Introducción a la programación en Java RMI mediante ejemplos

Introducción

El objetivo de esta guía es presentar varios ejemplos muy sencillos que permitan familiarizarse con los aspectos básicos del desarrollo de programas que usan Java RMI.

Bajo ningún concepto se pretende que esta guía constituya un curso sobre cómo programar en este entorno, estando ya disponibles en Internet numerosos cursos sobre este tema (el "oficial"). Simplemente se pretende, de una forma muy directa y aplicada, que un programador familiarizado con Java sea capaz de realizar aplicaciones usando RMI en muy poco tiempo y sin necesidad de revisar mucha documentación.

Esta guía está basada en ejemplos que intentan recoger algunos de los usos más típicos de Java RMI.

- Un servicio básico: servicio de eco
- Control de la concurrencia: servicio de log
- Referencias remotas como parámetros (*callbacks*): servicio de chat
- Referencias remotas como valor retornado (fábricas de referencias remotas): servicio simple de banco
- Usando clases definidas por el usuario y clases complejas: servicio de banco
- Descarga dinámica de clases: servicio de banco extendido

En este enlace dispone de los ejemplos usados en esta guía.

Un servicio básico: servicio de eco

Dividiremos el desarrollo de este servicio en las siguientes etapas:

- Definición del servicio
- Implementación del servicio
- Desarrollo del servidor
- Desarrollo del cliente
- Compilación
- Ejecución

Definición del servicio

En RMI para crear un servicio remoto es necesario definir una interfaz que derive de la interfaz `Remote` y que contenga los métodos requeridos por ese servicio, especificando en cada uno de ellos que pueden activar la excepción `RemoteException`, usada por RMI para notificar errores relacionados con la comunicación.

Este primer servicio ofrece únicamente un método remoto que retorna la cadena de carácteres recibida como argumento pero pasándola a mayúsculas.

A continuación, se muestra el código de esta definición de servicio (fichero `ServicioEco.java`):

```
import java.rmi.*;

interface ServicioEco extends Remote {
        String eco (String s) throws RemoteException;
}
```

Implementación del servicio

Es necesario desarrollar el código que implementa cada uno de los servicios remotos. Ese código debe estar incluido en una clase que implemente la interfaz de servicio definida en la etapa previa. Para permitir que los métodos remotos de esta clase puedan ser invocados externamente, la opción más sencilla es definir esta clase como derivada de la clase `UnicastRemoteObject`. La principal limitación de esta alternativa es que, debido al modelo de herencia simple de Java, nos impide que esta clase pueda derivar de otra relacionada con la propia esencia de la aplicación (así, por ejemplo, con esta solución no podría crearme una clase que deriva a la vez de `Empleado` y que implemente un cierto servicio remoto). En esta guía usaremos esta opción. Consulte la referencia previamente citada para estudiar la otra alternativa (basada en usar el método estático `exportObject` de `UnicastRemoteObject`).

Recapitulando, desarrollaremos una clase derivada de `UnicastRemoteObject` y que implemente la interfaz remota `ServicioEco` (fichero `ServicioEcoImpl.java`):

```
import java.rmi.*;
import java.rmi.server.*;

class ServicioEcoImpl extends UnicastRemoteObject implements ServicioEco {
    ServicioEcoImpl() throws RemoteException {
    }
    public String eco(String s) throws RemoteException {
        return s.toUpperCase();
    }
}
```

Observe la necesidad de hacer explícito el constructor para poder declarar que éste puede generar la excepción `RemoteException`.

Es importante entender que todos los objetos especificados como parámetros de un método remoto, así como el retornado por el mismo, se pasan por valor, y no por referencia como ocurre cuando se realiza una invocación a un método local. Esta característica tiene como consecuencia que cualquier cambio que se haga en el servidor sobre un objeto recibido como parámetro no afecta al objeto original en el cliente. Por ejemplo, este método remoto no llevará a cabo la labor que se le supone, aunque sí lo haría en caso de haber usado ese mismo código (sin la excepción `RemoteException`, evidentemente) para definir un método local.

```
    public void vuelta(StringBuffer s) throws RemoteException {
        s.reverse();
```

```
    }
```

Un último aspecto que conviene resaltar es que la clase que implementa la interfaz remota es a todos los efectos una clase convencional y, por tanto, puede incluir otros métodos, además de los especificados en la interfaz. Sin embargo, esos métodos no podrán ser invocados directamente por los clientes del servicio.

Desarrollo del servidor

El programa que actúe como servidor debe iniciar el servicio remoto y hacerlo públicamente accesible usando, por ejemplo, el `rmiregistry` (el servicio básico de *binding* en Java RMI). Nótese que se podría optar por usar la misma clase para implementar el servicio y para activarlo pero se ha preferido mantenerlos en clases separadas por claridad.

A continuación, se muestra el código del servidor (fichero `ServidorEco.java`):

```
import java.rmi.*;
import java.rmi.server.*;

class ServidorEco  {
    static public void main (String args[]) {
       if (args.length!=1) {
            System.err.println("Uso: ServidorEco numPuertoRegistro");
            return;
        }
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            ServicioEcoImpl srv = new ServicioEcoImpl();
            Naming.rebind("rmi://localhost:" + args[0] + "/Eco", srv);
        }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " + e.toString());
            System.exit(1);
        }
        catch (Exception e) {
            System.err.println("Excepcion en ServidorEco:");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Resaltamos los siguientes aspectos de ese código:

- El programa asume que ya está arrancado el `rmiregistry` previamente (otra opción hubiera sido que lo arrancase el propio programa usando el método estático `createRegistry` de `LocateRegistry`). En la sección que explica cómo ejecutar el programa se muestra el procedimiento para arrancar el `rmiregistry`. Nótese que el programa espera recibir como único argumento el número de puerto por que el que está escuchando el `rmiregistry`.
- Un aspecto clave en Java RMI es la seguridad. En el código del servidor se puede apreciar cómo éste instancia un gestor de seguridad (más sobre el tema en la sección dedicada a la ejecución del programa). Para ejemplos sencillos, podría eliminarse esta parte del código del servidor (y del cliente) pero es conveniente su uso para controlar mejor la seguridad y es un requisito en caso de que la aplicación requiera carga dinámica de clases.
- La parte principal de este programa está incluida en la sentencia `try` y consiste en crear un objeto de la clase que implementa el servicio remoto y darle de alta en el `rmiregistry` usando el método estático `rebind` que permite especificar la operación usando un formato de tipo URL. Nótese que el `rmiregistry` sólo permite que se den de alta servicios que ejecutan en su misma máquina.

Desarrollo del cliente

El cliente debe obtener una referencia remota (es decir, una referencia que corresponda a un objeto remoto) asociada al servicio para luego simplemente invocar de forma convencional sus métodos, aunque teniendo en cuenta que pueden generar la excepción `RemoteException`. En este ejemplo, la referencia la obtiene a través del `rmiregistry`.

A continuación, se muestra el código del cliente (fichero `ClienteEco.java`):

```
import java.rmi.*;
import java.rmi.server.*;

class ClienteEco {
    static public void main (String args[]) {
        if (args.length<2) {
            System.err.println("Uso: ClienteEco hostregistro numPuertoRegistro ...");
            return;
        }

        if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());

        try {

            ServicioEco srv = (ServicioEco) Naming.lookup("//" + args[0] + ":" + args[1] + "/Eco");

            for (int i=2; i<args.length; i++)
                System.out.println(srv.eco(args[i]));
        }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " + e.toString());
        }
        catch (Exception e) {
            System.err.println("Excepcion en ClienteEco:");
            e.printStackTrace();
        }
```

```
    }
}
```

Resaltamos los siguientes aspectos de ese código:

- El programa espera recibir como argumentos la máquina donde ejecuta `rmiregistry`, así como el puerto por el que escucha. El resto de los argumentos recibidos por el programa son las cadenas de caracteres que se quieren pasar a mayúsculas.
- Como ocurre con el servidor, es conveniente, e incluso obligatorio en caso de descarga dinámica de clases, la activación de un gestor de seguridad.
- El programa usa el método estático `lookup` para obtener del `rmiregistry` una referencia remota del servicio. Observe el uso de la operación de *cast* para adaptar la referencia devuelta por `lookup`, que corresponde a la interfaz `Remote`, al tipo de interfaz concreto, derivado de `Remote`, requerido por el programa (`ServicioEco`).
- Una vez obtenida la referencia remota, la invocación del método es convencional, requiriendo el tratamiento de las excepciones que puede generar.

Compilación

El proceso de compilación tanto del cliente como del servidor es el habitual en Java. El único punto que conviene resaltar es que para generar el programa cliente, además de la(s) clase(s) requerida(s) por la funcionalidad del mismo, se debe disponer del fichero `class` que define la interfaz (en este caso, `ServicioEco.class`), tanto para la compilación como para la ejecución del cliente. Esto se ha resuelto en este ejemplo creando un enlace simbólico. Si quiere probar el ejemplo usando dos máquinas, lo que recomendamos, deberá copiar el fichero `class` a la máquina donde se ejecutará el cliente. Obsérvese que no es necesario, ni incluso conveniente, disponer en el cliente de las clases que implementan el servicio.

Hay que resaltar que en la versión actual de Java (realmente, desde la versión 1.5) no es necesario usar ninguna herramienta para generar resguardos ni para el cliente (*proxy*) ni para el servidor (*skeleton*). En versiones anteriores, había que utilizar la herramienta `rmic` para generar las clases que realizan esta labor, pero gracias a la capacidad de reflexión de Java, este proceso ya no es necesario.

En el ejemplo que nos ocupa, dado que, por simplicidad, no se han definido paquetes ni se usan ficheros JAR, para generar el programa cliente y el servidor, basta con entrar en los directorios respectivos y ejecutar directamente:

```
javac *.java
```

Ejecución

Antes de ejecutar el programa, hay que arrancar el registro de Java RMI (`rmiregistry`). Este proceso ejecuta por defecto usando el puerto 1099, pero puede especificarse como argumento al arrancarlo otro número de puerto, lo que puede ser lo más conveniente para evitar colisiones en un entorno donde puede haber varias personas probando aplicaciones Java RMI en la misma máquina.

Hay que tener en cuenta que el `rmiregistry` tiene que conocer la ubicación de las clases de servicio. Para ello, puede necesitarse definir la variable de entorno `CLASSPATH` para el `rmiregistry` de manera que haga referencia a la localización de dichas clases. En cualquier caso, si el `rmiregistry` se arranca en el mismo directorio donde ejecutará posteriormente el servidor y en la programación del mismo no se han definido nuevos paquetes (todas las clases involucradas se han definido en el paquete por defecto), no es necesario definir esa variable de entorno. Así ocurre en este ejemplo:

```
cd servidor
rmiregistry 54321 &
```

Ya estamos a punto de poder ejecutar el servidor y el cliente, y si puede ser, mejor en dos máquinas diferentes. Sin embargo, queda un último aspecto vinculado con la seguridad. Dado que tanto en el cliente como en el servidor se ha activado un gestor de seguridad, si queremos poder definir nuestra propia política de seguridad, con independencia de la que haya definida por defecto en el sistema, debemos crear nuestros propios ficheros de políticas de seguridad.

Dado que estamos trabajando en un entorno de pruebas, lo más razonable es crear ficheros de políticas de seguridad que otorguen todos los permisos posibles tanto para el cliente (fichero que hemos llamada `cliente.permisos`) como para el servidor (fichero `servidor.permisos`):

```
grant  {
    permission java.security.AllPermission;
};
```

Dada la importancia de esta cuestión, se recomienda que el lector revise más en detalle la misma en las numerosas fuentes disponibles en Internet que tratan este tema.

Procedemos finalmente a la ejecución del servidor:

```
cd servidor
java -Djava.security.policy=servidor.permisos  ServidorEco 54321
```

Y la del cliente:

```
cd cliente
java -Djava.security.policy=cliente.permisos  ClienteEco localhost 54321 hola adios
HOLA
ADIOS
```

Control de la concurrencia: servicio de log

Un mecanismo de comunicación de tipo RPC o RMI no sólo libera al programador de todos los aspectos relacionados con la mensajería, sino también de todas las cuestiones vinculadas con el diseño de un servicio concurrente.

Java RMI se encarga automáticamente de desplegar los *threads* requeridos para dotar de concurrencia a un servicio implementado usando esta tecnología. Aunque esta característica es beneficiosa, el programador debe de ser consciente de que los métodos remotos en el servidor se ejecutan de manera concurrente, debiendo establecer mecanismos de sincronización en caso de que sea necesario.

Esta concurrencia automática hace que, como puede observarse en el ejemplo previo, el programa servidor no termine cuando completa su código, sino que se quede esperando indefinidamente la llegada de peticiones. Nótese cómo en el tratamiento de las excepciones se usa una llamada a `System.exit` para completar explícitamente su ejecución.

Este ejemplo intenta ilustrar esta cuestión creando un hipotético servicio de *log* que ofrece un método que permite al cliente enviar un mensaje al servidor para que lo almacene de alguna forma (fichero `ServicioLog.java`):

```
import java.rmi.*;

interface ServicioLog extends Remote {
        void log (String m) throws RemoteException;
}
```

Para ilustrar la cuestión que nos ocupa, este método va a enviar el mensaje a dos destinos: a la salida estándar del programa servidor y a un fichero especificado como argumento del programa servidor. Esta duplicidad un poco artificial pretende precisamente mostrar la *no atomicidad* en la ejecución de los servicios remotos.

A continuación, se muestra la clase que implementa esta interfaz remota (fichero `ServicioLogImpl.java`):

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

class ServicioLogImpl extends UnicastRemoteObject implements ServicioLog {
    PrintWriter fd;
    ServicioLogImpl(String f) throws RemoteException {
        try {
             fd = new PrintWriter(f);
        }
        catch (FileNotFoundException e) {
            System.err.println(e);
            System.exit(1);
        }
    }
    public void log(String m) throws RemoteException {
        System.out.println(m);
        fd.println(m);
        fd.flush();  // para asegurarnos de que no hay buffering
    }
}
```

No se incluye en este documento el código del servidor ni del cliente puesto que no aportan información adicional relevante.

Para ilustrar el carácter concurrente del servicio de Java RMI, se propone arrancar simultáneamente dos clientes que envíen un número elevado de mensajes (se muestra un extracto del fichero `ClienteLog.java`):

```
            for (int i=0; i<10000; i++)
                srv.log(args[2] + " " + i);
```

Se pretende comprobar que los mensajes pueden quedar en orden diferente en la salida estándar y en el fichero precisamente por la ejecución concurrente del método `log`.

A continuación, se muestra una ejecución donde se aprecia este problema:

```
cd servidor
remiregistry 54321 &
java -Djava.security.policy=servidor.permisos  ServidorLog 54321 fichero > salida &
cd ../cliente
java -Djava.security.policy=cliente.permisos  ClienteLog localhost 54321 yo &
java -Djava.security.policy=cliente.permisos  ClienteLog localhost 54321 tu
diff fichero salida
2544d2543
< tu 714
2545a2545
> tu 714
9985d9984
< yo 5997
9986a9986
> yo 5997
15325a15326
> yo 8469
15444d15444
< yo 8469
17708a17709
> tu 8229
17985d17985
< tu 8229
```

La solución en este caso es la habitual en Java: marcar el método como sincronizado:

```
    public synchronized void log(String m) throws RemoteException {
```

Referencias remotas como parámetros (*callbacks*): servicio de chat

En los ejemplos previos, los clientes obtenían las referencias remotas de servicios a través del `rmiregistry`. Sin embargo, teniendo en cuenta que estas referencias son objetos Java convencionales, éstas se pueden recibir también como parámetros de un método, como se ilustra en esta sección, o como valor de retorno del mismo, tal como se mostrará en la siguiente. De esta forma, se podría decir que el `rmiregistry` sirve como punto de contacto inicial para obtener la primera referencia remota, pero que, a continuación, los procesos implicados pueden pasarse referencias remotas adicionales entre sí.

Es importante resaltar que cuando se especifica como parámetro de un método RMI una referencia remota, a diferencia de lo que ocurre con el resto de los objetos, que se transfieren por valor, ésta se pasa por referencia (se podría decir que se pasa por valor la propia referencia).

Para ilustrar el uso de referencias remotas como parámetros se plantea en esta sección un servicio de chat. Este servicio permitirá que cuando un usuario, identificado por un apodo, se conecte al mismo, reciba todo lo que escriben el resto de los usuarios conectados y, a su vez, éstos reciban todo lo que escribe el mismo.

Esta aplicación se va a organizar con procesos clientes que atienden a los usuarios y un servidor que gestiona la información sobre los clientes/usuarios conectados.

De manera similar a los ejemplos previos, el servidor ofrecerá un servicio remoto para darse de alta y de baja, así como para enviarle la información que escribe cada usuario.

Sin embargo, en este caso, se requiere, además, que los clientes ofrezcan una interfaz remota para ser notificados de lo que escriben los otros clientes.

A continuación, se muestra la interfaz remota proporcionada por el servidor (fichero `ServicioChat`):

```
import java.rmi.*;

interface ServicioChat extends Remote {
    void alta(Cliente c) throws RemoteException;
    void baja(Cliente c) throws RemoteException;
    void envio(Cliente c, String apodo, String m) throws RemoteException;
}
```

El tipo `Cliente` que aparece como parámetro de los métodos corresponde a una interfaz remota implementada por el cliente y que permite notificar a un usuario de los mensajes recibidos por otros usuarios (a esta llamada a contracorriente, del servidor al cliente, se le suele denominar *callback*). Se trata, por tanto, de una referencia remota recibida como parámetro, sin necesidad de involucrar al `rmiregistry` en el proceso. A continuación, se muestra esa interfaz remota proporcionada por el cliente (fichero `Cliente`):

```
import java.rmi.*;

interface Cliente extends Remote {
    void notificacion(String apodo, String m) throws RemoteException;
}
```

Pasamos a la implementación del servicio de chat (fichero `ServicioChatImpl.java`) que usa un contenedor de tipo lista para guardar los clientes conectados:

```
import java.util.*;
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

class ServicioChatImpl extends UnicastRemoteObject implements ServicioChat {
    List<Cliente> l;
    ServicioChatImpl() throws RemoteException {
        l = new LinkedList<Cliente>();
    }
    public void alta(Cliente c) throws RemoteException {
        l.add(c);
    }
    public void baja(Cliente c) throws RemoteException {
        l.remove(l.indexOf(c));
    }
    public void envio(Cliente esc, String apodo, String m)
      throws RemoteException {
        for (Cliente c: l)
            if (!c.equals(esc))
                c.notificacion(apodo, m);
    }
}
```

Obsérvese como en `envio` se invoca el método `notificacion` de todos los clientes (es decir, de todas las interfaces remotas de cliente) en la lista, exceptuando la del propio escritor.

No se muestra el código del servidor (fichero `ServidorChat`) puesto que es similar a todos los servidores de los ejemplos previos. Sin embargo, sí es interesante el código del cliente (fichero `ClienteChat`), puesto que tiene que hacer el doble rol de cliente y de servidor: debe buscar en el `rmiregistry` el servicio de chat remoto, pero también tiene que instanciar un objeto que implemente la interfaz remota de cliente.

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

class ClienteChat {
    static public void main (String args[]) {
        if (args.length!=3) {
            System.err.println("Uso: ClienteChat hostregistro numPuertoRegistro apodo");
            return;
        }

       if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());

        try {
```

```
            ServicioChat srv = (ServicioChat) Naming.lookup("//" + args[0] + ":" + args[1] + "/Chat");
            ClienteImpl c = new ClienteImpl();
            srv.alta(c);
            Scanner ent = new Scanner(System.in);
            String apodo = args[2];
            System.out.print(apodo + "> ");
            while (ent.hasNextLine()) {
                srv.envio(c, apodo, ent.nextLine());
                System.out.print(apodo + "> ");
            }
            srv.baja(c);
            System.exit(0);
        }
        catch (RemoteException e) {
            System.err.println("Error de comunicacion: " + e.toString());
        }
        catch (Exception e) {
            System.err.println("Excepcion en ClienteChat:");
            e.printStackTrace();
        }
    }
```

Nótese que al tener también un perfil de servidor, es necesario terminar su ejecución explícitamente con `System.exit`.

Por último, se muestra la implementación (fichero `ClienteImpl.java`):

```
import java.rmi.*;
import java.rmi.server.*;

class ClienteImpl extends UnicastRemoteObject implements Cliente {
    ClienteImpl() throws RemoteException {
    }
    public void notificacion(String apodo, String m) throws RemoteException {
        System.out.println("\n" + apodo + "> " + m);
    }
}
```

Hay que resaltar que el método `notificacion` se ejecutará de forma asíncrona con respecto al flujo de ejecución del cliente.

Con respecto a la compilación y ejecución de estos programas, en este caso es necesario también disponer en la máquina que ejecuta el servidor del fichero `class` correspondiente a la interfaz remota de cliente (`Cliente.class`).

Referencias remotas como valor retornado (fábricas de referencias remotas): servicio simple de banco

Además de poder ser recibidas como parámetros de un método, puede obtenerse una referencia remota como el valor de retorno de un método (al fin y al cabo, eso es lo que hace el método `lookup` del `rmiregistry`).

Dentro de este ámbito, es muy frecuente el uso de un esquema de tipo *fábrica de referencias remotas*. Este esquema se suele usar cuando se requiere ir creando dinámicamente objetos remotos (por ejemplo, un objeto que actúe como cerrojo). Con este modelo, el servidor crea, y registra en el `rmiregistry`, un servicio remoto que ofrece una operación para crear un nuevo objeto remoto (en el ejemplo, un servicio de *fabricación* de cerrojos con un método para crear uno nuevo). Esa operación instancia un nuevo objeto remoto y retorna una referencia remota al mismo.

Para ilustrar este escenario típico, vamos a crear un servicio bancario trivial, que permite crear dinámicamente cuentas bancarias.

A continuación, se muestra la interfaz remota correspondiente a la *fábrica* de cuentas (fichero `Banco.java`), que será la que se registre en el `rmiregistry`:

```
import java.rmi.*;

interface Banco extends Remote {
    Cuenta crearCuenta(String nombre) throws RemoteException;
}
```

La clase que implementa esta interfaz (fichero `BancoImpl`) meramente crea un nuevo objeto que implementa la interfaz remota `Cuenta`:

```
import java.rmi.*;
import java.rmi.server.*;

class BancoImpl extends UnicastRemoteObject implements Banco {
    BancoImpl() throws RemoteException {
    }
    public Cuenta crearCuenta(String nombre) throws RemoteException {
        return new CuentaImpl(nombre);
    }
}
```

La interfaz remota correspondiente a una cuenta bancaria (fichero `Cuenta`) especifica unos métodos para operar, hipotéticamente, con esa cuenta:

```
import java.rmi.*;

interface Cuenta extends Remote {
    String obtenerNombre() throws RemoteException;
    float obtenerSaldo() throws RemoteException;
    float operacion(float valor) throws RemoteException;
}
```

Y, a continuación, se incluye la clase (fichero `CuentaImpl`) que implementa esos métodos:

```
import java.rmi.*;
import java.rmi.server.*;

class CuentaImpl extends UnicastRemoteObject implements Cuenta {
    private String nombre;
    private float saldo = 0;
    CuentaImpl(String n) throws RemoteException {
        nombre = n;
    }
    public String obtenerNombre() throws RemoteException {
        return nombre;
    }
    public float obtenerSaldo() throws RemoteException {
        return saldo;
    }
    public float operacion(float valor) throws RemoteException {
        saldo += valor;
        return saldo;
    }
}
```

Dado que tanto el cliente (fichero `ClienteBanco.java`) como el servidor (fichero `ServidorBanco.java`) son similares a los de los ejemplos previos, no se incluye su código en este documento. Simplemente, se muestra un extracto del cliente para mostrar el uso de este servicio:

```
            Banco srv = (Banco) Naming.lookup("//" + args[0] + ":" + args[1] + "/Banco");
            Cuenta c = srv.crearCuenta(args[2]);
            c.operacion(30);
            System.out.println(c.obtenerNombre() + ": " + c.obtenerSaldo());
```

Por último, es conveniente hacer algún comentario sobre el ciclo de vida de los objetos remotos creados dinámicamente. Al igual que ocurre con cualquier objeto en Java, el objeto seguirá *vivo* mientras haya alguna referencia al mismo. En el caso de Java RMI, esto se extiende a toda la aplicación distribuida: el objeto que implementa una interfaz remota seguirá vivo mientras haya una referencia local o remota al mismo. Java RMI, por tanto, implementa un recolector de basura distribuido para poder hacer un seguimiento de la evolución de los objetos remotos.

Si el proceso que ha instanciado un objeto remoto desea saber cuándo no quedan más referencias remotas a ese objeto en el sistema, aunque sí pueda existir alguna referencia local (por ejemplo, porque ese proceso ha incluido el objeto remoto en algún contenedor), puede implementar la interfaz `Unreferenced` y será notificado, invocándose el método `unreferenced` de dicha interfaz, cuando ocurra esa circunstancia. Nótese que la detección y notificación de ese estado puede diferirse considerablemente (por defecto, puede retrasarse diez minutos, aunque se puede reducir ese valor cambiando la propiedad `java.rmi.dgc.leaseValue`, lo que puede implicar, sin embargo, mayor sobrecarga de mensajes de estado en el sistema).

Usando clases definidas por el usuario y clases complejas: servicio de banco

Tanto los parámetros de un método RMI como el resultado devuelto por el mismo pueden ser objetos de cualquier tipo, siempre que sean *serializables* (la mayoría de los objetos lo son, excepto aquéllos que por su propia esencia estén vinculados con recursos locales y no tenga sentido transferirlos a otra máquina como, por ejemplo, un descriptor de fichero, un *thread* o un *socket*). Por tanto, en un método RMI se pueden usar objetos de clases definidas por el usuario y objetos de clases complejas, como puede ser un contenedor: el proceso de *serialización* se encarga de *empaquetar* toda la información vinculada con ese objeto, de manera que luego se pueda recuperar en el mismo estado.

En esta sección, vamos a extender el servicio bancario previo de manera que utilice una clase definida por el usuario, así como una clase compleja, como puede ser una lista de cuentas bancarias.

En esta nueva versión, una cuenta bancaria va a quedar identificada por el nombre y el número de identificación del titular, en lugar de sólo por el nombre. Esta doble identificación va a quedar englobada en una nueva clase que representa al titular de una cuenta (fichero `Titular`):

```
import java.io.*;

class Titular implements Serializable {
     private String nombre;
     private String iD;
     Titular(String n, String i) {
         nombre = n;
         iD = i;
     }
     public String obtenerNombre() {
         return nombre;
     }
     public String obtenerID() {
         return iD;
     }
     public String toString() {
         return nombre + " | " + iD;
     }
}
```

Como se puede observar, se trata de una clase trivial pero que presenta una característica importante: dado que se van a usar objetos de esta clase como parámetros y valores de retorno de métodos RMI, es necesario especificar que esta clase implemente la interfaz `Serializable`, declarando así el programador que esa clase puede *serializarse*.

En esta nueva versión, la interfaz que declara una cuenta (fichero `Cuenta.java`) queda:

```
import java.rmi.*;

interface Cuenta extends Remote {
```

```
    Titular obtenerTitular() throws RemoteException;
    float obtenerSaldo() throws RemoteException;
    float operacion(float valor) throws RemoteException;
}
```

Y su implementación (fichero `CuentaImpl.java`):

```
import java.rmi.*;
import java.rmi.server.*;

class CuentaImpl extends UnicastRemoteObject implements Cuenta {
    private Titular tit;
    private float saldo = 0;
    CuentaImpl(Titular t) throws RemoteException {
        tit = t;
    }
    public Titular obtenerTitular() throws RemoteException {
        return tit;
    }
    public float obtenerSaldo() throws RemoteException {
        return saldo;
    }
    public float operacion(float valor) throws RemoteException {
        saldo += valor;
        return saldo;
    }
}
```

Por otro lado, se ha modificado el servicio bancario para que almacene las cuentas creadas en un contenedor (concretamente, en una lista enlazada como en el ejemplo del servicio de chat) y ofrezca un nuevo método remoto que devuelta toda la lista de cuentas.

A continuación, se muestra la interfaz remota correspondiente a esta *fábrica* de cuentas (fichero `Banco.java`):

```
import java.rmi.*;
import java.util.*;

interface Banco extends Remote {
    Cuenta crearCuenta(Titular t) throws RemoteException;
    List<Cuenta> obtenerCuentas() throws RemoteException;
}
```

Y la clase que implementa esta interfaz (fichero `BancoImpl`):

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

class BancoImpl extends UnicastRemoteObject implements Banco {
    List<Cuenta> l;
    BancoImpl() throws RemoteException {
        l = new LinkedList<Cuenta>();
    }
    public Cuenta crearCuenta(Titular t) throws RemoteException {
        Cuenta c = new CuentaImpl(t);
        l.add(c);
        return c;
    }
    public List<Cuenta> obtenerCuentas() throws RemoteException {
       return l;
    }
}
```

Nótese cómo el nuevo método retorna directamente una lista. El proceso de *serialización* en el que se apoya RMI reconstruye automáticamente esa lista de referencias remotas en la JVM del cliente.

Nuevamente, no se incluye el código del cliente (fichero `ClienteBanco.java`) ni del servidor (fichero `ServidorBanco.java`) puesto que no aportan información novedosa. Simplemente se incluye un extracto del cliente para mostrar el uso del nuevo método para imprimir los datos de todas las cuentas:

```
            Banco srv = (Banco) Naming.lookup("//" + args[0] + ":" + args[1] + "/Banco");
            Titular tit = new Titular(args[2], args[3]);
            Cuenta c = srv.crearCuenta(tit);
            c.operacion(30);

            List <Cuenta> l;
            l = srv.obtenerCuentas();
            for (Cuenta i: l) {
                Titular t = i.obtenerTitular();
                System.out.println(t + ": " + i.obtenerSaldo());
            }
```

En este punto se considera conveniente incidir en la diferencia que existe entre cuando se usan en un método RMI referencias a objetos remotos y cuando se utilizan referencias a objetos convencionales.

Vamos a fijarnos en ese extracto previo en la variable `c` y en la variable `t`. Ambas almacenan una referencia a un objeto retornado por un método RMI (`crearCuenta` y `obtenerTitular`, respectivamente). Sin embargo, hay una diferencia muy importante:

- La variable `c` guarda una referencia a un objeto remoto que implementa la interfaz `Cuenta`. Todas las llamadas a métodos que se hagan usando esa referencia (por ejemplo, `c.operacion(30)`) se convierten en operaciones remotas que, en caso de que provoquen algún tipo de actualización, repercutirá directamente sobre el objeto en el servidor.
- La variable `t` guarda una referencia a un objeto local que es una copia del objeto en el servidor. Todas las llamadas a métodos que se hagan utilizando esa referencia (nótese que la sentencia `println(t...` invoca automáticamente al método `toString` del objeto) serán operaciones locales y, por tanto, en caso de que causen algún tipo de actualización, no afectará al objeto en el servidor.

Como comentario final, tenga en cuenta que el fichero `class` que define el titular de una cuenta (fichero `Titular.class`) tiene que estar presente en las máquinas donde se generarán y ejecutarán los programas cliente y servidor.

Descarga dinámica de clases: servicio de banco extendido

En los ejemplos planteados hasta el momento no se ha explotado una de las características más interesantes de Java RMI: la descarga dinámica de clases. Este potente mecanismo permite que un proceso pueda usar el código de clases que no estaban presentes en su JVM cuando inició su ejecución, descargándose automáticamente en tiempo de ejecución desde otra JVM. Basándose en este mecanismo se pueden implementar técnicas sofisticadas tales como, por ejemplo, la incorporación automática de nuevos protocolos o la instalación automática en un cliente del manejador de un nuevo dispositivo. Esta técnica está en el corazón de muchas tecnologías distribuidas de Java, como, por ejemplo, Jini, y con ella se podría decir que se extiende al sistema distribuido el poder de la orientación a objetos, la herencia y el polimorfismo.

Podría parecer a priori que en los ejemplos previos ya había descargas automáticas de información entre clientes y servidores, pero se correspondían con transferencias de objetos, no de clases. Para poder usar en un método RMI dentro de una determinada JVM un objeto de un cierto tipo (ya sea clase o interfaz), era necesario disponer de la definición de ese tipo (fichero `class`) en esa JVM.

Pero entonces, ¿dónde surge la necesidad de la carga dinámica de clases y cómo se artícula?

Pensemos en qué ocurriría si a un método RMI se le pasa un objeto que es de un tipo derivado del especificado como parámetro formal (por ejemplo, el parámetro formal de un método RMI es de tipo `FormaGeometrica` y el parámetro real es de tipo `Circulo`). El proceso que implementa ese método remoto debe disponer en su JVM de la clase derivada correspondiente; pero, gracias al mecanismo de descarga dinámica de clases, no es necesario que exista a priori en su JVM, sino que puede descargarlo dinámicamente desde la máquina que posee la definición de ese subtipo. Se podría decir que el servidor va *aprendiendo* a hacer nuevas cosas dinámicamente (en el ejemplo geométrico, el servidor *aprende* a manejar círculos).

Para ilustrar este comportamiento, se ha extendido el ejemplo del banco de manera que un cierto cliente (directorio `cliente1`) va a extender la clase `Titular` para el caso de que la persona que posee una cuenta sea menor de edad. La clase derivada (fichero `TitularMenor.java`) incorpora un nuevo campo para incluir el nombre del tutor responsable, así como un nuevo método para leer este campo y una sobrescritura del método `toSting` para incorporar la nueva información:

```
import java.io.*;

class TitularMenor extends Titular {
     private String nombreTutor;
     TitularMenor(String n, String i, String t) {
         super(n, i);
         nombreTutor = t;
     }
     public String obtenerTutor() {
         return nombreTutor;
     }
     public String toString() {
         return super.toString() + " | " + nombreTutor;
     }
}
```

No ha habido modificaciones en ninguna de las clases del ejemplo de la sección anterior. A continuación, se muestra la única línea del cliente (fichero `ClienteBanco.java` en el directorio `cliente1`) que ha cambiado a la hora de crear una nueva cuenta para un menor:

```
            Titular tit = new TitularMenor(args[2], args[3], args[4]);
```

Gracias a este mecanismo, el servidor y otros clientes, sin ninguna modificación, pueden gestionar objetos de la clase derivada sin disponer a priori de su código, de manera que cuando invoquen un método sobrescrito en la subclase se ejecute esa nueva versión.

Así, una vez creadas nuevas cuentas bancarias para menores, el cliente original incluido en el directorio `cliente2` podrá imprimir correctamente todos sus datos a pesar de que no sabe nada de las cuentas para menores.

Sin embargo, para que esta descarga dinámica funcione, a la hora de ejecutar el programa que reside en la JVM que incluye esa nueva clase, debe especificarse la propiedad `java.rmi.server.codebase` para indicar la URL dónde está almacenada esa clase:

```
java -Djava.rmi.server.codebase=file:$PWD/ -Djava.security.policy=cliente.permisos  ClienteBanco localhost 54321  NombreMenor ID NombreTutor
```

Y habilitar esa descarga en el programa que recibirá la nueva clase poniendo a falso para ello la propiedad `java.rmi.server.useCodebaseOnly`en la activación del servidor:

```
java -Djava.rmi.server.useCodebaseOnly=false -Djava.security.policy=servidor.permisos ServidorBanco 54321
```

NOTA: Para que funcionara la descarga dinámica, ha sido necesario modificar la clase base `Titular` para añadirla un calificador `public`:

```
public class Titular implements Serializable {
```