Replicación de objetos distribuidos

Juan Pavón Mestras, Luis Miguel Peña Cabañas
Departamento de Sistemas Informáticos y Programación
Universidad Complutense Madrid
http://bogart.sip.ucm.es/~juan

INTRODUCCION

Un aspecto escasamente tenido en cuenta a la hora de desarrollar aplicaciones distribuidas es el de la fiabilidad [Birman96], lo cual ocasiona degradaciones de la calidad de servicio o su interrupción temporal. Ello es debido en gran parte a que los diseñadores dejan este aspecto para la fase de implementación y a la falta de soporte para la implementación de mecanismos estándares que permitan reducir la complejidad añadida para su tratamiento.
Un método común para mejorar la disponibilidad de los sistemas es la replicación del software en máquinas distintas de la red. Si se está utilizando un modelo de objetos distribuidos como CORBA o Java RMI, entonces parece razonable considerar el objeto como la entidad software a replicar. Para facilitar su implementación en aplicaciones distribuidas es conveniente soportar la transparencia de replicación para el programador. Esto implica:

- Que el cliente vea una sola referencia para un objeto replicado, y no varias (una por cada replica del objeto).
- Que el sistema dé las facilidades para implementar la consistencia de estado de las réplicas. Esto no siempre es posible hacerlo de forma automática porque en general el estado de los objetos depende en gran medida de la aplicación. Pero es posible declarar métodos que el programador pueda definir y que sirvan de punto de entrada para los mecanismos que el sistema utiliza para mantener el estado consistente de cada grupo de réplicas.
  Concretamente, la infraestructura del sistema debe soportar un modelo de sincronía virtual, para lo cual implementa mecanismos de comunicación por difusión o multipunto, un servicio de gestión de grupos, y esquemas de transferencia de estado entre las réplicas.

La especificación reciente del servicio de tolerancia a fallos para CORBA [orbos 2000-01-19] propone una solución concreta a este tema, muy enfocada hacia la replicación pasiva, dejando varios aspectos de la interfaz abiertos a implementaciones particulares.
Nuestro trabajo se ha enfocado inicialmente en la definicion de mecanismos genéricos para la transferencia de estado en sistemas de objetos distribuidos, sea CORBA o RMI.Estos mecanismos se implementaron inicialmente sobre Ensemble, un sistema disponible de comunicaciones fiables multipunto. A partir de aqui desarrollamos nuestro propio sistema de comunicaciones fiable en java, presentando un interfaces RMI y CORBA. Sobre este sistema de comunicaciones, y con el soporte de los mecanismos de transferencia de estado, nuestro enfasis ha sido la automatizacion del proceso de fiabilizacion de una aplicacion.
Este articulo estudia una metodologia generica que puede aplicarse a la creacion de aplicaciones fiables e introduce UMA, una herramienta que hemos desarrollado para la automatizacion de estos procesos. El objetivo de esta herramienta es la generacion de todo el codigo necesario para, a partir de un componente no fiable, obtener un segundo componente con toda la logica de replicacion imbuida. Una aplicacion directa es la fiabilizacion por replicacion de componentes CORBA o javaBeans.

SenseiGMS

Aplicando el modelo de sincronia virtual y un algoritmo de paso de Token, y soportando exclusivamente orden total de mensajes, hemos desarrollado un sistema de comunicaciones fiables multipunto denominado SenseiGMS. Usa primitivas de comunicaciones de alto nivel, mas especificamente RMI o CORBA lo que puede implicar un peor rendimiento que otras soluciones directamente construidas con sockets sobre TCP o UDP. Por otro lado, implica que todas las comunicaciones entre los miembros del grupo se hacen de forma estandar sobre el substrato de comunicaciones elegido, lo que permite aprovechar las caracteristicas del mismo, como por ejemplo la seguridad.
El protocolo de paso de Token implica tambien una importante degradacion del rendimiento cuando el numero de replicas aumenta; por ello, y aunque nuestro trabajo se centra sobre aplicaciones con un reducido numero de replicas, Sensei (el nombre que abarca todo nuestro proyecto) soporta igualmente Ensemble y su protocolo propietario de comunicaciones. En este caso, encapsulamos su funcionalidad bajo el mismo interfaz soportado por SenseiGMS, de tal forma que ambos sistemas sean intercambiables, aunque en este caso se pierde el soporte para RMI.
En todos los casos, las comunicaciones fiables se basan en dos clases, GroupMember y GroupHandler. GroupMember es el interfaz a implementar por las clases usando comunicaciones en grupo, y GroupHandler es

el mecanismo empleado para realizar esas comunicaciones. Las definiciones de las dos clases son consistentes, sin variación según el middleware empleado. La definición bajo CORBA es:

```
typedef long GroupMemberId;
valuetype Message {};

interface GroupHandler
{
	GroupMemberId getGroupMemberId();
	boolean isValidGroup();
	boolean leaveGroup();
	boolean castMessage(in Message msg) raises (InvalidGroupException);
	boolean sendMessage(in GroupMemberId target, in Message msg)
		raises (InvalidGroupException);
};

interface GroupMember
{
	void processPTPMessage(in GroupMemberId sender, in Message msg);
	void processCastMessage(in GroupMemberId sender, in Message msg);
	void memberAccepted(in GroupMemberId identity,
		in GroupHandler handler, in View theView);
	void installView(in View theView);
	void excludedFromGroup();
};
```

Una aplicación puede definir sus propios mensajes que enviara a traves del *GroupHandler*; esos mensajes, asi como los demas enviados por otros miembros del grupo seran recibidos fiablemente y con orden total a traves del interface *GroupMember*. Usando RMI, los mensajes se definen como clases serializables pertenecientes a la jerarquia *Message*. Bajo CORBA, hemos optado por una característica recien incluida en la especificación: objetos por valor. Las otras dos estructuras posibles, *struct* o *interface* no son apropiadas. La ultima porque el acceso al contenido mensaje supondría de nuevo un acceso remoto, y la primera porque CORBA no soporta herencia de estructuras.

Desarrollo de aplicaciones fiables

En una aplicacion cliente/servidor, el servidor define un interfaz que el cliente emplea para solicitar sus servicios. Generalmente esto implica que el cliente queda bloqueado mientras el servidor procesa su solicitud, aunque esto no es siempre cierto, como es el caso de las operaciones *oneway* en CORBA. En una aplicacion fiable, este servidor debe contactar con las demas replicas y obtener asi una respuesta consensuada. Por ejemplo, supongamos un servidor muy simple que genera numeros unicos de forma secuencial, con el siguiente interfaz:

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

Si no hay replicacion, el servidor simplemente devuelve un numero, incrementando una variable interna que mantenga el ultimo numero generado. Si hay varias replicas, el servidor debe contactar primero a las otras replicas de tal forma que dos servidores no devuelvan el mismo numero. Empleando orden total y puesto que todo mensaje enviado al grupo es recibido asi mismo por la replica que lo envia, basta con que el servidor genere el numero cuando recibe el mensaje apropiado.
Desde un punto de vista logico, hay un desfase entre la solicitud del numero y su generacion:

1. El servidor recibe solicitud de un cliente.
2. El servidor envia esta solicitud al resto de replicas mediante un mensaje fiable.
3. Cuando un servidor recibe un mensaje de solicitud de numero, incrementa su variable interna.
4. Si el servidor que recibe el mensaje es el que recibio la solicitud del cliente, debe asimismo devolver el numero generado al cliente.

Desde el punto de vista de implementacion, la solicitud del cliente en un servidor queda bloqueada hasta que se recibe el mensaje asociado a ese cliente en ese servidor, lo que implica el empleo de semaforos o monitores. Ademas, puesto que empleamos un modelo de sincronia virtual apoyado en detectores de fallos imperfectos, un miembro puede ser expulsado del grupo sin recibir el mensaje que habia enviado. Por lo tanto, para evitar que el cliente no espere indefinidamente por su solicitud, ese servidor debe actuar correctamente si recibe la notificacion de expulsion del grupo.
Si el interfaz presenta varias operaciones, la anterior logistica debe repetirse para cada operacion. Ademas, deben definirse mensajes distintos segun la operacion involucrada. Por ejemplo, si la operacion tiene la forma:

```
	long modifyBalance(in Account account, in long newAmount);
```

el mensaje que se envia a las replicas debe contener los datos de *Account* y *newAmount*, de tal forma que todas las replicas procesen la misma informacion:

```
valuetype ModifyBalanceMessage : Message
{
	public long    requestId;       //to identify the request
	public Account account;
	public long    newAmount;
};
```

En este caso particular, todos los parametros son 'in'. Si son 'out', no hace falta incluirlos, puesto que solo el servidor que procesa la solicitud del cliente debe devolverle un valor determinado, al igual que pasa con el valor de retorno de la funcion.
El ejemplo anterior ha mostrado un posible algoritmo para la implementacion de miembros de grupo. Es una programación repetitiva que puede automatizarse fácilmente: definir mensajes, asociar monitores a las solicitudes de clientes, enviar los mensajes de forma fiable, etc. Sensei incluye una herramienta denominada UMA que pretende facilitar estas tareas. Sin embargo, el alcance de UMA trasciende esta generación basica de codigo.

SenseiUMA

UMA permite crear todo el codigo y clases necesarias para definir un objeto replicado a partir de su definición simple. Esta generación puede realizarse para RMI, a partir de su interfaz Java, o para CORBA, a partir de su interfaz CORBA, y en este ultimo caso puede emplearse el núcleo de comunicaciones de Sensei o bien Ensemble: en ambos casos es solo necesario el interfaz y una clase que implementa ese interfaz sin soporte de replicación.
UMA puede ejecutarse sobre un interfaz o clase Java o bien sobre un interfaz IDL CORBA, y generara automaticamente el codigo necesario para soportar fiablidad en todas las operaciones definidas. Puesto que algunas operaciones no precisan esta fiabilidad, se puede bloquear la generacion de mensajes y codigo para esas operaciones. Por ejemplo, en una clase *Vector*, la inclusion o eliminacion de objetos debe hacerse de forma consensuada, sin embargo obtener el numero de objetos es una operacion que puede realizar una replica sin comunicarselo a las demas.
Supongamos una clase *NumberGeneratorImpl* que implementa el interfaz *NumberGenerator* definido en el punto anterior, sin ningun soporte de replicacion. Siguiendo el esquema mostrado en el anterior apartado, hace falta un unico mensaje que no contiene ninguna variable interna ademas de la identidad de la solicitud:

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

Genera asimismo una clase *GroupNumberGeneratorImpl* que implementa igualmente el interfaz *NumberGenerator* e incluye toda la logica de grupo necesaria, que no detallamos aqui por motivos de extension. La logica de aplicacion se delega a la instancia de la clase *NumberGeneratorImpl*, con lo que la operacion *getNumber* resulta ahora (eliminado parte de la complejidad asociada) es:

```
long getNumber() throws InvalidGroupException
{
	group.castMessage(new GetNumberMessage());
	long ret = theNumberGeneratorImpl.getNumber();
	group.subgroupCastMessageProcessed();
}
```

Esta implementacion usa un *GroupHandler* especifico que bloquea al objeto que envia el mensaje hasta que este es recibido por ese objeto. Este *GroupHandler* intercepta la recepcion de mensajes, de tal forma que desbloquea al servidor que envio el mensaje, y lo deja pasar cuando el mensaje ha sido enviado por otras replicas. Estas otras replicas deben emular la operacion como si recibieran directamente la peticion del cliente, pero sin necesidad de devolver ningun valor:

```
public void processCastMessage(int sender, Message message)
{
	if (message instanceof GetNumberMessage)
	{
		GetNumberMessage gnMessage = (GetNumberMessage) message;
		theNumberGeneratorImpl.getNumber();
```

```
        }
}
```

Es decir, es posible crear una aplicación totalmente replicada a partir de esa aplicación existente sin soporte de replicación. Por ejemplo, es posible ejecutar UMA sobre la clase *java.util.Vector* de java para obtener una clase *GroupVector* donde las operaciones son inmediatamente replicadas a las demas replicas en el mismo grupo.
No obstante, esta generacion automatica de codigo fiable mediante replicacion es solo posible en el caso de aplicaciones muy simples.

Identificación de estructuras replicables

El anterior enfoque implica replicar el comportamiento de la aplicación a patir de su interfaz. Pero un interfaz define las operaciones a soportar, no la implementacion o estructuras de datos necesarios. Si en el ejemplo anterior el generador de numeros unicos debiera devolver numeros aleatorios, su interfaz seria aun el mismo:

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

Internamente, la aplicación puede definir ahora una estructura de datos incluyendo todos los numeros ya generados. Cuando se solicita un numero, se obtiene un numero al azar y se comprueba en la anterior estructura de datos si esta libre o no; si no lo esta, se puede generar otro o simplemente iterar en la tabla a partir del numero anterior buscando un numero libre. Esta tabla es finita, pero obviamos este hecho para los propósitos de este ejemplo.
La replicación de esta aplicación con el enfoque indicado en el punto anterior implica que cada replica mantiene su propia estructura de datos. Cuando un cliente solicita un numero, el servidor se lo comunica a sus replicas, que generan cada una un numero unico, y este servidor generara igualmente un numero unico que devolvera al cliente. Obviamente, cada replica habra generado un numero diferente, salvo que el generador de numeros aleatorios sea francamente nefasto, y por lo tanto dos servidores distintos podran suministrar eventualmente el mismo numero, violando el requisito del servicio de que todo numero suministrado es unico.
Este ejemplo se puede solucionar de varias maneras. Por ejemplo, el servidor puede generar primero un numero aleatorio, que envia en el mensaje a las demas replicas. Si todas parten del mismo numero, todas deberían alcanzar el mismo resultado.
Otra solucion, mucho mas genérica, es identificar las estructuras de datos que necesitan replicación y replicarlas. En este ejemplo, es la estructura que mantiene la lista de numeros generados la estructura a replicar. El servidor no se encuentra en este caso replicado, cuando varios clientes acceden a varios servidores simultáneamente, estos no deben ponerse de acuerdo, pero cuando acceden a su lista particular de numeros generados, este acceso es un acceso a una estructura replicada.
En el ejemplo del generador de numeros, este nuevo enfoque implica definir un nuevo interfaz para esa estructura de datos, por ejemplo:

```
interface NumberSet
{
        boolean testNumber(in long Number);
        void setNumber(in long Number);
};
```

La aplicación usara entonces de modo natural una clase implementando el anterior interface, por ejemplo *NumberSetImpl*. Con soporte de replicación, UMA generara una nueva clase *GroupNumberSetImpl*, implementando el interfaz *NumberSet* y usando la clase *NumberSetImpl* para la logica de aplicación. El unico cambio que es entonces necesario en la aplicación es cambiar la instancia de *NumberSet* que se crea.
Notese que en este ejemplo especifico, la operación *setNumber* debera comunicarse a las demas replicas, pero la operación *testNumber* no precisa de ninguna comunicación de grupo.

Acceso concurrente a estructuras replicadas

Una aplicación CORBA puede usar el modelo multithreaded para soportar el acceso concurrente de varios clientes. En este caso debe emplear monitores o semáforos que le aseguren un acceso seguro a sus estructuras de datos, o bien emplear librerias que soporten ya un acceso concurrente.
La replicación de una estructura de datos tal como hemos mostrado en el punto anterior implica que desde un punto de vista lógico hay un acceso concurrente a la estructura de datos. En el ejemplo anterior, dos replicas podrían simultáneamente acceder al mismo numero: la primera replica testearia (*testNumber*) si ese numero esta libre, y si lo esta, lo definiria como generado (*setNumber*) y lo devolveria a la aplicación. Sin embargo, la

segunda replica podria testear el mismo numero antes de que la primera lo definiera como generado, y generar por lo tanto el mismo numero. Lo que en programacion concurrente se denomina *race condition*.
De nuevo hay varias maneras de solucionar este problema. Una solucion es redefinir el interfaz como:

```
interface NumberSet
{
	boolean setNumber(long Number);
};
```

En este caso, *setNumber* se define como una operación que devuelve *true* si el numero no estaba ya ocupado. Esta operación es atomica, se asegura que solo una replica podra definir como ocupado un determinado numero.
La segunda solucion, mas genérica, implica el empleo de monitores o semáforos replicados. En este caso, a cada estructura replicada se le asocia un monitor replicado que permite secuencializar el acceso a la estructura de datos, y el codigo a emplear presentaria el siguiente aspecto:

```
{
	…
	testNumber.lock();
	while(testNumber(number))
		number++;
	setNumber(number);
	testNumber.unlock();
	…
}
```

Este enfoque permite un acesso generico a las estructuras de datos, pero puede resultar menos eficiente. Adicionalmente, en el caso anterior, implica que la operación *testNumber* debe estar replicada, y debe comunicarse a los demas miembros del grupo antes de procesarse.

Soporte de estructuras replicadas basicas en Sensei

Al desplazar el foco de replicación desde el interfaz general de la aplicación hacia sus estructuras de datos basicas, comprobamos la necesidad de soportar monitores replicados. Un monitor replicado es una de las estructuras replicadas basicas soportadas en sensei. Su interfaz es simple:

```
interface GroupMonitor : GroupMember
{
	void lock() raises (InvalidGroupException);
	void unlock() raises (InvalidGroupException);
};
```

La implementacion es tambien basica, siguiendo las pautas marcadas en el primer punto de este articulo y soportando codigo reentrante. Es decir, un mismo thread puede obtener el mismo monitor repetidamente. Si un monitor esta asignado a una replica que se cae y es expulsada del grupo, el monitor es automáticamente liberado.
El empleo de monitores no es suficiente en todos los casos. Si una replica tiene un monitor, realiza una serie de operaciones y se cae antes de liberar el monitor, la aplicación puede necesitar deshacer las operaciones realizadas por aquella replica. Es decir, puede precisar transacciones. Sensei soporta estructuras transaccionables basicas:

```
interface GroupTransactionable : GroupMember
{
	void startTransaction() raises (InvalidGroupException);
	void endTransaction() raises (InvalidGroupException);
};
```

Las estructuras anteriores estan disponibles para RMI y para CORBA.
Basado en Java RMI, JavaSoft ha desarrollado *JavaSpaces*, un paradigma de programación distribuida que no se basa en mensajes sino en el acceso a una estructura de datos distribuida denominada *Space*. Es una estructura basica, fundamentalmente un hash map, cuyas operaciones son definidas atómicamente y permitiendo transacciones. Las aplicaciones se comunican entre si modificando entradas en esa estructura de datos. La especificación de estos *Spaces* no incluyen su replicación para obtener tolerancia a fallos, aunque una implementacion especifica puede incluir esa replicación. Interesantemente, es posible implementar fácilmente esta tecnología usando UMA, aunque este no es el punto que pretendemos destacar aquí. La idea es que empleando transacciones y una estructura de datos (hash map) disponible distribuidamente es perfectamente posible implementar ciertas aplicaciones distribuidas, cuya logica queda mejor capturada mediante este paradigma que mediante el empleo de comunicaciones por mensajes.
A partir de esta idea, hemos incluido una tercera estructura basica en sensei: *GroupHashMap*, que replica el interfaz *Map* de Java. Para su desarrollo, ha sido suficiente con ejecutar UMA sobre este interfaz, que ha generado entonces todas las clases y mensajes necesarios. Evidentemente, podriamos realizar lo mismo con

todas las clases de colecciones en Java, y tener asi un *GroupVector*, *GroupList*, etc, pero esto puede realizarse en cualquier momento por el usuario de UMA, sin incrementar inútilmente el tamaNo de esta herramienta y clases asociadas.
La idea es basica: si se emplea RMI, es posible ejecutar UMA sobre la clase a replicar, que puede ser cualquiera de las clases o interfaces definidos en Java, tales como java.util.BitSet, java.util.Dictionary, etc, y automáticamente dispondremos de esa clase con soporte de replicación. Si se emplea CORBA, es necesario primero definir el interfaz CORBA en IDL. Puesto que se generar mensajes para todas las operaciones, resulta en general conveniente definir unicamente las operaciones que precisan replicación. Por ejemplo, un objeto usando un *Vector* no empleara probablemente todas las operaciones, y puede por lo tanto definir un nuevo interfaz que incluya solo las operaciones necesarias, en aras de una optimizacion del sistema.

Conclusiones

Sensei define un sistema de comunicaciones fiables que permite el desarrollo de aplicaciones fiables empleando RMI o CORBA. Incluye soporte de Ensemble de tal forma que las aplicaciones CORBA puedan emplear su protocolo de comunicaciones. SenseiUMA define una herramienta que permite replicar automáticamente una aplicación RMI o CORBA. Esto implica la posibilidad de generar automáticamente JavaBeans replicados, tolerantes a fallos. Y esto implica a su vez la posibilidad de generar componentes CORBA tolerantes a fallos de una forma automatica.
La palabra automatica puede estar quizas usada abusivamente aquí, de la misma forma que la palabra transparente se usa abusivamente al definir operaciones distribuidas bajo el modelo CORBA o RMI. El proceso puede ser ciertamente completamente automatico, pero la obtención de una aplicación replicada eficiente implicara identificar en cada caso las estructuras de datos basicas y reprogramar la aplicación en torno a un acceso concurrente a esas estructuras que ahora deben estar replicadas. Implica asimismo optimizar esas estructuras de datos para su uso por replicas, pero estos pasos son fácilmente identificables y faciles de realizar. Por lo tanto, SenseiUMA simplifica enormemente el desarrollo de aplicaciones fiables, permitiendo el desarrollo de JavaBeans, componentes CORBA o simples interfaces RMI o CORBA con soporte total de replicación.
El coste de esta replicación es por supuesto el rendimiento. Todo acceso al componente debe ser ahora secuencializado y comunicado al resto de replicas, implicando un tiempo de respuesta considerablemente mas lento, de varios ordenes de magnitud.
Este rendimiento, va a depender en gran medida del numero de replicas y del sustrato de comunicaciones empleado. Sensei emplea un algoritmo de comunicaciones de paso de token, ineficiente cuando el numero de replicas es elevado. Empleando Ensemble, es posible emplear un gran numero de replicas sin una importante merma en el rendimiento, y según las condiciones del entorno sera posible incluso usar UDP, evidentemente mas eficiente. Sin embargo, la aproximación de replicar activamente un componente para obtener tolerancia a fallos implicara normalmente un numero pequeño de replicas, y en ese caso el algoritmo de paso de token de Sensei es perfectamente valido.

Bibliografía

[Birman96] K.P. Birman, *Building Secure and Reliable Network Applications*, Prentice-Hall, 1996.
[orbos 00-01-19] *Fault Tolerance Join Revised Submission*. OMG TC Document orbos 99-12-19, 20 Dic. 1999