tanto, es necesario asociar cada mensaje al *thread* que lo envía, y por eso cada mensaje incluye una identidad fácilmente asociable a ese *thread*. Usando JavaRMI, esta identidad podría ser directamente la asignada por Java al *thread*, pero en CORBA debe ser independiente del lenguaje de programación final. Escogemos un valor entero para identificar al *thread*, y el algoritmo de envío de mensajes y bloqueo de *threads* debe ser capaz de asignar a cada *thread* una identidad única. Es importante notar que esta identidad sólo tiene sentido para la réplica que envía el mensaje, por lo que se puede asignar una misma identidad a mensajes provenientes de distintas réplicas.

Con CORBA, todos los parámetros *in* y *inout* deben incluirse en el mensaje. La operación *double obtainPercentage (in string key, inout long values, out boolean error)* se transforma en el mensaje:

```
valuetype ObtainPercentageMessage : Message
{
        public long id; //identidad del mensaje
        public string key; //parámetro de entrada
        public long values; //parámetro de entrada
};
```

No es necesario incluir información de los parámetros de salida, pues no es preciso que las demás réplicas devuelvan esa información. En el caso de JavaRMI, todos los parámetros son de entrada, luego el mensaje incluye todos los parámetros[6] de la operación.

Esta transformación de operaciones en mensajes es fácilmente automatizable, y una herramienta puede generar los mensajes necesarios a partir de la definición de las interfaces. Este proceso de automatización es extensible al propio servidor

---

[6] Que los parámetros sean de entrada, no implica que no puedan tratarse como parámetros de salida. Por ejemplo, es posible enviar a un método Java una instancia *StringBuffer* que únicamente se emplee para devolver una cadena de caracteres. En este caso, no es necesario incluir el atributo asociado en la definición del mensaje.

replicado: es posible crear todo el código y clases necesarias para definir un objeto replicado a partir de la definición de su interfaz y una implementación no replicada de ese servidor.

Para entender esta automatización, partimos de una clase *NumberGeneratorImpl* que implementa sin ningún soporte de replicación la interfaz *NumberGenerator* definida en el apartado anterior. El primer paso es la generación de un mensaje para la operación *getNumber*. Este mensaje no contiene ningún atributo asociado a la operación, sólo la identidad de mensaje:

```
valuetype GetNumberMessage : Message
{
        public long id;
};
```

Se genera también una clase *GroupNumberGeneratorImpl* que implementa igualmente la interfaz *NumberGenerator* e incluye toda la lógica de grupo necesaria. La lógica de aplicación se delega a la instancia de la clase *NumberGeneratorImpl*, con lo que la operación *getNumber* resulta ahora en Java:

```
int getNumber() throws InvalidGroupException
{
        int id = getRequestId();
        Message message = new GetNumberMessage(id);
        castMessageAndBlockUntilReceived(message);
        int ret = theNumberGeneratorImpl.getNumber();
        messageProcessed(id);
        return ret;
}
```

Las operaciones en cursiva se explican posteriormente; este código realiza los siguientes pasos:

- Obtiene una identidad única para la operación en curso.
- Crea el mensaje asociado a la operación, incluyendo la identidad obtenida.
- Envía el mensaje al grupo y se bloquea hasta que se reciba el mensaje.
- Se invoca la operación sobre la instancia no replicada del servidor.
- Se comunica que la operación ha sido procesada, tal como se describió en el anterior apartado.
- El valor devuelto por la instancia no replicada es el valor que se devuelve al cliente del grupo.

El código de recepción del mensaje sigue las pautas dadas anteriormente:

```
void processCastMessage(int sender, Message message)
{
        if (message instanceof GetNumberMessage) {
                GetNumberMessage msg = (GetNumberMessage) message;
                if (sender==myself) {
                        unblockThreadAndWait(msg.id);
                }
                else {
                        theNumberGeneratorImpl.getNumber();
                }
        }
        else
                .  .  .  .
}
```

- Se verifica cada uno de los tipos de mensaje esperado.
- Si la réplica que recibe el mensaje es la que lo envió, despierta el *thread* asociado y se bloquea hasta que ese *thread* concluye.
- En caso contrario, se invoca directamente la operación sobre la instancia no replicada. Si esta operación precisa de parámetros de entrada, sus valores se leen del mensaje recibido[7].

Esta automatización emplea el segundo escenario descrito para la sincronización de la respuesta en el anterior apartado, incluso para operaciones que no devuelven valores, pues es válido para todo tipo de operaciones.

El código descrito ha empleado cuatro primitivas, comunes para cualquier servidor:

- *getRequestId:* devuelve una identidad única. Es fácilmente implementable a partir de una variable con tipo entero que se incrementa con cada llamada.
- *castMessageAndBlockUntilReceived:* envía el mensaje al grupo y bloquea al *thread* que lo llama hasta que se reciba el mensaje.
- *unblockThreadAndWait:* desbloquea al *thread* que tenga la identidad dada como parámetro y bloquea automáticamente al *thread* que lo llama.
- *messageProcessed:* desbloquea al *thread* asociado al mensaje especificado.

Esta automatización permite, por lo tanto, generar rápidamente un servidor replicado a partir de una instancia no replicada de ese servidor. El proceso descrito no es completo, ya que es necesario escribir manualmente el código asociado a las

[7] En C++, para los parámetros de salida deberían definirse variables temporales que se descartarían tras invocarse la operación. Lo mismo ocurriría en Java para los parámetros manualmente excluidos del mensaje, como explicó la anterior anotación.

operaciones de transferencia de estado. Empleando JavaRMI, es posible emplear sus mecanismos de serialización para implementar esa transferencia, pero con CORBA la implementación debe realizarse independientemente.

## 9.3. Comportamiento no determinista

A pesar de los beneficios aparentes de la automatización descrita en la sección previa, sólo las aplicaciones más sencillas podrán replicarse de esa manera. El anterior enfoque implica replicar el comportamiento de la aplicación a partir de su interfaz. Pero una interfaz define las operaciones a soportar, no la implementación o estructuras de datos necesarias. Si en el ejemplo anterior el generador de números únicos debiera devolver números aleatorios, su interfaz sería aún la misma:

```
interface NumberGenerator
{
   long getNumber();
};
```

Sin embargo, la implementación pasa a considerarse no determinista: la misma secuencia de eventos no produce la misma secuencia de resultados.

Internamente, la aplicación puede definir ahora una estructura de datos que incluya todos los números ya generados. Cuando se solicita un nuevo número, se obtiene uno al azar y se comprueba en la anterior estructura de datos si está libre o ha sido ya asignado; si no lo está, se puede generar otro o simplemente iterar en la tabla a partir del número anterior buscando el primer número libre. Esta tabla es evidentemente finita, pero obviamos este hecho para los propósitos de este ejemplo.

Al no haber cambiado la interfaz *idl*, el proceso de automatización produciría el mismo código detallado en la sección previa, lo que provocaría resultados incorrectos:

- Cuando una réplica recibe la solicitud *getNumber*, envía el mensaje *GetNumberMessage*.
- Cada réplica, al recibir este mensaje, invoca la instancia no replicada del servidor.
- Las distintas réplicas, si el algoritmo de aleatoriedad es bueno, producirán distintos números aleatorios, con lo que la consistencia de sus estados se pierde. Distintas réplicas consideran que los números asignados son distintos, con lo que producirán eventualmente números no únicos.

La solución a este problema es sencilla: la réplica que recibe la solicitud *getNumber* genera directamente un número aleatorio, que es incluido en el mensaje *GetNumberMessage*. Los demás miembros del grupo comprueban si ese número está libre y, si no es así, iteran sobre su estructura de datos interna hasta obtener el

siguiente número libre, pues generar otro número aleatorio produciría el mismo resultado erróneo. Pero a pesar de la sencillez de la solución, el problema es ya incompatible con la automatización descrita.

Este problema no se asocia sólo a comportamientos no deterministas. Si por ejemplo el servidor debe enviar el resultado de la operación a una página *Web* o, en general, actuar como cliente de un servidor diferente, sólo una de las réplicas debería efectuar la operación correspondiente. Este problema es inherente al servicio de tolerancia a fallos de CORBA, como detallamos al final de este capítulo (de hecho, ese servicio especifica claramente que las aplicaciones soportadas deben ser deterministas).

## 9.4. Replicación de componentes

Un comportamiento no determinista es uno de los casos que impiden la automatización propuesta en la replicación de un servidor. Otro caso ya señalado se da cuando ese servidor debe acceder a ciertos recursos externos; si por ejemplo un servicio debe acceder a un servidor GPS que devuelve la posición geográfica de un determinado localizador, y ese servidor cobra por cada acceso, el servidor a replicar debería evitar que todas las réplicas consultaran al servicio GPS.

En general, todo servidor cuya implementación dependa de otros servidores no podrá automatizarse con el método descrito. Y este problema persiste incluso si la dependencia se produce a nivel de interfaz. Por ejemplo, la clase *java.util.Vector* que implementa la interfaz *java.util.List* podría utilizarse para crear un contenedor replicado; sin embargo, esta clase devuelve en algunas operaciones *iteradores* que permiten observar los elementos contenidos en el *Vector*. Y esos iteradores deben modificarse para soportar las características replicadas del contenedor asociado, lo que implica modificar el código que produce esos iteradores para crear los nuevos con soporte de replicación.

Por lo tanto, la automatización del proceso de replicación es únicamente aplicable a servidores sencillos. Por otra parte, es posible identificar estructuras en el servidor que puedan replicarse automáticamente. Volviendo al ejemplo del generador de números únicos y aleatorios, se identifica fácilmente como replicable la estructura de datos interna que contiene los números ya generados. Esta estructura se programa separadamente con la siguiente interfaz:

```
interface NumberSet
{
        boolean set (in long Number);
};
```

La única operación definida, fija como asignado el número dado en el parámetro y devuelve *true* si el número estaba todavía libre (no asignado).

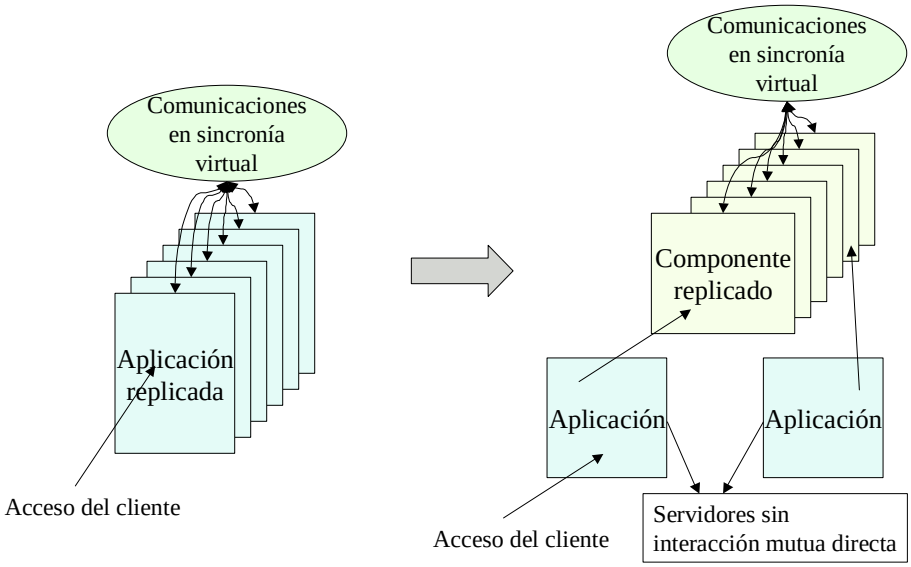


Figura 9.3. Enfoque en la replicación de componentes

El servidor no replicado se programa empleando una instancia de la interfaz *NumberSet*, denominada *theNumberSet* en el siguiente código, que debería ser mejorado para una situación real donde el contenedor de números no es infinito. No obstante, muestra la idea del algoritmo: iterar sobre el contenedor hasta encontrar una posición libre, comenzando desde una posición aleatoria:

```
int getNumber()
{
        int initial = getRandomNumber();
        while (!theNumberSet.set(initial)) {
                ++initial;
        }
        return initial;
}
```

La interfaz *NumberSet* sí es automáticamente replicable, con lo que sólo es necesario cambiar el código del servidor para instanciar y emplear la clase que implementa el comportamiento replicado. Este servidor, al recibir una petición *getNumber* no se comunica con sus réplicas sino que ejecuta el mismo código empleado en el servidor no replicado. Sin embargo, cada vez que ahora accede a *theNumberSet* emplea una estructura replicada que sí se comunica con sus demás réplicas.

La figura 9.3 muestra el cambio de enfoque de esta aproximación: no se replica el servidor completo, sino los componentes que lo precisan. Los servidores forman un grupo tolerante a fallos pero no interaccionan directamente entre ellos, sino a

través de esos componentes. La abstracción que esos componentes replicados crean es la de un único componente compartido (memoria compartida).

## 9.5. Librerías de componentes replicados

El lenguaje C++ [ISO98, Stroustroup97] se creó y extendió inicialmente sin una librería de contenedores, lo que provocó que cada programador diseñara e implementara sus propias soluciones para pilas, listas, árboles binarios, etc, o que se adhiriera a determinadas soluciones comerciales, pocas veces compatibles entre sí. La estandarización posterior del lenguaje incluyó una librería estándar de *templates* (STL) [ISO98, Plauger95], simplificando considerablemente este escenario. Un programador puede emplear distintas implementaciones sabiendo que son compatibles, y usar la que produzca mejores rendimientos. El empleo de servicios replicados está poco extendido a pesar de sus indudables beneficios. Y la causa no es únicamente el menor rendimiento que se consigue, sino la dificultad de su diseño. Si se dispone de librerías de componentes replicados, ese diseño se facilitará considerablemente.

El principal problema es entonces considerar qué componentes deben ser replicados. En general, el comportamiento de una aplicación depende del estado de sus datos. Esto no implica un determinismo, que empleando los mismos datos esa aplicación produzca siempre los mismos resultados, pero sí una consistencia de éstos. Consecuentemente, es comprensible enfocarse en los componentes que almacenan esos datos.

Cada aplicación define sus propias estructuras de datos, que pueden ser desde simple tipos existentes en el lenguaje, como enteros, a complejas estructuras, como un árbol binario que contenga a su vez listas de valores enteros. Sin embargo, es posible factorizar esas estructuras complejas en términos de los contenedores estándar.

El ejemplo mostrado en la sección anterior podría simplificarse si dispusiéramos en Java de un *java.util.Set* replicado o, hablando en términos C++, de un *std::set* replicado. Además de la simplificación que se obtiene en el diseño de servidores replicados, la principal ventaja es la optimización que puede y debe lograrse sobre esos componentes. SenseiUMA es la parte de este proyecto que trata el diseño de estos contenedores replicados. Este diseño está directamente basado en la colección de componentes disponibles en Java 1.2.

## 9.6. Soporte de concurrencia

Al enfocar la replicación sobre los componentes de una aplicación, es necesario estudiar los problemas de concurrencia que pueden aparecer. En una aplicación no

replicada, el empleo de múltiples *threads* complica la implementación de sus componentes para soportar el acceso concurrente desde esos *threads*. En una aplicación basada en componentes replicados, esos componentes pueden entenderse como objetos compartidos al que acceden las distintos aplicaciones del grupo, por lo que deben soportar directamente un acceso concurrente.

Por ejemplo, la interfaz *NumberSet* utilizada en la sección de replicación de componentes se diseñó de tal manera que permitiera acceso concurrente desde varios servidores. Si se hubieran seguido las reglas generales para escribir buen código, que dictan la definición de múltiples operaciones simples en lugar de complicadas operaciones multifunción [Maguire93], se habrían definido dos métodos claramente diferenciados:

```
interface NumberSet
{
        void set (in long Number);
        boolean test(in long Number);
};
```

Un método comprueba si un número ha sido asignado, y el otro lo asigna. Sin embargo, esta interfaz no podría haberse usado directamente para producir componentes replicados: dos servidores podrían comprobar simultáneamente si un mismo número estaba libre, obteniendo ambos una respuesta afirmativa y asignándolo erróneamente [figura 9.4]. Al juntar los dos métodos, se obtiene una operación atómica que soporta accesos concurrentes.

Esta definición de operaciones atómicas específicas no es posible al emplear componentes genéricos. En este ejemplo, el componente empleado sería un *Set*, definido como un contenedor de elementos que no admite duplicados, o un *BitSet*, definido como un contenedor numerado de valores booleanos. Basándonos en la clase *java.util.BitSet*, serían necesarias las dos operaciones siguientes, expresadas en Java como:

- *boolean get (int bitIndex);* Devuelve el valor del bit en el índice dado.
- *void set (int bitIndex);* Asigna al bit dado el valor sí/cierto/*true*.

Y tal como sucedía con las operaciones de la interfaz previa *NumberSet*, se observan los mismos problemas de concurrencia descritos. La solución en el caso de acceso concurrente bajo múltiples *threads* es el empleo de monitores: un *thread* debe adquirir un monitor antes de efectuar una operación crítica, liberándolo al concluirla. Distintos *threads* acceden al mismo monitor, lo que impide que ejecuten concurrentemente la operación crítica.

Por extensión, la solución para el problema de concurrencia en componentes replicados es el empleo de monitores replicados. Estos monitores deben soportar dos operaciones, con las siguientes semánticas:

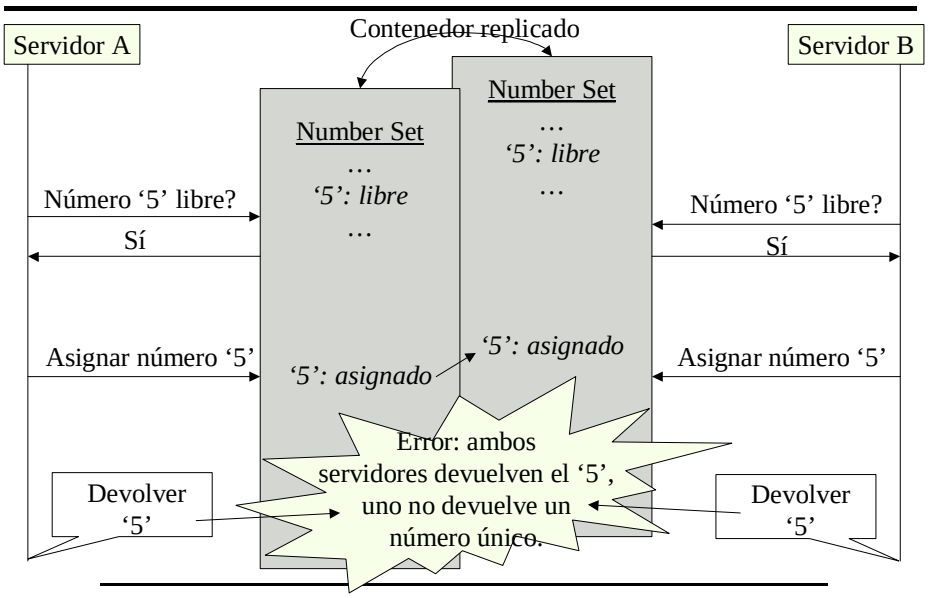


Figura 9.4. Problema de concurrencia en *NumberSet*

- *lock*: adquisición del monitor. Ante una adquisición concurrente desde varias réplicas, sólo una lo obtiene. Las demás réplicas que pretendan adquirir ese monitor quedan bloqueadas hasta que se libere el monitor o hasta que el componente que lo ha adquirido se caiga o sea excluido del grupo. Una vez que el monitor se libere, cualquiera de las réplicas bloqueadas puede obtenerlo, pasando a estado no bloqueado. Una misma réplica puede adquirir un monitor repetidas veces, es decir, el monitor es reentrante.
- *unlock*: libera un monitor previamente adquirido, produciéndose un error si la réplica no había adquirido el monitor. Si esa réplica hubiera adquirido el monitor múltiples veces, debe invocarse esta operación el mismo número de veces para liberar efectivamente el monitor,

Estas dos operaciones son suficientes si no se plantean condiciones de guarda (*wait/notify*), necesarias normalmente en aplicaciones *multithread* cuando diversos *threads* deben comunicarse entre sí.

Con esta definición, una réplica que adquiere un monitor puede obtenerlo repetidamente sin precisar liberarlo entre adquisiciones. Este monitor controla, de esta manera, regiones críticas entre diferentes réplicas, pero no entre diferentes *threads* de una misma réplica.

Empleando un monitor y un *bitset* replicados, el servidor *NumberSet* podría codificarse como:

```
number = getRandomNumber();
replicatedMonitor.lock();
```

```
while (replicatedBitSet.get(number)) {
     number++;
}
replicatedBitSet.set(number);
replicatedMonitor.unlock();
return number;
```

Este código es completamente análogo al empleado en una aplicación *multithread*; la diferencia existe sólo en los objetos empleados que, al estar replicados, pueden provocar errores en sus comunicaciones con las demás réplicas. Esta situación es paralela a la existente en cualquier aplicación CORBA: el acceso a objetos remotos se realiza bajo la abstracción de que son locales y la transparencia sólo se pierde en el menor rendimiento obtenido y en la necesidad de verificar los problemas de comunicaciones.

## 9.7. Transferencia de estado

Al promover la factorización de aplicaciones replicadas en componentes replicados, el empleo de una interfaz de transferencia de estado flexible resulta obviamente útil. Sin soporte de transferencia de estado, cada componente debería implementar sus propios protocolos y, si la interfaz de transferencia existente es poco flexible, provocaría una mala adaptación de esos componentes que podría redundar en un pobre rendimiento. En cualquier caso, la transferencia de estado ha sido el tema central en varios capítulos anteriores, con lo que nos limitamos aquí a exponer de nuevo la necesidad de un soporte flexible de transferencia de estado.

Un aspecto de los protocolos de transferencia que se expuso en los capítulos previos fue la necesidad de transferencias en varios pasos, especialmente útil en el caso de estados grandes. Al factorizarse una aplicación en componentes menores, ese requisito es aún necesario por varias razones:

- La aplicación puede enviar el estado de todos sus componentes en un solo paso o aprovechar el protocolo para enviar el estado en sucesivos pasos, cada componente por separado.
- Un componente separado puede ser aún suficientemente grande por sí solo. Tomando un contenedor como ejemplo, puede emplear sucesivos pasos para transferir su estado cuando contiene muchos elementos, o transferirlo en un único paso en caso contrario.

Como se vio anteriormente, un servidor no puede, en principio, procesar mensajes de otras réplicas o de clientes mientras efectúa una transferencia. En el caso de transferencias grandes, esa pérdida de servicio puede ser no permisible, con lo que sería preciso que la aplicación no se bloqueara y transmitiera su estado en diferentes etapas, implementando algoritmos que le permitieran transferir los cambios de estados producidos durante la transferencia en curso. Estos algoritmos

no son triviales, lo que supone otra ventaja para el argumento de emplear librerías de componentes replicados, que implementan ya estos algoritmos de forma optimizada.

## 9.8. Gestión de componentes replicados

El empleo de componentes replicados ha mostrado tres ventajas respecto a la replicación monolítica de la aplicación:

- Mayor facilidad para automatizar su implementación a partir de un componente no replicado.
- Minimizar el tiempo de desarrollo mediante reusabilidad de código, al emplear librerías con componentes replicados ya desarrollados, probados y optimizados.
- Simplificación del código, al acceder a esos componentes replicados como componentes estándar, no replicados.

Sin embargo, estos componentes no son objetos *normales*, tienen un ciclo de vida especial. Cada componente, una vez creado, debe acceder a un servicio GMS para crear un grupo o incluirse en uno ya existente. Y cada componente pertenece a un grupo distinto, con lo que la aplicación debe esperar a que todos los componentes pertenezcan a sus respectivos grupos para poder emplearlos.

Además, como se vio en los anteriores capítulos, un miembro puede ser excluido de un grupo accidentalmente, en caso de que resulte lento o sus comunicaciones con otras réplicas sean lentas. La exclusión de un componente dado de entre los muchos que una aplicación puede contener supondrá una merma total o parcial de la disponibilidad de esa aplicación. En este caso, es posible corregir esa expulsión incluyendo al componente de nuevo en un grupo, pero las causas que han provocado su exclusión accidental implican que muy probablemente otros componentes habrán sido expulsados igualmente de sus grupos. En cualquier caso, la aplicación debe manejar estos eventos y parar eventualmente su actividad.

Hay dos aproximaciones en el empleo de múltiples componentes. La primera es crear múltiples grupos, lo que implica la necesidad de coordinación desde la aplicación. La segunda es crear un único grupo donde cada componente aparece como *subgrupo*. Para minimizar la confusión con grupos y subgrupos, llamaremos *dominio* al grupo principal, que es el único que debe emplear el GMS para crear o incluirse en su grupo de réplicas, y componentes a esos subgrupos. Cuando un componente se comunica con sus réplicas, el dominio incluye una identidad asociada a ese subgrupo en los mensajes, de tal forma que el dominio destino pueda demultiplexar éstos al componente adecuado, como muestra la figura 9.5.

El modelo de dominios supone además un empleo menor de recursos. Empleando SenseiGMS, cada grupo crea su propio anillo, con un testigo independiente; cada miembro comprueba sus comunicaciones con el grupo, vigila la

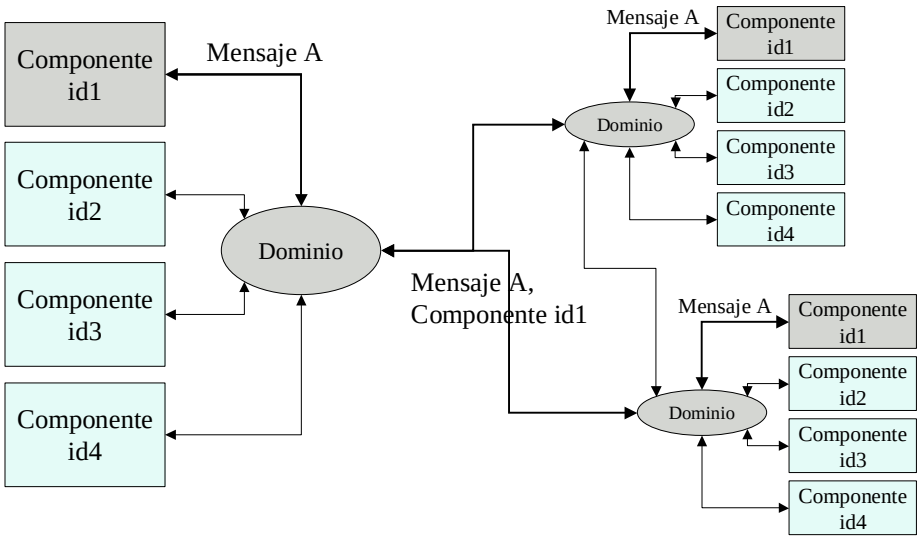


Figura 9.5. Empleo de *dominios*

pérdida o duplicación del testigo, etc. Si existe un único grupo, se limita ese empleo de recursos. Además se simplifica la interacción entre réplicas, al no tener que coordinarse los grupos de los distintos componentes; en su lugar, los componentes son los subgrupos que deben subscribirse al dominio mediante una sencilla operación. El dominio sólo debe asegurar que cada componente tenga la misma identidad en cada miembro del dominio. El estado del dominio lo forman el estado de esos componentes más las propiedades asociadas al dominio, como son las identidades de los componentes. Como el primer paso tras la inclusión del dominio en su grupo es la transferencia de estado, la coherencia en la asignación de identidades a componentes en los miembros del dominio puede comprobarse desde un principio. Este razonamiento implica que los componentes se registran en el dominio antes de que éste se una a su grupo.

Para entender mejor el concepto, emplearemos un ejemplo de un servicio de ficheros replicado. En este servicio [figura 9.6], los componentes son los directorios y ficheros, donde los directorios agrupan ficheros lógicamente. Todos los miembros del dominio contienen los mismos directorios y ficheros, de tal forma que el servicio es tolerante a fallos. Cada uno de los componentes en este ejemplo se registra en el dominio y la aplicación incluye a continuación el dominio en su grupo de réplicas. Cuando el dominio recibe los mensajes de transferencia de estado, obtiene el estado de los directorios y de los ficheros, así como información de los otros miembros en el dominio. En particular, recibe la confirmación de que todos los miembros han definido los mismos componentes con las mismas identidades.

Este ejemplo demuestra también la necesidad de dinamismo en el empleo de componentes. El servidor mostrado no puede extender el número de ficheros que

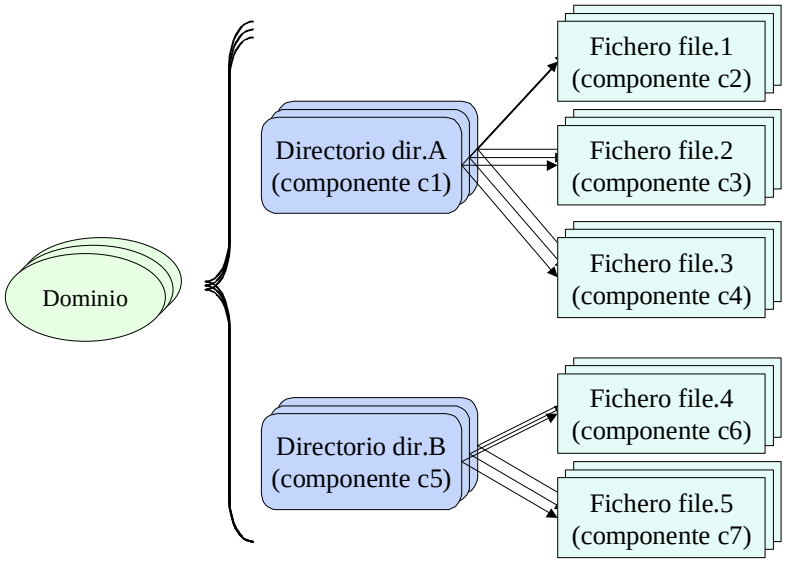


Figura 9.6. Servicio replicado de ficheros

sirve, puesto que los componentes deben ser conocidos antes de la inclusión en el grupo, momento en el que se podría obtener fácilmente esa información. Además, si se pretende poder emplear componentes replicados sin diferenciarlos de otros componentes *normales* en la aplicación, es preciso que se puedan crear esos componentes dinámicamente. Siguiendo con el escenario descrito, un cliente de la aplicación replicada podría solicitar la creación de un nuevo fichero en un directorio dado: el servidor crea en este caso el componente que representa el fichero y le asigna una identidad, comunicándoselo entonces a las demás réplicas.

Hay, por lo tanto, componentes estáticos, que se registran antes de incluirse en el grupo de réplicas, y componentes dinámicos, registrados tras la inclusión en el grupo. Todos los miembros del dominio deben registrar los mismos componentes estáticos, mientras que los componentes dinámicos son creados por un miembro del dominio, propagándose su creación a los otros miembros.

Un dominio no es más que un grupo que multiplexa/demultiplexa los mensajes de sus componentes, pero define a su vez sus propios mensajes. La creación de un componente dinámico puede implementarse entonces a partir de un mensaje específico del dominio, incluyendo información sobre el componente que se crea. Este componente debe existir en el espacio de memoria de cada una de las réplicas, por lo que cuando el dominio recibe el mensaje de creación del componente, solicita a la aplicación que cree dinámicamente ese componente con la información obtenida. De este requisito se deduce que la aplicación debe soportar una operación *callback* para crear componentes dinámicamente. Por ejemplo, en Sensei este método es:

```
GroupMember subgroupCreated(  in GroupMemberId creator, in SubgroupId id,
                              in DynamicSubgroupInfo info);
```

Se recibe la identidad dada al componente e información sobre el tipo de componente, tal como la haya dado la réplica que lo creó.

Durante el proceso de transferencia de estado, un dominio que se incluye en un grupo recibe información sobre los grupos dinámicos ya creados, de tal forma que pueda crearlos en su espacio de memoria antes de recibir el estado asociado a esos componentes. Es también posible eliminar componentes dinámicos, lo que se traduce en otro mensaje del dominio y en la necesidad de otro método *callback* en la aplicación. Sin embargo, no es posible eliminar los componentes estáticos, pues su existencia se verifica con cada transferencia de estado. Introducir estas diferencias entre ambos tipos de componentes puede parecer una complicación, teniendo en cuenta, además, que los dinámicos son más flexibles que los estáticos, pero las aplicaciones replicadas que precisen únicamente de componentes estáticos se benefician de no tener que soportar la funcionalidad adicional necesaria por los dinámicos.

Que las identidades de los componentes sean únicas implicará generalmente una necesidad de comunicaciones en el dominio para consensuar la unicidad de cada identidad. Por esta razón, es el mismo dominio el que genera las identidades en caso de componentes dinámicos. Esta es la otra diferencia con los componentes estáticos; cuando éstos se registran, el dominio aún no pertenece a un grupo, con lo que no puede comunicarse con otros miembros, lo cual deja tres opciones para la obtención de identidades:

- Las identidades son asignadas secuencialmente según se registran los componentes. Todos los dominios deben registrar entonces en ese mismo orden todos sus componentes estáticos, lo que puede originar problemas en caso de aplicaciones *multithread*.
- Las identidades son dadas por la aplicación al registrar el componente estático.
- La asignación de identidades se pospone hasta la inserción del dominio en un grupo: complica la aplicación y los componentes, que deben implementar la funcionalidad necesaria para recibir esa información. Además, si esa asignación se produce siguiendo el orden en que se registraron los componentes, se da el mismo problema que con la primera opción.

Por estos motivos, es la aplicación la que define las identidades de los componentes estáticos y el dominio el que las asigna para componentes dinámicos. Debe notarse que, sin el concepto dado de dominios, el empleo de componentes dinámicos es también posible cuando cada componente emplea su propio grupo de réplicas. En este caso la aplicación debe definir otro grupo especial para manejar los mensajes para la creación y destrucción de esos componentes.

Volviendo al ejemplo dado de servidores replicados de ficheros, su implementación puede realizarse ahora empleando únicamente componentes

dinámicos. Sin embargo, si la estructura de ficheros es jerárquica, el directorio raíz siempre estará presente, lo que da sentido a tener un único componente estático asociado a ese directorio. Cuando este servidor recibe una petición de creación de un fichero, crea el componente asociado:

```
public FileDescriptor createFile(String name, DirectoryDescriptor descriptor)
  throws DirectoryNotExisting, FileAlreadyExists
{
        Directory dir = getDirectory(descriptor);
        if (dir.contains(name)) {
                throw new FileAlreadyExists();
        }
        File file = new File(domain, name, descriptor);
        dir.add(file);
        return file.getDescriptor();
}
```

Aunque este código resulta muy similar al de un servidor no replicado, la primera diferencia se da ya en el constructor para el componente *File*, que necesita dos parámetros adicionales: el dominio, donde debe registrarse, y el descriptor del directorio, pues esta información debe propagarse a las demás réplicas:

```
File(Domain domain, String name, DirectoryDescriptor descriptor)
{
        DynamicSubgroupInfo info = new FileInfo(name, descriptor);
        subgroupId  = domain.createSubgroup(info, this);
}
```

Cuando el dominio crea el subgrupo, envía un mensaje a las demás réplicas del dominio, y sólo lo considera creado cuando ese mensaje es procesado, asignándole entonces una identidad única. Cada réplica debe suministrar su propio *callback* para crear el componente:

```
GroupMember subgroupCreated(int creator, int id, DynamicSubgroupInfo info)
{
        if (info instanceof FileInfo) {
                FileInfo fileInfo = (FileInfo) info;
                File file = new File(id, info.name);
                Directory dir = getDirectory(descriptor);
                dir.add(file);
                return file;
        }
        . . .
}
```

En este código, que no incluye condiciones de error, la aplicación decide el tipo de componente a crear en función de la información recibida. El constructor

empleado para el componente que representa el fichero es diferente al que se empleó anteriormente: ahora no precisa registrarse en el dominio, pues ya lo está, recibe su identidad asociada, que ya es conocida, y no precisa recibir el directorio en que se encuentra, pues esa información es redundante para el componente *fichero*.

El empleo de componentes dinámicos es entonces más complicado, con la necesidad de funcionalidad en la aplicación para crear componentes en demanda, y en los componentes para soportar diferentes constructores con cada escenario.

## 9.9. Transacciones

Los monitores replicados se han definido de forma similar a los monitores empleados en aplicaciones *multithread* pues solucionan un problema también similar: secuencializar el acceso concurrente a un componente o región crítica de código para evitar actualizaciones paralelas incompatibles.

Sin embargo, la concurrencia en componentes replicados implica un problema adicional que no se asocia a aplicaciones *multithread*, sino a bases de datos distribuidas: es preciso sincronizar el acceso a la base de datos para impedir que dos aplicaciones la actualicen de forma incompatible. Además, es necesario proteger a la base de datos en caso de que una aplicación se caiga sin haber completado sus modificaciones. La solución a este problema es el empleo de *transacciones* [Gray93].

En el caso de una aplicación que deba actualizar varios componentes replicados de forma coordinada, su caída, o expulsión accidental del grupo, antes de completar todos los cambios conlleva una inconsistencia en el estado de esos componentes. Es necesario, entonces, un mecanismo que proteja a los componentes replicados de caídas de las aplicaciones cuando éstas realizan actualizaciones sobre varios componentes simultáneamente. Este mecanismo lo llamamos, por paralelismo, transacción.

Empleando el ejemplo del servicio replicado de ficheros, la operación de mover un fichero a un directorio distinto podría escribirse de la siguiente manera, sin detallar cómo comenzar o finalizar la transacción:

```
void moveFile( String name, DirectoryDescriptor origin, DirectoryDescriptor target)
  throws DirectoryNotExisting, FileNotExisting FileAlreadyExists
{
        Directory dirOrigin = getDirectory(origin);
        Directory dirTarget = getDirectory(target);
        File file = dirOrigin.get(name);
        Start transaction
        dirOrigin.remove(file);
        dirTarget.add(file);
        End transaction
}
```

La transacción es necesaria para evitar que, si la aplicación cae tras borrar el fichero del primer directorio pero antes de incluirlo en el segundo, el resultado sea la pérdida de ese fichero.

El mecanismo de transacciones puede basarse en componentes que incluyan su soporte directamente: deben ser capaces de deshacer, no sólo el último cambio efectuado, sino todos los cambios realizados desde el momento en que se inició la transacción. Evidentemente, este soporte puede resultar muy complicado de implementar. Además, se complica el empleo de esos componentes, que deben recibir información sobre la transacción en curso; así, el bloque anterior de código relativo a la transacción podría escribirse como:

```
. . .
      Transaction transaction = startTransaction();
      dirOrigin.remove(file, transaction);
      dirTarget.add(file, transaction);
      endTransaction(transaction);
. . .
```

Es decir, las operaciones sobre el componente incluyen la información de la transacción. Lo que significa que la aplicación debe emplear transacciones incluso para actualizaciones que no las precisan, como es el caso en que afectan a un único componente. O bien, es necesario duplicar el número de operaciones en el componente, con y sin soporte de transacciones, complicando aún más ese componente.

Una posibilidad alternativa se basa en limitar los cambios en los componentes a la réplica que realiza la transacción, haciéndolos visibles a las demás réplicas cuando la transacción se completa. Como las operaciones sobre los componentes se traducen en mensajes al grupo, esta solución implica encolar esos mensajes durante la transacción. Esos mensajes deben, sin embargo, ser recibidos por la réplica que realiza la transacción, que sí debe ser capaz de observar los cambios efectuados. Este bloqueo de mensajes es fácil de realizar al emplear el concepto de dominios explicado en la sección anterior: puesto que todos los mensajes son multiplexados a través del dominio, es simple encolarlos a este nivel, mientras que si cada componente definiera su propio grupo, cada componente sería responsable de realizar ese bloqueo. El código relativo a la transacción se transforma ahora en:

```
. . .
      domain.startTransaction();
      dirOrigin.remove(file);
      dirTarget.add(file);
      domain.endTransaction();
. . .
```

Y el uso de la transacción resulta transparente para los componentes.

Hay dos problemas con esta solución por sus implicaciones con el modelo de sincronía virtual. El primero es que no todos los miembros del grupo observan los mismos mensajes en el mismo orden. Si otro miembro envía un mensaje durante la transacción, la réplica que realiza esa transacción observa ese mensaje entre los demás mensajes que la réplica está enviando, pues no hay un bloqueo de los mensajes que se reciben. Sin embargo, las demás réplicas ven primero el mensaje dado y luego todos los mensajes de la transacción, enviados cuando ha finalizado.

Por otro lado, el grupo podría instalar una nueva vista mientras un miembro efectúa una transacción, lo que presenta tres opciones:

- Posponer la instalación de la nueva vista hasta que la transacción se complete. Pero la transacción es un mecanismo lógico implementado sobre el GMS, que entonces debería ser extendido para conocer el estado de las transacciones. Esto implicaría un cambio muy importante, principalmente por la necesidad de redefinir el comportamiento del GMS (especialmente la detección de miembros caídos).
- Cancelar la transacción, que la aplicación debe reiniciar una vez que se instala la vista. Implica volver a escribir la lógica de tratamiento de vistas en la aplicación, rompiendo la transparencia que se logra con el empleo de componentes replicados. Además, los componentes en la réplica que realiza la transacción deberán deshacer los cambios efectuados hasta el momento, solución que hemos descartado por compleja.
- Instalar la vista y completar la transacción en la siguiente vista. Esta solución es la más viable, pero presenta una nueva incompatibilidad con el modelo de sincronía virtual: todos los miembros del grupo no observarán los mismos mensajes entre dos vistas consecutivas. El miembro que realiza la transacción ve los mensajes en la vista que se envían, mientras que sus réplicas los reciben en la vista en que concluye la transacción.

Por consiguiente, esta segunda implementación de las transacciones implica una menor complejidad de los componentes y de su empleo, pero viola el modelo de sincronía virtual. Sin embargo, como el correcto uso de las transacciones, tal como ocurre con en las bases de datos o con los monitores en las aplicaciones *multithread*, depende de la aplicación, ésta puede escoger cuándo esa violación es permisible.

Este argumento se basa en el cambio de enfoque realizado: la aplicación se basa ahora en componentes replicados, sin precisar un conocimiento de detalles de bajo nivel como es el cambio de vistas. La actualización del componente *directorio* que contiene las referencias a unos componentes *fichero* no se ve influida por el cambio en la composición del grupo de réplicas de ese componente *directorio* o componentes *fichero*, en tanto la actualización se realice apropiadamente en todos los componentes y las nuevas réplicas que se incluyan reciban un estado consistente.

Así, si un miembro inicia una transacción y recibe durante ésta un mensaje de otro miembro, la aplicación debe haber sido programada de tal manera que ese mensaje no afecte a la transacción. La transacción encierra, por lo tanto, una región crítica de código, pero ésta es una responsabilidad de los monitores y resulta lógico, consecuentemente, implementar las transacciones con monitores:

```
. . .
      domain.startTransaction(monitor);
      dirOrigin.remove(file);
      dirTarget.add(file);
      domain.endTransaction();
. . .
```

El inicio de la transacción implica la adquisición del monitor especificado, que es liberado cuando la transacción finaliza. Si el monitor no está libre, la transacción no puede comenzar.

Siguiendo con el mismo ejemplo, si el mensaje que la aplicación recibe durante esta transacción es el correspondiente a una operación de renombrado de un fichero tal como *`dirTarget.rename(file, newName)`*, implica que la aplicación no ha protegido correctamente la actualización del recurso *dirTarget* con un monitor, pues permite que dos réplicas modifiquen concurrentemente ese mismo recurso. En condiciones normales, las dos réplicas deberían haber escrito un código similar a:

***Réplica A***

```
. . .
      monitorTarget.lock();
      domain.startTransaction(monitorOrigin);
      dirOrigin.remove(file);
      dirTarget.add(file);
      domain.endTransaction();
      monitorTarget.unlock();
. . .
```

***Réplica B***

```
. . .
monitorTarget.lock();

dirTarget.rename(file,newName);

monitorTarget.unlock();
. . .
```

En estas condiciones, la réplica B no hubiera podido enviar el mensaje dado, pues estaría bloqueada esperando a adquirir el monitor asociado al directorio destino (o sería la réplica A la bloqueada). Los únicos mensajes que la réplica A puede recibir, siempre y cuando la aplicación se programe adecuadamente, son aquellos que no afecten al resultado de la transacción en curso: mensajes relativos a componentes no actualizados durante la transacción o mensajes que no actualicen componentes.

Es el mismo razonamiento empleado en el modelo de sincronía virtual. Este modelo permite la inclusión o expulsión dinámica de miembros, pero los miembros que pertenecen al grupo deben recibir los mismos mensajes en el mismo orden. Posteriormente, ese requisito de orden total puede reducirse y emplear órdenes menos restrictivos, siempre que una réplica sea incapaz de detectar sus

inconsistencias de estado con otras réplicas: si esta condición se cumple, los clientes de esas réplicas tampoco podrán observar las inconsistencias. Al emplear transacciones, el orden se altera y la consistencia entre miembros está sólo asegurada en los momentos donde no haya transacciones, siendo responsabilidad de la aplicación el ocultar esas inconsistencias temporales.

Sin embargo, la replicación de una aplicación específica puede aún requerir el empleo de orden total causal y comportamiento dinámico uniforme: si una réplica ha efectuado una determinada operación, las demás réplicas la deben efectuar igualmente, incluso si la primera ha caído. Si esa operación se realiza bajo una transacción, los demás miembros no tendrán ninguna información de las operaciones efectuadas si el miembro se cae antes de finalizarla, pues ésa es la finalidad buscada. Como consecuencia, este tipo de aplicaciones no admite el empleo de transacciones.

Las bases de datos permiten tres operaciones básicas con transacciones: iniciarlas, completarlas y abortarlas. Además, soportan transacciones anidadas [Moss85]. La segunda solución para las transacciones tiene un inconveniente con respecto a la primera planteada, basada en componentes capaces de deshacer sus cambios: una réplica no puede abortar una transacción.

La posibilidad de anidar transacciones es, por otra parte, muy conveniente; como cada transacción se realiza en torno a un monitor, implica, primero, la adquisición de nuevos monitores y, segundo, el bloqueo de los mensajes en tanto la última transacción no haya finalizado. Puesto que los monitores replicados se han definido como reentrantes, no es un problema emplear para una transacción anidada un monitor ya adquirido en una transacción externa.

Por otro lado, que una transacción bloquee los mensajes en la réplica que los envía, implica un problema con el empleo de monitores: los demás miembros deben saber que ese monitor ha sido adquirido, o podrían adquirirlo simultáneamente. Pero si esa adquisición del monitor precisa de comunicaciones con el grupo, no es posible su empleo durante las transacciones, lo que impide, a su vez, el empleo de transacciones anidadas. Ésta es la razón por la que el último listado de código no se escribió como:

***Réplica A***

```
 . . .
        domain.startTransaction(monitorOrigin);
        monitorTarget.lock();
        dirOrigin.remove(file);
        dirTarget.add(file);
        monitorTarget.unlock();
        domain.endTransaction();
 . . .
```

***Réplica B***

```
 . . .

monitorTarget.lock();

dirTarget.rename(file,newName);
monitorTarget.unlock();
```

El monitor *monitorTarget* es obtenido durante la transacción, pero la réplica B sólo recibe información de esa adquisición tras concluir la transferencia, cuando recibe a su vez información de su liberación. Al desconocer esa adquisición, ha podido realizar una actualización inconsistente sobre un recurso con acceso supuestamente exclusivo.

En la transferencia de estado se bloquean todos los mensajes entrantes en tanto la transferencia se realiza, pero la aplicación puede mejorar el rendimiento de la aplicación definiendo mensajes que pueden pasar ese bloqueo. De la misma forma, la aplicación puede definir mensajes que no son bloqueados durante la transacción, de forma que puedan emplearse los escenarios previamente descritos.

Las dos operaciones soportadas por un monitor replicado, *lock* y *unlock*, implican la definición de dos mensajes. El mensaje asociado a *lock* no debe ser bloqueado durante transacciones y es recibido por todos los miembros. Un monitor replicado está programado para reaccionar correctamente a la caída del miembro que lo posee, por lo que su exclusión de la transacción no implica posibles inconsistencias. El mensaje asociado a *unlock* sí debe ser bloqueado durante la transacción; en caso contrario, el miembro podría realizar, protegido por un monitor, una serie de cambios sobre uno o más componentes, y otra réplica, al adquirir el monitor liberado, realizar cambios incompatibles con los del primer miembro, que aún no ha finalizado su transacción y, por consiguiente, no ha hecho públicos sus cambios efectuados.

Esta solución al problema implica que los mensajes enviados en el dominio deben tener una propiedad que precise su comportamiento ante transacciones, de la misma forma que es necesaria otra propiedad para su comportamiento ante una transferencia de estado.

Hay, precisamente, una interacción importante entre transacciones y transferencia de estado: una réplica no debe coordinar una transferencia de estado en tanto no complete todas sus transacciones en curso. Este razonamiento se explica con la figura 9.7, donde una réplica inicia una transacción y, antes de concluirla, recibe una vista con un nuevo miembro al que debe coordinar. Antes de recibir la vista, la réplica envía un mensaje *m1* al grupo, que la transacción bloquea; sin embargo, la réplica procesa ese mensaje, actualizando el estado del componente *C*, que pasa a tener un estado {*m1*}. Si la transacción no impide la transferencia, envía inmediatamente su estado al nuevo miembro (en un mensaje que no se bloquea por la transacción), que recibe por tanto ese componente *C* con su estado actual, {*m1*}. A continuación, la réplica continúa su transacción, y envía un segundo mensaje *m2*, también bloqueado, que actualiza el estado de C a {*m1*,*m2*}. Al finalizar la transacción, el nuevo miembro recibe los mensajes que han sido encolados: *m1* y *m2*. Si procesa ambos mensajes, su estado pasa a ser {*m1*,*m1*,*m2*}, inconsistente con el del grupo salvo que *m1* sea idempotente. En caso contrario, debe obtener información adicional para ser capaz de descartar el mensaje *m1*. El escenario se

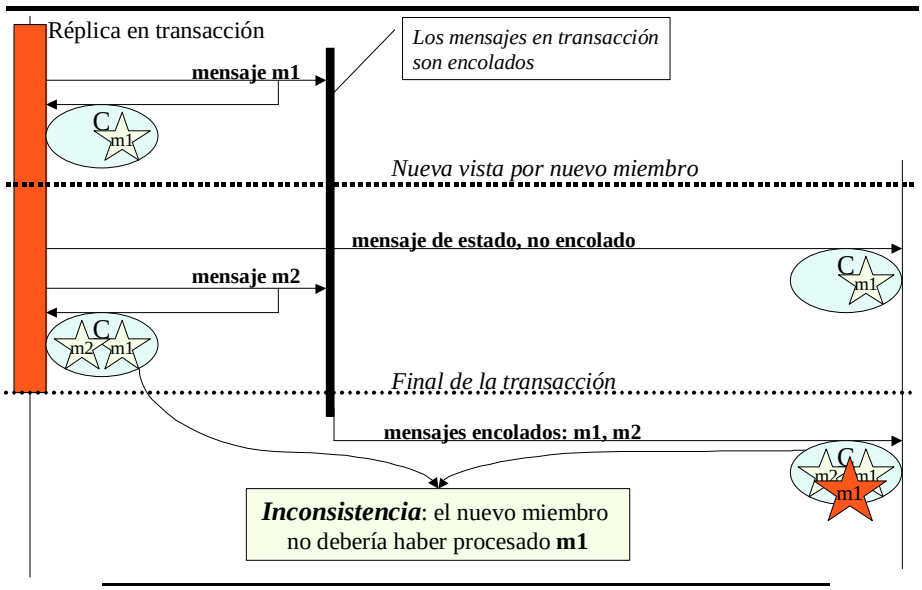


Figura 9.7. Interacción entre transacciones y transferencia de estado

complica aún más al considerar que la réplica en transacción puede también enviar mensajes que la transacción no bloquea.

Es posible complicar los mecanismos de transferencia de estado para solucionar este escenario, incluyendo cierta información adicional en los mensajes o, por ejemplo, bloqueando el mensaje de transferencia de estado durante la transacción. Pero impedir que un miembro que procesa una transacción sea el coordinador de una transferencia resulta además conveniente. Una transacción implica la adquisición de un monitor, y una réplica que adquiere un monitor bloquea a otras réplicas, por lo que resulta apropiado que un miembro en transacción (o simplemente poseyendo un monitor) no sea escogido como coordinador de una transferencia, o retrase ésta en tanto la finaliza.

Una consecuencia de definir un mensaje como no transaccionable es que tampoco puede ser bloqueado durante una transferencia de estado. Para comprobar esta afirmación, suponemos un caso donde un miembro del grupo está realizando una transferencia y un nuevo miembro se incluye en este mismo grupo. Aunque la transferencia no se inicia hasta finalizar la transacción, ambos miembros deben ya bloquear los mensajes del grupo. Un componente dado de este miembro ha definido un mensaje específico como no transaccionable y lo envía al grupo, sin que este envío sea bloqueado; sin embargo, el mensaje no está definido como no bloqueable durante las transferencias, con lo que ni alcanza al nuevo miembro, que lo encola, ni puede ser procesado por este miembro, que no puede entonces evolucionar en su transacción.

Consecuentemente, la definición de un mensaje como no transaccionable implica que el componente que lo emplea debe ser capaz de construir su estado aunque reciba mensajes en desorden (los bloqueados y no bloqueados en transferencia se reciben en distinto orden), lo que complica sus algoritmos de transferencia. De los componentes definidos en SenseiUMA, ninguno hace uso de este tipo de mensajes, limitándose, de momento, su empleo a los monitores.

Finalmente, existe un efecto lateral asociado al bloqueo de mensajes durante las transacciones: puesto que los mensajes no son enviados hasta la finalización de la transacción, se mejora el rendimiento de la aplicación. Cada operación sobre un componente se traduce, siguiendo las primeras secciones de este capítulo, en un mensaje al grupo y el bloqueo de la réplica en tanto el mensaje se recibe y se procesa. Bajo una transacción, esos bloqueos son resueltos inmediatamente, resultando en un código más rápido. Además, puesto que todos los mensajes son enviados al finalizar la transacción, se obtiene un beneficio adicional, pues los sistemas de comunicaciones fiables en grupo, incluido SenseiGMS, tienen un rendimiento mejor cuando una réplica envía sucesivos mensajes: la fiabilidad de las comunicaciones precisa que los mensajes sean enviados y confirmados (y descartados, según el sistema), pero el envío de un segundo mensaje implica la confirmación del anterior, ahorrándose el sistema un paso del protocolo. Dependiendo del tamaño de cada mensaje, es también posible agrupar los correspondientes a la transacción en un único mensaje, que es luego dividido en el dominio distinto.

### 9.9.1. Transacciones y el modelo de sincronía virtual

El modelo de componentes desarrollado en Sensei sobre el modelo de sincronía virtual ha impuesto la necesidad de transacciones para mantener la coherencia de los datos en caso de caídas. Sin embargo, las relaciones entre el modelo de transacciones y el modelo de sincronía virtual han sido ya ampliamente estudiadas con anterioridad, pues ambos modelos sirven, independientemente, para el mismo fin: la construcción de aplicaciones fiables.

Tanto los algoritmos de transacciones, como los empleados para las comunicaciones multipunto en el modelo sincronía virtual, deben resolver problemas de *consenso*. Se ha demostrado que existe un mecanismo básico, *DT-multicast*, con el que se pueden construir ambos modelos [Guerraoui95], partiendo de una implementación del protocolo de compromiso (*commit protocol*).

Sin embargo, la integración de ambos modelos se ha realizado normalmente mediante la implementación de uno sobre el otro. Por ejemplo, puesto que un mensaje fiable multipunto tiene por sí mismo semánticas transaccionales [Schiper96], es posible obtener la atomicidad requerida para una transacción encapsulándola en un único mensaje fiable multipunto. La propuesta de implementación de transacciones en SenseiDomains emplea este mismo enfoque.

Es más habitual emplear el modelo de sincronía virtual para incrementar la disponibilidad de sistemas basados en transacciones. En este caso, el sistema está basado en transacciones y el modelo de sincronía virtual queda generalmente oculto. Una opción [Little00] es que las transacciones se efectúen sobre sistemas replicados, pero el principal inconveniente es que cada una de las réplicas debe procesar cada transacción, incluso aquellas que no se completan y deben abortarse. Se obtiene un mejor rendimiento cuando los cambios se realizan sobre una única réplica que, a continuación, los propaga a las otras réplicas [Patiño00]; la transacción todavía puede abortarse, en caso de colisión con otras transacciones.

SenseiDomains también procesa cada transacción en una única réplica, propagando los cambios al final, pero el enfoque es radicalmente diferente. SenseiDomains no es un sistema basado en transacciones, sino un sistema basado en el modelo de sincronía virtual que soporta un mecanismo similar a las transacciones para preservar la consistencia de varios componentes en caso de caída de una réplica.

Una transacción debe verificar cuatro propiedades, denominadas *ACID* por su denominación en idioma inglés: Atomicidad (*Atomicity*), consistencia (*Consistency*), aislamiento (*Isolation*) y persistencia (*Durability*). Una transacción, o se completa y los cambios efectuados no se pierden incluso ante fallos o caídas, o se aborta, en cuyo caso el estado del sistema es el que tuviera antes de que se efectuara la transacción. La propiedad de aislamiento implica que las transacciones concurrentes no observan los cambios que se producen en otras transferencias hasta que se completan, y esta propiedad no se verifica en SenseiDomains para las transacciones anidadas ni las concurrentes iniciadas por una misma réplica.

En SenseiDomains, una transacción protege exclusivamente sobre caídas en un miembro y no es el mecanismo básico para obtener fiabilidad. Por esta razón, aislar los cambios producidos por transacciones iniciadas por una misma réplica no conlleva ningún beneficio al modelo, donde el empleo de las transacciones es totalmente transparente para la réplica que las efectúa (si una transacción no observa los cambios que se producen en otra, esta transparencia se pierde).

Además, la concurrencia de las transacciones no es tampoco una prioridad; un sistema basado en transacciones busca optimizar esta concurrencia, de tal forma que puedan procesarse tantas transacciones concurrentes como sea posible. Los mecanismos de control de concurrencia se clasifican como pesimistas u optimistas, y el empleo de monitores, aun siendo el más habitualmente empleado, entra en el grupo de mecanismos pesimistas. Sin embargo, hay varios tipos de monitores y los empleados en SenseiDomains son de lectura/escritura, considerados los menos concurrentes. Otros sistemas basados en transacciones, como *TransLib* [Jiménez99], emplean monitores basados en conmutación, donde es posible definir las condiciones que permiten conmutar dos operaciones y procesarlas concurrentemente. Por ejemplo, con un monitor de lectura/escritura, una operación de creación de un fichero en un directorio no es concurrente con ninguna otra

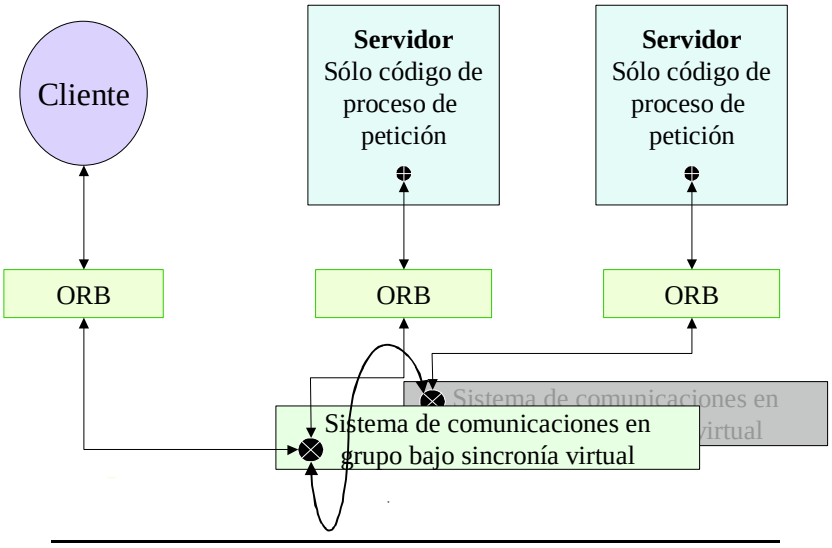


Figura 9.8. Modelo del servicio de tolerancia a fallos CORBA

operación de modificación de ese directorio; con un monitor basado en conmutación de operaciones, es posible crear un fichero y borrar simultáneamente otro, siempre que no tengan el mismo nombre.

## 9.10. Comparación con el modelo de CORBA

Como se expuso en capítulos anteriores, el servicio de tolerancia a fallos de CORBA soporta replicación activa y pasiva y, dentro de la pasiva, diferentes opciones de *backup*. En todos los casos la aplicación se programa bajo un mismo modelo, lo que permite cambiar el tipo de replicación sin modificar la aplicación, al menos teóricamente.

Esta filosofía implica que las réplicas se programan sin ningún conocimiento de las otras réplicas (aunque pueden obtener esa información) y es el servicio el encargado de emplear el modelo de sincronía virtual e implementarlo transparentemente para las réplicas.

La figura 9.8 muestra el modelo empleado en este servicio CORBA. Su especificación incluye la posibilidad de que el modelo de sincronía virtual que sustenta las operaciones en grupo se implemente por debajo del ORB, para así obtener un mejor rendimiento. En este caso, la petición del cliente es recibida directamente por este sistema de comunicaciones, que la envía a los demás miembros del grupo. Este sistema ordena todas las comunicaciones al grupo,

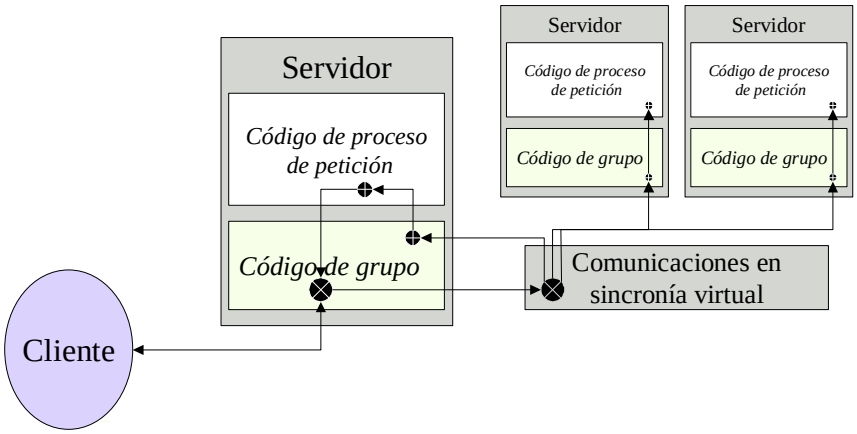


Figura 9.9. Esquema simple de la sincronización de la respuesta

enviándolas secuencialmente a las réplicas. Las réplicas no precisan incluir ningún código de grupo, sólo el código de procesado de las peticiones de los clientes.

Por otra parte, la figura 9.9 muestra de forma esquemática el modelo de sincronización expuesto en este capítulo: el cliente accede a un servidor, que incluye internamente el código para comunicar a sus réplicas la operación en curso, que se procesa en orden total con respecto a las demás operaciones que el grupo está realizando. Es posible diferenciar el código que procesa realmente la petición, similar o igual al que precisaría el servidor si no estuviera replicado, del código empleado para mantener la consistencia en el grupo.

Ambos modelos son muy similares, pero el servicio CORBA oculta todas las comunicaciones al grupo, lo que permite un diseño de las réplicas muy simplificado. En principio, la única diferencia entre un servidor no replicado y otro que lo está, es el soporte de las primitivas de transferencia de estado. Sin embargo, esta simplificación supone importantes inconvenientes.

El primer problema que presenta es la imposibilidad de optimizar las comunicaciones. Un importante número de operaciones en todo servidor lo constituyen las operaciones de consulta, que no actualizan su estado. Estas operaciones no precisan comunicarse al grupo y pueden procesarse inmediatamente. Bajo CORBA, esta optimización no es inicialmente posible; una posibilidad sería que esas operaciones se definieran en la interfaz IDL a partir de atributos de sólo lectura, que se traducen en operaciones de lectura de ese atributo. Sin embargo, la especificación CORBA no incluye la limitación de que estas operaciones no puedan

actualizar el estado del servidor, por lo que sería preciso una extensión propietaria en la interfaz IDL para realizar esta optimización.

Si el servidor es capaz de procesar una petición de consulta directamente, es posible suponer que el siguiente código podría producir un error:

```
server.assign("Nombre", "Luis");
if (server.read("Nombre") =! "Luis") {
        ERROR("Incongruencia");
}
```

Si la operación *read* se procesa inmediatamente, puede, de hecho, procesarse antes que la operación *assign*, con lo que el código anterior produciría un error. Sin embargo, el cliente que ejecuta el anterior código, lo hace secuencialmente, es decir, no ejecuta la operación *read* hasta que *assign* ha finalizado, lo que evita la anterior incongruencia.

El segundo problema está asociado al modelo de transferencia de estado, basado en mecanismos de archivo: obtener periódicamente el estado de la réplica que se almacena junto con todos los mensajes recibidos desde que se obtuvo el estado. El estado se puede obtener cada vez de forma completa o devolviendo simplemente los cambios efectuados.

La especificación del servicio no detalla estos mecanismos, en particular no precisa si todas las réplicas deben devolver periódicamente su estado o sólo se accede a una de las réplicas. Tampoco precisa cómo almacenar el estado. Una solución sería emplear un servicio de archivo tolerante a fallos, con lo que bastaría con acceder a una de las réplicas. Sin embargo, esta solución es complicada por la dificultad asociada a ese servicio de archivo y la solución básica es acceder a cada réplica y almacenar su estado directamente en la máquina donde se ejecuta.

Este tipo de transferencia es perfectamente lógico en caso de replicación pasiva, donde sólo una réplica conoce el estado de la aplicación. Pero la obtención del estado implica el bloqueo de la réplica que lo transfiere, con lo que bloquear todos los miembros del grupo de forma periódica sólo puede redundar en un mal rendimiento.

El tercer y más importante problema está ligado con la imposibilidad del servicio de soportar aplicaciones no deterministas. Pero, tal como ocurría con los mecanismos de automatización del proceso de replicación desarrollados al principio de este capítulo, el problema real se traduce en la imposibilidad de acceder a componentes externos. Volviendo al ejemplo del generador de números únicos aleatorios, todas las réplicas deben ser capaces de generar el mismo número aleatorio para mantener la consistencia del grupo. La solución planteada fue el empleo de un componente externo que generara ese número aleatorio, pero esta solución es, si no imposible, al menos muy difícil de implementar bajo CORBA.

Si todas las réplicas acceden al componente externo, ese componente recibe múltiples accesos: en el caso del generador de números, no generará uno solo, como se desea, sino tantos como réplicas haya. Consecuentemente, el servicio de tolerancia a fallos debe interceptar no sólo las llamadas a la réplica, sino las llamadas efectuadas por ésta, y coordinarlas con las llamadas de las otras réplicas del grupo para que se realice efectivamente un único acceso al componente dado.

Es posible realizar esta interceptación y coordinación de llamadas, aun cuando no está incluida en la especificación CORBA e implica un peor rendimiento del sistema. Además el problema persiste porque no todos los accesos externos se realizan a componentes CORBA y el servicio será incapaz de interceptar y coordinar llamadas a un servidor *ftp* o *http*, por ejemplo. E incluso con respecto a componentes CORBA, no todas las llamadas deben ser tampoco interceptadas.

Por ejemplo, una aplicación puede emplear una segunda aplicación que ofrece un servicio de traceado y que se ejecuta en la misma máquina que la primera. Asumiendo que esta segunda aplicación es CORBA, no tendría sentido que el servicio de tolerancia a fallos interceptara las llamadas de todas las réplicas y enviara sólo información de trazas de una de las réplicas: al realizar este traceado se pretende saber qué ocurre con cada réplica. Pero si ése fuera precisamente el objetivo, la aplicación no podría emplear ese servicio de traceado en el caso que su interfaz no fuera CORBA. Por ejemplo, con una interfaz basada en un protocolo propietario empleando directamente *sockets*.

Por lo tanto, el modelo CORBA soporta las mismas limitaciones que la automatización del proceso de replicación que se propuso en este capítulo, donde se producía un servidor replicado a partir de su interfaz y una instancia del servidor no replicado. Esa automatización dejaba sin cubrir un aspecto: las operaciones de transferencia de estado, que son precisamente las mismas operaciones que deben añadirse sobre un servidor no replicado al emplear el servicio de tolerancia a fallos de CORBA.

En el caso de la automatización, las limitaciones se resolvieron factorizando la aplicación en componentes, pero este paso no es posible bajo CORBA, al menos bajo la especificación actual. Como consecuencia, y salvo que se use replicación pasiva, el dominio de aplicación del servicio de tolerancia a fallos de CORBA se reduce a servidores sencillos.

## 9.11. Conclusiones

La metodología de construcción de aplicaciones tolerantes a fallos expuesta a lo largo de este capítulo se basa en el empleo de componentes replicados, intentando reducir al máximo la lógica asociada a las comunicaciones en grupo entre réplicas en una aplicación tolerante a fallos, que se programa mediante el empleo de esos componentes. Esta propuesta se fundamenta en dos ideas: el empleo de librerías

existentes de componentes ya probados y optimizados, y la posibilidad de automatizar el proceso de replicación de componentes sencillos.

Un servicio tolerante a fallos contiene, en este caso, servidores sin un conocimiento mutuo, donde toda interacción se realiza a través de los componentes replicados, que ofrecen la abstracción de componentes *compartidos*. Esta compartición de recursos implica que los servidores los acceden simultáneamente, por lo que precisan de soluciones que impidan resultados inconsistentes por actualizaciones concurrentes. Estas soluciones son los monitores replicados y las transacciones.

Para facilitar la utilización de esos componentes, su empleo se simplifica mediante el concepto de *dominios*, que permiten gestionarlos en su conjunto, y se soporta su dinamismo: los componentes pueden crearse y eliminarse dinámicamente, aumentando su semejanza a componentes *normales* y, por consiguiente, su facilidad de uso.

En comparación, el servicio de tolerancia a fallos de CORBA define un modelo muy simplificado, sin estos problemas de concurrencia. La diferencia estriba en la granularidad en la replicación: CORBA replica un servidor de forma monolítica mientras que la metodología expuesta en este capítulo permite definir esa replicación a nivel de componentes. Como consecuencia, el servicio de tolerancia a fallos CORBA tiene muy poca aplicabilidad en replicación activa, limitándose a servidores sencillos.

El empleo de componentes replicados implica la posibilidad de diseñar aplicaciones tolerantes a fallos a alto nivel, sin necesidad de emplear las primitivas directamente soportadas en el modelo de sincronía virtual, basadas en vistas y mensajes. Esta metodología se basa totalmente en este modelo, que queda sin embargo oculto para la aplicación.

Este capítulo ha definido las facilidades necesarias para soportar esta replicación basada en componentes y ha especificado su implementación en términos del soporte dado por un sistema de comunicaciones en grupo basado en sincronía virtual. De hecho, hace uso exclusivo de las primitivas de comunicaciones empleando orden total causal, sin precisar el soporte de otros órdenes menos restrictivos, lo que, sin embargo, implica la necesidad de que el grupo de comunicaciones sobre el que se implemente tenga un buen rendimiento con las comunicaciones con orden total.