Aspectos Avanzados de RMI

1. Introducción

Java RMI se describió como ejemplo de un sistema de objetos distribuidos.

Este capítulo analizará algunas de las características avanzadas de RMI más interesantes, a saber, descarga de resguardo, gestores de seguridad, y callback de cliente. Aunque no se trata de características inherentes del paradigma de objetos distribuidos, se trata de mecanismos que pueden ser útiles para los desarrolladores de aplicaciones.

2. Callback de cliente

Considérese una aplicación RMI donde un servidor de objeto debe notificar a los procesos participantes la ocurrencia de algún evento. Como ejemplos, en un chat, cuando un nuevo participante entra, se avisa al resto de los participantes de este hecho; en un sistema de subastas en tiempo real, cuando empiezan las ofertas, se debe avisar a los procesos participantes. Dentro del entorno API básica de RMI es imposible que el servidor inicie una llamada al cliente para transmitirle alguna clase de información que esté disponible, debido a que una llamada a método remoto es unidireccional (del cliente al servidor). Una forma de llevar a cabo la transmisión de información es que cada proceso cliente realice un **sondeo** al servidor de objeto, invocando de forma repetida un método remoto, que supóngase que se llama *haComenzadoOferta*, hasta que el método devuelva el valor booleano verdadero:

```
InterfazServidor h =
   (InterfazServidor) Naming.lookup(URLRegistro);
while (!(h.haComenzadoOferta()) {;}
   // comienza la oferta
```

El **sondeo** (**polling**) es de hecho una técnica empleada en muchos programas de red. Pero se trata de una técnica muy costosa en términos de recursos del sistema, ya que cada invocación de un método remoto implica un hilo separado en la máquina servidora, además de los recursos de sistema que su ejecución conlleva. Una técnica más eficiente se denomina ***callback***: permite que cada cliente de objeto interesado en la ocurrencia de un evento se registre a sí mismo con el servidor de objeto, de forma que el servidor inicie una invocación de un método remoto del cliente de objeto cuando dicho evento ocurra.

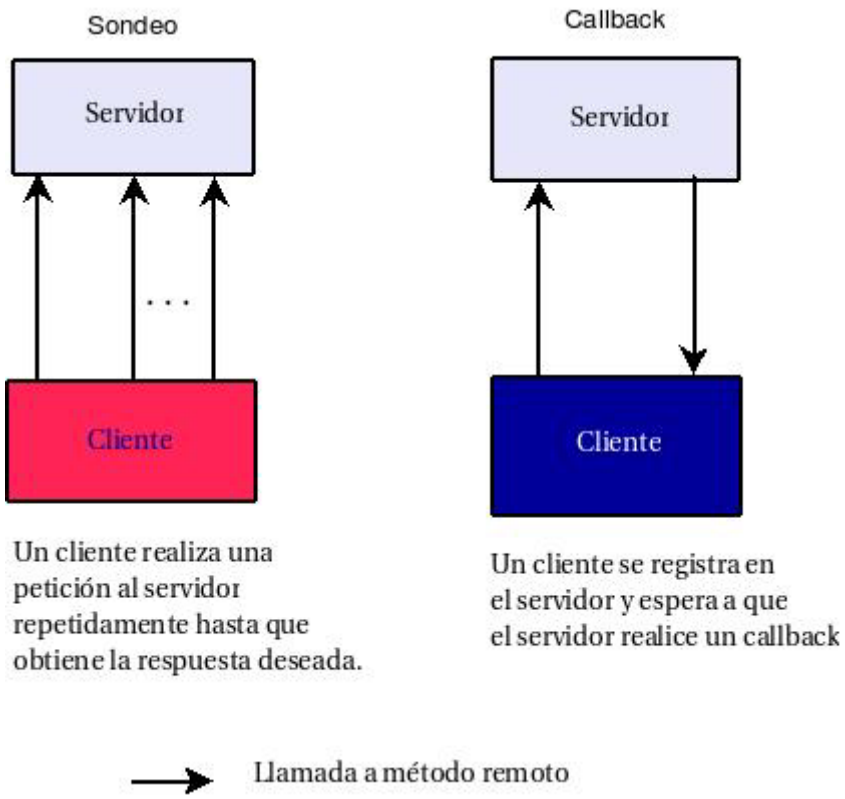


Figura 1. Sondeo (polling) frente a callback.

En RMI, el ***callback* de cliente** es una característica que permite a un cliente de objeto registrarse a sí mismo con un servidor de objeto remoto para ***callbacks***, de forma que el servidor puede llevar a cabo una invocación del método del cliente cuando el evento ocurra. Hay que observar que con los *callbacks* de clientes, las invocaciones de los métodos remotos se convierten en bidireccionales, o *dúplex*, desde el cliente al servidor y viceversa.

Cuando un servidor de objeto realiza un *callback*, los papeles de los dos procesos se invierten: el servidor de objeto se convierte en cliente de objeto, debido a que el primero inicia una invocación de método remoto en el segundo.

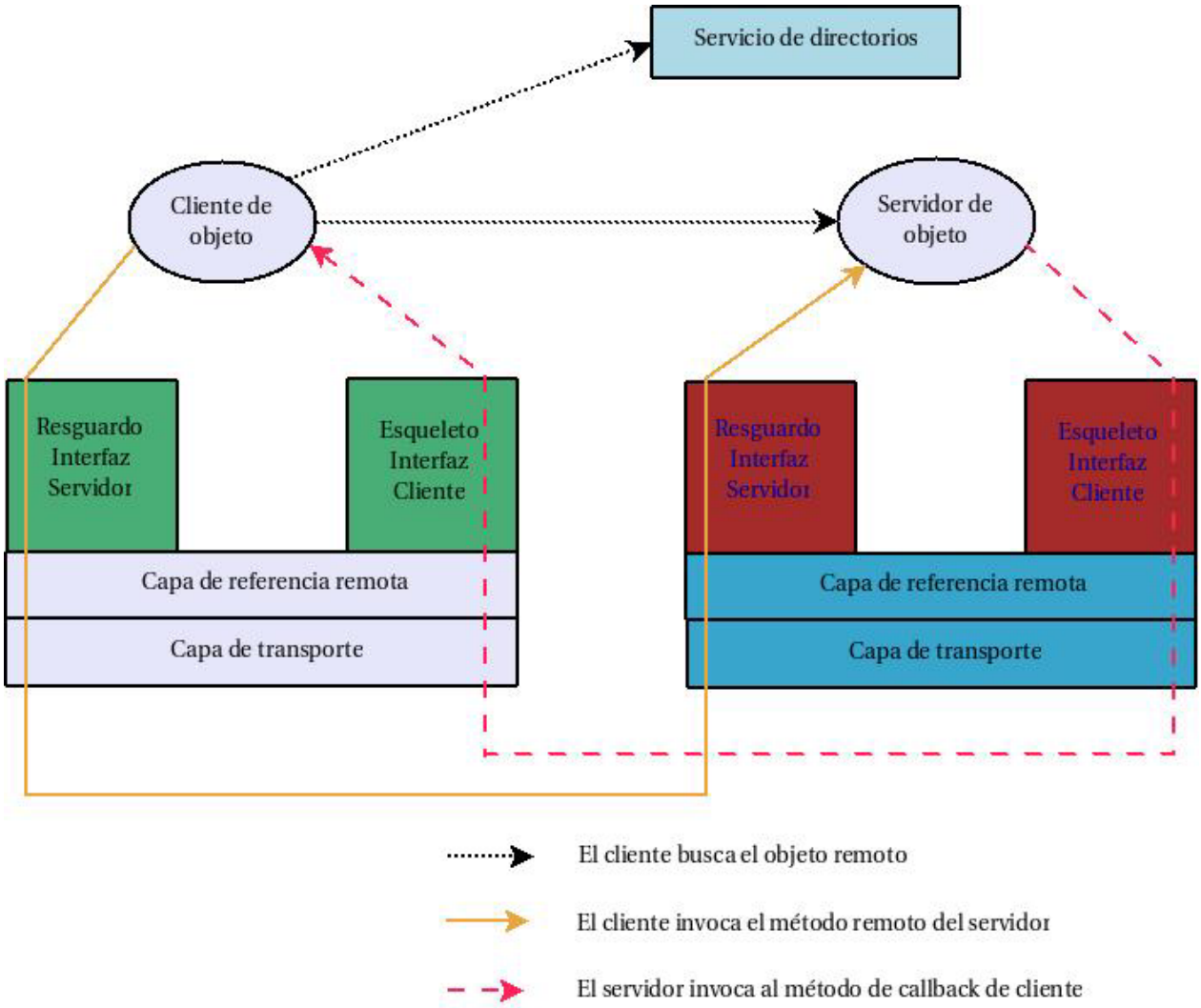


Figura 2. La arquitectura de RMI con callback de cliente.

2. 1. Extensión de la parte cliente para callback de cliente

Para el *callback*, el cliente debe proporcionar un método remoto que permita al servidor notificarle el evento correspondiente. Esto puede hacerse de una forma similar a los métodos remotos del servidor de objeto.

7. 2. 2. La interfaz remota de cliente

Es importante recordar que el servidor de objeto proporciona una interfaz remota que declara los métodos que un cliente de objeto puede invocar. Para el *callback*, es necesario que el cliente de objeto proporcione una interfaz remota similar. Se le denominará interfaz remota de cliente, por oposición a la interfaz remota de servidor. La interfaz remota de cliente debe contener al menos un método que será invocado por el servidor en el *callback*.

```
public interfaz InterfazCallbackCliente
  extends java.rmi.Remote {
  // Este método remoto es invocado por un servidor
  // que realice un callback al cliente que implementa
  // esta interfaz.
  // El parámetro es una cadena de caracteres que
  // contiene información procesada por el cliente
  // una vez realizado el callback.
  // Este método devuelve el mensaje al servidor
  public String notificame(String mensaje)
    throws java.rmi.RemoteException;

} // final de la interfaz
```

El servidor debe invocar el método *notifícame* cuando realiza el *callback*, pasando como argumento una cadena de caracteres (*String*).

2. 3. La implementación de la interfaz remota de cliente

Al igual que la interfaz remota de servidor, es necesario implementar la interfaz remota de cliente en una clase, denominada *ImplCallbackCliente* en el ejemplo.

```
import java.rmi.*;
import java.rmi.server.*;
public class ImplCallbackCliente extends UnicastRemoteObject
   implements InterfazCallbackCliente {
   public ImplCallbackCliente() throws RemoteException {
      super();
   }
   public String notifícame(String mensaje) {
      String mensajeRet =”Callback recibido: “ + mensaje;
      System.out.println(mensajeRet);
      return mensajeRet;
   }
}  // final clase ImplCallbackCliente
```

Al igual que la interfaz remota de servidor, se debe utilizar el compilador ***rmic*** con la implementación de la interfaz remota de cliente para generar los *proxies* necesarios en tiempo de ejecución.

2. 4. Extensión de la clase cliente

En la clase del cliente de objeto, se necesita añadir código al cliente para que instancia un objeto de la implementación de la interfaz remota de cliente. Un ejemplo de cómo debe realizarse esto se muestra a continuación:

```
InterfazCallbackServidor h =
  (InterfazCallbackServidor) Naming.lookup(URLRegistro);
InterfazCallbackCliente objCallback =
  new ImplCallbackCliente();
// registrar el objeto para callback
h.registrarCallback(objCallback);
```

Las Figuras 3 a a 5 presentan el código del software de la parte cliente para la aplicación *HolaMundo* modificada.

Figura 3. Fichero *InterfazCallbackCliente.java* de la aplicación *HolaMundo* modificada.

```
import java.rmi.*;

/**
*  Esto es una interfaz remota para ilustrar el
*  callback de cliente
*/

public interface InterfazCallbackCliente
      extends java.rmi.Remote{
   // Este método remoto se invoca mediante callback
   // de servidor, de forma que realiza un callback a
   // un cliente que implementa esta interfaz.
   // @message una cadena de caracteres que contiene
   // información procesada por el cliente.

   public void notificame(String mensaje)
      throws java.rmi.RemoteException;

}  // fin interfaz
```

Figura 4. Fichero *ImplCallbackCliente.java* de la aplicación *HolaMundo* modificada.

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

/**
*  Esta clase implementa la interfaz remota
*  InterfazCallbackCliente.
*/

public class ImplCallbackCliente extends UnicastRemoteObject implements
      InterfazCallbackCliente {

   public ImplCallbackCliente () throws RemoteException {
     super( );
   }

   public String notificame(String mensaje){
     String mensajeRet = "Callback recibido: " + mensaje;
     System.out.println(mensajeRet);
       return mensajeRet;
   }
}  // fin class ImplCallbackCliente
```

Figura 5. Fichero *ClienteEjemplo.java* de la aplicación *HolaMundo* modificada.

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

/**
*  Esta clase representa el cliente de objeto para un objeto
*  distribuido de la clase ImplCallbackServidor,
*  que implementa la interfaz remota
*  InterfazCallbackServidor. También acepta callbacks
*  del servidor.
*/
```

```
public class ClienteEjemplo {

   public static void main(String args[]) {
     try {
       int puertoRMI;
       String nombreNodo;
       InputStreamReader ent =
          new InputStreamReader(System.in);
       BufferedReader buf= new BufferedReader(ent);
       System.out.prinln(
          "Introduce el nombre de nodo del registro RMI:");
       nombreNodo = buf.readLine();
       System.out.println(
          "Introduce el número de puerto del registro RMI:");
       String numPuerto = buf.readLine();
       puertoRMI = Integer.parseInt(numPuerto);
       System.out.println(
          "Introduce cuantos segundos va a permanecer registrado:");
       String duracionTiempo = buf.readLine();
       int tiempo = Integer.parseInt(duracionTiempo);
       String URLRegistro =
          "rmi://localhost:" + numPuerto + "/callback";
       // Búsqueda del objeto remoto y cast al objeto
       // de la interfaz
       InterfazCallbackServidor h =
          (InterfazCallbackServidor) Naming.lookup(URLRegistro);
       System.out.println("Búsqueda completa ");
       System.out.println("El servidor dice " + h.decirHola());
       InterfazCallbackCliente objCallback =
          new ImplCallbackCliente();
       // registrar para callback
       h.registrarCallback(objCallback);
       System.out.prinln("Registrado para callback.");
       try {
          Thread.sleep(tiempo*1000);
       }
       catch (InterruptedException exc) { //sobre el método sleep
          h.eliminarRegistroCallback(objCallback);
          System.out.println("No registrado para callback.");
       }
       } // fin try
       catch (Exception e) {
          System.out.println(
            "Excepción en ClienteEjemplo: " + e);
       }
   }  // fin main
}  // fin class
```

2. 5. Extensión de la parte servidora para callback de cliente

En la parte del servidor, se necesita añadir un método remoto para que el cliente pueda registrarse para *callback*. En el caso más sencillo, la cabecera del método puede ser análoga a la siguiente:

```
public void registrarCallback(
// Se puede elegir el nombre de método deseado
InterfazCallbackCliente objCallbackCliente
) throws java.rmi.RemoteException;
```

Como argumento se pasa una referencia a un objeto que implementa la interfaz remota de cliente (*InterfazCallbackCliente*, no *ImplCallbackCliente*). También se puede proporcionar un método *eliminarRegistroCallback*, para que

un cliente pueda cancelar el registro. La implementación de estos métodos se muestra en la Figura 7. La Figura 6 muestra el fichero con la interfaz del servidor, que contiene las cabeceras de los métodos adicionales.

Figura 6. Fichero *InterfazCallbackServidor.java* de la aplicación *HolaMundo* modificada.

```
import java.rmi.*;

/**
*  Esto es una interfaz remota para ilustrar el
*  callback de cliente
*/

public interface InterfazCallbackServidor extends Remote{

   public String decirHola( )
     throws java.rmi.RemoteException;

     // Este método remoto permite a un cliente
     // de objeto registrarse para callback
     // @param objClienteCallback es una referencia
     // al cliente de objeto; el servidor lo
     // utiliza para realizar los callbacks

   public void registrarCallback(
        InterfazCallbackCliente objCallbackCliente)
     throws java.rmi.RemoteExcpetion;

     // Este método remoto permite a un cliente
     // de objeto cancelar su registro para callback

   public void eliminarRegistroCallback(
        InterfazCallbackCliente objCallbackCliente)
     throws java.rmi.RemoteExcpetion;
}  // fin interface
```

Figura 7. Fichero *ImplCallbackServidor.java* de la aplicación *HolaMundo* modificada.

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

/**
*  Esta clase implementa la interfaz remota
*  InterfazCallbackServidor
*/

public class ImplCallbackServidor extends UnicastRemoteObject
     implements InterfazCallbackServidor {

   private Vector listaClientes;


   public ImplCallbackServidor () throws RemoteException {
     super( );
     listaClientes = new Vector();
   }

   public String decirHola()
        throws java.rmi.RemoteException {
     return("Hola Mundo");
   }

   public void registrarCallback(
        InterfazCallbackCliente objCallbackCliente)
        throws java.rmi.RemoteException {
```

```
      // almacena el objeto callback en el vector
      if (!(listaClientes.contains(objCallbackCliente))) {
         listaClientes.addElement(objCallbackCliente);
         System.out.println("Nuevo cliente registrado ");
         hacerCallbacks();
      }  // fin if
   }

   // Este método remoto permite a un cliente de objeto
   // cancelar su registro para callback
   // @param id es un identificador para el cliente;
   // el servidor lo utiliza únicamente para identificar al cliente
      registrado.
   public synchronized void eliminarRegistroCallback(
        InterfazCallbackCliente objCallbackCliente)
        throws java.rmi.RemoteException{
      if (listaClientes.removeElement(objCallbackCliente)){
        System.out.println("Cliente no registrado ");
      } else {
        System.out.println(
           "eliminarRegistro: el cliente no fue registrado.")
      }
   }
   private synchronized void hacerCallbacks( ) throws
   java.rmi.RemoteException {
     // realizar callback de un cliente registrado
     System.out.println(
       ************************************\n" +
       "Callback iniciado - ");
     for (int i=0; i<listaClientes.size(); i++) {
       System.out.println("hacienda callback número" + i "\n");
       // convertir el objeto vector a un objeto callback
       InterfazCallbackCliente proxCliente =
          (InterfazCallbackCliente) listaClientes.elementAt(i);
       // invocar el método de callback
       proxCliente.notificame("Número de clientes registrados="
          + listaClientes.size());
     } // fin for
     System.out.println("************************************\n"
       + "Servidor completo callbacks -");
   } // fin hacerCallbacks
}  // fin class ImplCallbackServidor
```

Figura 8. Fichero *ServidorEjemplo.java* de la aplicación *HolaMundo* modificada.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.net.*;
import java.io.*;

/**
*  Esta clase representa el servidor de objeto para un
*  objeto distribuido de la clase Callback, que
*  implementa la interfaz remota InterfazCallback
*/

public class ServidorEjemplo {
   public static void main(String args[]) {
     InputStreamReader ent =
        new InputStreamReader(System.in);
     BufferedReader buf= new BufferedReader(ent);
     String numPuerto, URLRegistro;
     try {
        System.out.println(
        "Introducir el número de puerto del registro RMI:");
        numPuerto=(buf.readLine()).trim();
        int numPuertoRMI = Integer.parseInt(numPuerto);
```

```
            arrancarRegistro(numPuertoRMI);
            ImplCallbackServidor objExportado =
              new ImplCallbackServidor( );
            URLRegistro =
              "rmi://localhost:" + numPuerto + "/callback";
            Naming.rebind(URLRegistro, objExportado);
            System.out.println("Servidor callback preparado.");
          }  // fin try
          catch (Exception exc) {
            System.out.println(
              "Excepción en ServidorEjemplo.main: " + exc);
          } // fin catch
        } // fin main

        // Este método arranca un registro RMI en el nodo
        // local, si no existe en el número de puerto especificado
        private static void arrancarRegistro(int numPuertoRMI)
          throws RemoteException {
          try {
            Registry registro =
            LocalRegistry.getRegistry(numPuertoRMI);
            registro.list();
            // Esta llamada lanza una excepción
            // si el registro no existe
          }
          catch (RemoteException e) {
            // No existe registro válido en el puerto
            Registry registro =
            LocateRegistry.createRegistry(numPuertoRMI);
          }
        } // fin arrancarRegistro
      }  // fin class
```

El servidor necesita emplear una estructura de datos que mantenga una lista de las referencias a la interfaz de cliente registradas para *callbacks*. Cada llamada a *registrarCallback* implica añadir una referencia al vector, mientras que cada llamada a *eliminarRegistroCallback* supone borrar una referencia del vector.

En el ejemplo, el servidor realiza un *callback* siempre que se lleva a cabo una llamada a *registrarCallback*, donde se envía al cliente a través de *callback* el número de clientes actualmente registrados.

Un cliente elimina su registro después de un determinado periodo de tiempo. En las aplicaciones reales, la cancelación del registro se puede realizar al final de la sesión del cliente.

2. 6. Pasos para construir una aplicación RMI con callback de cliente

2. 6. 1. Algoritmo para desarrollar el software de la parte del servidor

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota de servidor en *InterfazCallbackServidor.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.

3. Implementar la interfaz en *ImplCallbackServidor.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilar RMI rmic para procesar la clase de la implementación y generar las ficheros resguardo y esqueleto para el objeto remoto:

   ```
   rmic ImplCallbackServidor
   ```

   Los pasos 3 y 4 deben repetirse cada vez que se cambie la implementación de la interfaz.
5. Obtener una copia del fichero *class* de la interfaz remota del cliente. Alternativamente, obtener una copia del fichero fuente para la interfaz remota y compilarlo utilizando *javac* para generar el fichero *class* de la interfaz *InterfazCallbackCliente.class*.
6. Crear el programa correspondiente al servidor de objeto *ServidorEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
7. Obtener una copia del fichero resguardo de la interfaz remota del cliente.
8. Activar el servidor de objeto

   ```
   java ServidorEjemplo
   ```

2. 6. 2. Algoritmo para desarrollar el software de la parte cliente

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota de cliente en *InterfazCallbackCliente.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
3. Implementar la interfaz en *ImplCallbackCliente.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilador RMI rmic para procesar la clase de la implementación *ImplCallbackCliente.class* y generar los fichero resguardo y esqueleto *ImplCallbackCliente_Skel.class* y *ImplCallbackCliente_Stub.class* para el objetivo remoto:

   ```
   rmic ImplCallbackCliente
   ```

   Los pasos 3 y 4 deben repetirse cada vez que se cambie la implementación de la interfaz.
5. Obtener una copia del fichero *class* de la interfaz remota del servidor. Alternativamente, obtener una copia del fichero fuente para la interfaz remota y compilarlo utilizando *javac* para generar el fichero *class* de la interfaz.

6. Crear el programa correspondiente al cliente de objeto *ClienteEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
7. Obtener una copia del fichero resguardo de la interfaz remota del servidor *ImplCallbackServidor_Stub.class*. Activar el cliente de objeto

```
java ClienteEjemplo
```

La Figura 9 muestra los ficheros que se necesitan en los dos extremos, cliente y servidor, cuando se utilizan *callback* de cliente.

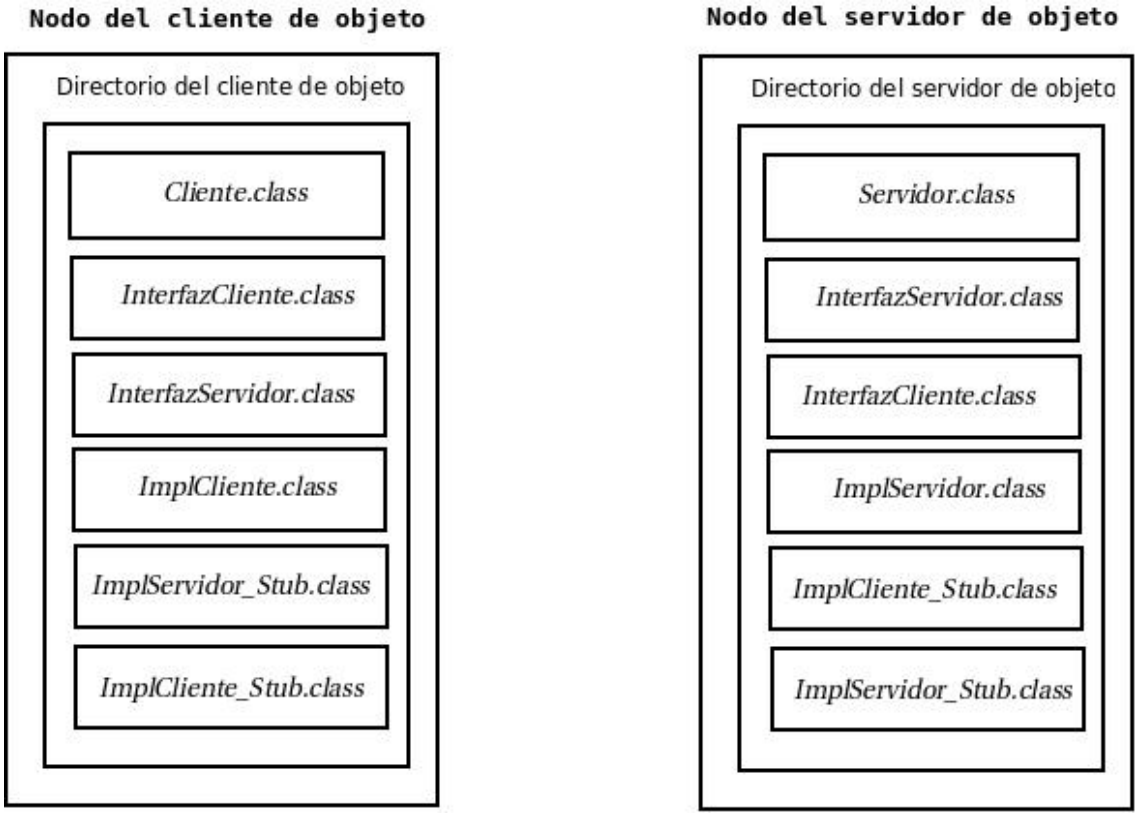


Figura 9. Colocación de los ficheros en una aplicación RMI con callback de cliente.

3. Descarga de resguardo

En la arquitectura de un sistema de objetos distribuidos se requiere un Proxy para interactuar con la llamada a un método remoto de un cliente de objeto. En Java RMI, este Proxy o intermediario es el resguardo de la interfaz remota del Servidor. La clase resguardo generada debe estar en la máquina cliente en tiempo de ejecución cuando un programa cliente se ejecute.

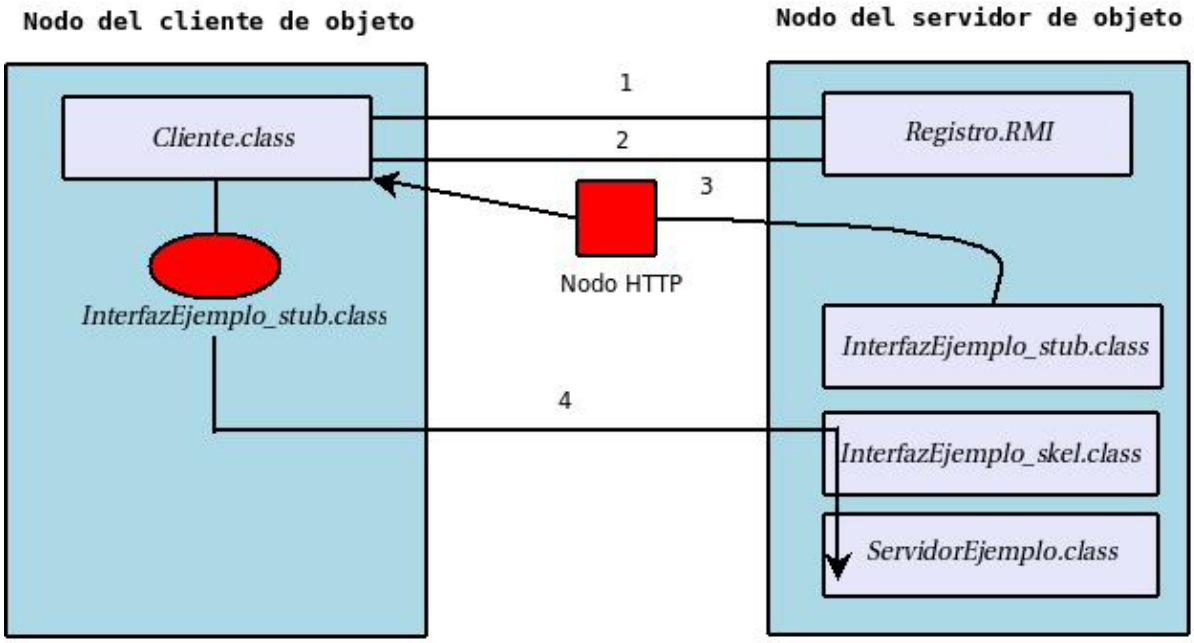
Java RMI proporciona un mecanismo que permite que los clientes obtengan dinámicamente los resguardos necesarios. Mediante descarga dinámica

de resguardo, no se necesita una copia de la clase del resguardo en máquina cliente.

Mediante el uso de descarga dinámica, el desarrollador almacena una clase resguardo en un servidor web como un documento web, que puede ser descargado (mediante http) cuando un cliente de objeto se ejecuta, de la misma forma que se lleva a cabo la descarga de applets.

Al igual que antes, un servidor explota un objeto contactando con el registro RMI y registrando una referencia remota al objeto, especificando un nombre simbólico para la referencia. Si se desea utilizar descarga de resguardo, el servidor debe también indicar al registro el URL donde se encuentra almacenada la clase resguardo.

Un cliente que desee invocar un método remoto de un objeto exportado contacta con el registro RMI en la máquina servidora para traer la referencia remota a través del nombre. Sin descarga de resguardo, el objeto resguardo (un fichero *class* Java) debe colocarse en la máquina cliente manualmente y la máquina virtual Java debe poder localizarlo. Si se utiliza descarga de resguardo entonces se puede obtener dinámicamente la clase resguardo de un servidor http de forma que puede interactuar con el cliente de objeto y el soporte en tiempo real de RMI. La clase resguardo descargada no es persistente, es decir, no se almacena de forma permanente en la máquina cliente, sino que por el contrario el sistema libera la clase correspondiente cuando la sesión del cliente finaliza.



1. El cliente busca el objetivo de la interfaz en el registro RMI del nodo servidor.

2. El registro RMI devuelve una referencia remota al objeto de la interfaz.

3. Si el resguardo del objeto de la interfaz no está en el nodo cliente y el servidor está configurado para ello, se descarga el resguardo de un servidor HTTP.

4. A través del resguardo del servidor, el proceso cliente interactúa con el esqueleto del objeto de la interfaz para acceder a los métodos del objeto servidor.

Figura 10. Descarga de resguardo.

4. El gestor de seguridad de RMI

Aunque la descarga de resguardo es una característica útil, su uso supone un problema para el sistema de seguridad. Este problema no está asociado al uso de RMI, sino a la descarga de objetos en general. Cuando un objeto como un resguardo RMI se transfiere desde un nodo remoto, su ejecución entraña el riesgo de ataques maliciosos al nodo local.

Para evitar los problemas de seguridad del uso de descarga de resguardo, Java proporciona una clase denominada *RMISecurityManager*. Un programa RMI puede instanciar un objeto de esta clase. Una vez instanciando, el objeto supervisa durante la ejecución del programa todas las acciones que puedan suponer un riesgo de seguridad. Estas acciones incluyen el acceso a ficheros locales y la realización de conexiones de red, ya que dichas acciones podrían suponer modificaciones de los recursos locales no deseadas o mal uso de los recursos en red. El soporte en tiempo de exportar cualquier objeto que requiera

descarga de resguardo, y que un cliente instale un **gestor de seguridad** antes de que pueda realizar la descarga del resguardo.

Por defecto, un gestor de seguridad RMI es muy restrictivo: no permite acceso a los ficheros y sólo permite conexiones a la máquina origen. Esta restricción no permite a un cliente de objeto RMI contactar con el **registro RMI** del servidor de objeto y tampoco le permite llevar a cabo descarga de resguardo. Es posible relajar estas condiciones instalando un fichero especial conocido como **fichero de política de seguridad** cuya sintaxis especifíca el tipo de restricción que un gestor de seguridad, incluyendo los gestores de seguridad RMI , debe utilizar.

A continuación, se describe cómo una aplicación utiliza el gestor de seguridad de RMI.

3. 1. Instanciación de un gestor de seguridad en un programa RMI

La clase *RMISecurityManager* se puede instanciar tanto en el cliente de objeto como en el servidor de objeto utilizando la siguiente sentencia:

```
System.setSecurityManager(new RMISecurityManager());
```

Esta sentencia debería aparecer antes del código de acceso al registro RMI.

Figura 11. HolaMundoServidor.java haciendo uso de un gestor de seguridad.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.net.*;
import java.io.*;

/*
*  Esta clase representa el servidor de objeto para un
*  objeto distribuido de la clase HolaMundo, que
*  implementa la interfaz remota InterfazHolaMundo. Se
*  instala un gestor de seguridad para realizar descarga de resguardo
*/

public class HolaMundoServidor {
   public static void main(String args[]) {
      InputStreamReader ent =
         New InputStreamReader(System.in);
      BufferedReader buf= new BufferedReader(ent);
      String numPuerto, URLRegistro;
      try {
        System.out.println(
        "Introducir el numero de puerto del registro RMI:");
        numPuerto=(buf.readLina()).trim();
        int numPuertoRMI = Integer.parseInt(numPuerto);

        // arrancar un gestor de seguridad - esto es
        // necesario si se utiliza descarga de resguardo
        System.setSecurityManager(
          new RMISecurityManager());
```

```
        arrancarRegistro(numPuertoRMI);
        ImplHolaMundo objExportado = new ImplHolaMundo();
        URLRegistro =
        "rmi://localhost:" + numPuerto + "/holaMundo";
        Naming.rebind(URLRegistro, objExportado);
        System.out.println(
        "Servidor registrado. El registro contiene:");
        // listar los nombres registrado actualmente
        listarregistro(URLRegistro);
        System.out.println("Servidor Hola Mundo preparado");
      } // fin try
      catch (Exception exc) {
        System.out.println(
        "Excepción en HolaMundoServidor.main: " + exc);
      } // fin catch
   } // fin main

   // Este método arranca un registro RMI en el nodo
   // local, si no existe en el número de puerto especificado.
   private static void arrancarRegistro(int numPuertoRMI)
      throws RemoteException {
      try {
         Registry registro =
         LocateRegistry.getRegistry(numPuertoRMI);
         Registro.list();  // Esta llamada lanza
         // una excepción si el registro no existe
      }
      catch (RemoteException e) {
        // No existe registro válido en el puerto
        System.out.println(
        "El registro RMI no se localiza en este puerto "
        + numPuertoRMI);
        Registry registro =
        LocateRegistry.createRegistry(numPuertoRMI);
        System.out.println(
        "Registro RMI creado en el puerto "+ numPuertoRMI);
      } // fin catch
   }  // fin arrancarRegistro

   // Este método lista los nombres registrado en RMI
   private static void listarRegistro(String URLRegistro)
     throws RemoteException, MalformedURLException {
     System.out.println(
     "Registro " + URLRegistro + " contiene: ");
     String [] nombres = Naming.list(URLRegistro);
     for (int i=0; i< nombres.length; i++)
        System.out.println(nombres[i]);
   } // fin listarRegistro

}  // fin class
```

Figura 12. HolaMundoCliente.java haciendo uso de un gestor de seguridad.

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

/**
*  Esta clase representa el cliente de objeto para un
*  objeto distribuido de la clase HolaMundo, que
*  implementa la interfaz remota InterfazHolaMundo.
* Se instala un gestor de seguridad para realizar descarga de resguardo segura.
*/

public class HolaMundoCliente {

   public static void main(String args[]) {
      try {
         int puertoRMI;
         String nombreNodo;
         InputStreamReader ent =
```

```
            new InputStreamReader (System.in);
          BufferedReader buf= new BufferedReader(ent);
          System.out.println(
          "Introduce el nombre de nodo del registro RMI:");
          nombreNodo = buf.readLine();
          System.out.prinln(
          "Introduce el número de puerto del registro RMI:");
          String numPuerto = buf.readLine();
          puertoRMI = Integer.parseInt(numPuerto);

          // arrancar un gestor de seguridad - esto es
          // necesario si se utiliza descarga de resguardo
          System.setSecurityManager(new RMISecurityManager());

          String URLRegistro =
          "rmi://localhost:" + numPuerto + "/holaMundo";
          // Búsqueda del objeto remoto y cast al
          // objeto de la interfaz
          InterfazHolaMundo h =
          (InterfazHolaMundo)Naming.lookup(URLRegistro);
          System.out.println("Búsqueda completa ");
          // invocar el método remoto
          String mensaje=h.decirHola();
          System.out.println("HolaMundoCliente: " + mensaje);
        }  // fin try
        catch (Exception e) {
            System.out.println(
            "Excepción en HolaMundoCliente: " + e);
        } // fin catch
      }  // fin main
    }  // fin class
```

3. 2. La sintaxis de un fichero de políticas de seguridad de Java

Un fichero de políticas de seguridad de Java es un fichero de texto que contiene códigos que permiten especificar la concesión de permisos específicos. A continuación se muestra un fichero típico java.policy para una aplicación RMI.

```
grant {
   // Este permiso permite a los clientes RMI realizar
   // conexiones de sockets a los puertos públicos de
   // cualquier computador.
   // Si se arranca un puerto en el registro RMI en este
   // rango, no existirá una violación de acceso de
   // conexión.
   // permission java.net.SocketPermission "*:1024-65535",
   //  "connect,accept,resolve";
   // Este permiso permite a los sockets acceder al Puerto
   // 80, el puerto por defecto http que el cliente
   // necesita para contactar con el servidor http para
   // descarga de resguardo
   permission java.net.SocketPermission "*:80", "connect";
};
```

Cuanto se active el cliente, hay que utilizar la opción del mandato que permite especificar que el proceso cliente debe tener los permisos definidos en el fichero de políticas, de la siguiente forma:

```
java -Djava.security.policy=java.policy ClienteEjemplo
```

Del mismo modo, el servidor debe activarse del siguiente modo:

```
java –Djava.security.policy=java.policy ServidorEjemplo
```

Estos dos mandatos asumen que el fichero de políticas se llama *java.policy* y está disponible en el directorio actual de la parte servidora y cliente.

3. 3. Uso de descarga de resguardo y un fichero de políticas de seguridad

1. Si debe descargarse el resguardo de un servidor HTTP, transfiera la clase resguardo a un directorio apropiado del servidor http.
2. Cuando se activa el servidor, se debe especificar las siguientes opciones del mandato:

```
java –Djava.rmi.server.codbase=<URL>
      -Djava.security.policy=
      <ruta completa del fichero de políticas de seguridad>
```

donde:

<URL> es el URL del directorio donde se encuentra la clase resguardo.

<ruta completa del fichero de políticas de seguridad> especifica el fichero de políticas de seguridad de la aplicación.

La Figura 13 muestra el conjunto de ficheros que se necesitan para una aplicación RMI y dónde se deben colocar, suponiendo descarga dinámica de resguardo.

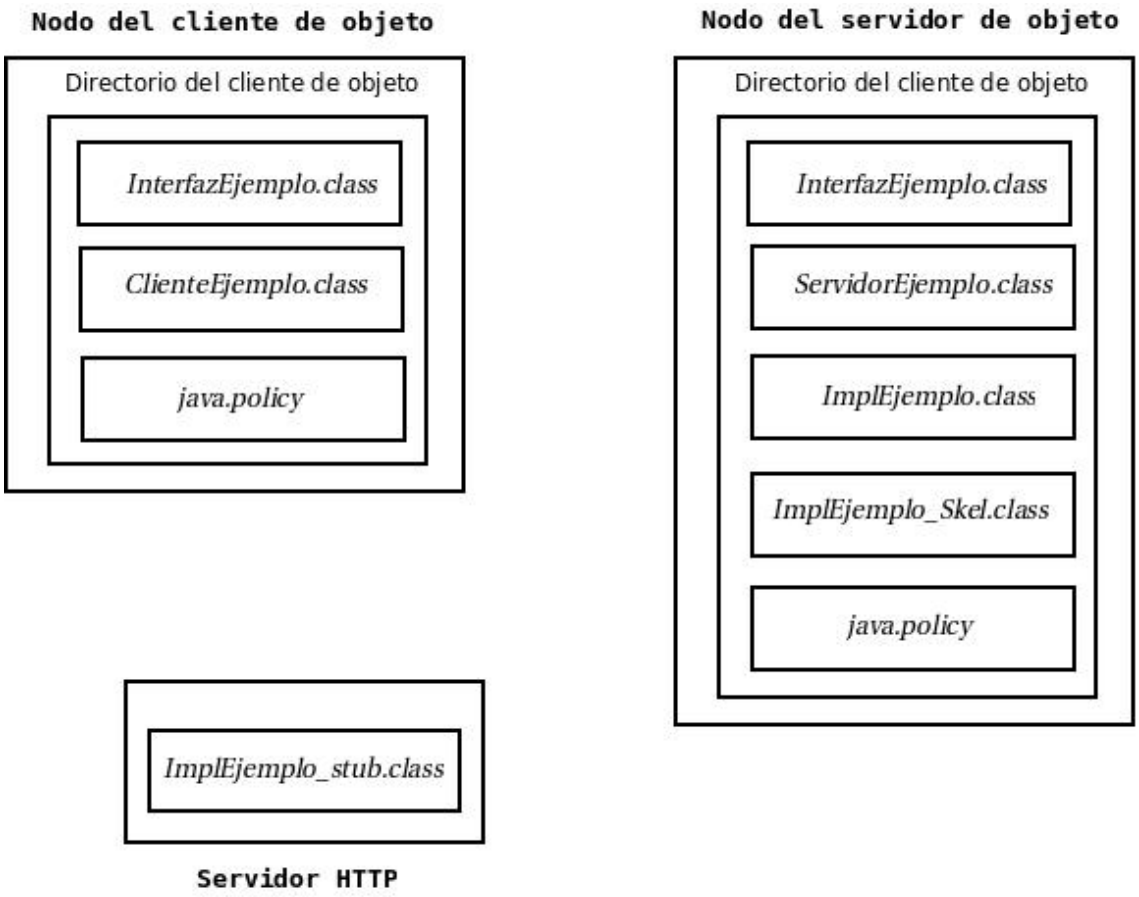


Figura 13. Colocación de los ficheros RMI en una aplicación que utiliza descarga de resguardo.

En la parte del servidor, los ficheros necesarios son los ficheros *class* del servidor, la interfaz remota, la implementación de la interfaz (generada por *javac*), el fichero *class* del resguardo (generado por *rmic*), la clase del esqueleto (generado por *rmic*), y el fichero de políticas de seguridad de la aplicación. En la parte cliente, los ficheros que se necesitan son el fichero *class* del cliente, el fichero *class* de la interfaz remota del servidor y el fichero de políticas de seguridad de la aplicación. Finalmente, el fichero *class* del resguardo se debe almacenar en el nodo HTTP del cual se descarga el resguardo.

3. 4. Algoritmos para construir una aplicación RMI, que permita descarga de resguardo

A continuación se realiza una descripción del procedimiento paso a paso para la construcción de una aplicación RMI, teniendo en cuenta el uso de descarga de resguardo.

3. 4. 1. Algoritmo para desarrollar el software de la parte del servidor

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota de servidor en *IntefazEjemplo.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
3. Implementar la interfaz en *ImplEjemplo.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilador RMI *rmic* para procesar la clase de la implementación y generar los ficheros resguardo y esqueleto para el objeto remoto:

```
rmic ImplEjemplo
```

   Los pasos 3 y 4 deben repetirse cada vez que se cambie la implementación de la interfaz.
5. Crear el programa del servidor de objeto *ServidorEjmplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
6. Si se desea descarga de resguardo, copiar el fichero resguardo en un directorio apropiado del servidor HTTP.
7. Si se utiliza el registro RMI y no ha sido ya activado, activarlo.
8. Construir un fichero de políticas de seguridad para la aplicación denominado *java.policy*, y colocarlo en un directorio apropiado o en el directorio actual.
9. Activar el servidor, especificando el campo *codebase* si se utiliza descarga de resguardo, y el fichero de políticas de seguridad.

```
java –Djava.rmi.server.codebase=http://nodo.dom.edu/resguardos/
-djava.security.policy=java.policy ServidorEjemplo
```

   Este mandato se ejecuta en una única línea, aunque se puede utilizar un carácter de continuación de línea ('\') en un sistema UNIX.

3. 4. 2. Algoritmo para desarrollar el software de la parte cliente

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Obtener una copia del fichero *class* de la interfaz remota del servidor.
3. Crear el programa cliente *ClienteEjemplo.java* y compilarlo para generar la clase cliente.
4. Si se desea utilizar descarga de resguardo, obtener una copia del fichero *class* del resguardo y colocarlo en el directorio actual.
5. Construir un fichero de políticas de seguridad para la aplicación denominado *java.policy*, y colocarlo en un directorio apropiado o en el directorio actual.

6. Activar el cliente, especificando el fichero de políticas de seguridad.

   ```
   java –Djava.security.policy=java.policy ClienteEjemplo
   ```

   Este mandato se ejecuta en una única línea, aunque se puede utilizar un carácter de continuación de línea (‘\’) en un sistema UNIX.