Lenguajes y *Middlewares* para Programación Distribuida

1 Java y RMI

1.1 Introducción

RMI (*Remote Method Invocation*) es una de las dos formas que Java propone para trabajar con objetos distribuidos. La otra alternativa es mediante la interfaz Java-IDL. IDL (*Interface Definition Language*) se ha diseñado para comunicar objetos Java y objetos creados en otros lenguajes (e.g. C++) mediante *Object Request Brokers* (normalmente, aquellos que satisfacen el estándar CORBA).

Suponga que desea coleccionar información localmente en un cliente y enviarla a través de la red a un servidor. Por ejemplo, un usuario rellena un formulario de petición de información, el cual se envía al vendedor, y este devuelve la información del producto solicitado. Para hacer esto, existen varios métodos:

1) **Conexiones con *sockets*** para enviar flujos de bytes entre el cliente y el servidor. Este método es útil si sólo es necesario enviar datos en bruto a través de la red.
2) **JDBC** para realizar consultas y actualizaciones de bases de datos. Es útil cuando la información que se envía encaja en el modelo de tablas de una base de datos relacional.
3) **RMI** permite implementar el formulario de petición y la información del producto como objetos remotos. Utilizando este método, los objetos pueden ser transportados entre el cliente y el servidor.

RMI proporciona los siguientes beneficios:

- El programador no tiene que manejar flujos de bytes.
- Se pueden utilizar objetos de cualquier tipo.
- Se pueden invocar llamadas a métodos en objetos situados en otra computadora sin tener que desplazarlos a la computadora que realiza la invocación.

Al igual que con RPC, la terminología cliente-servidor sólo se aplica a la llamada de un único método. Las funciones de los procesos se pueden invertir, y así el servidor de una llamada previa puede actuar como cliente en la siguiente llamada.

1.2 Invocaciones Remotas de Métodos

1.2.1 Cómo RMI amplía las llamadas locales

Los clientes RMI interaccionan con objetos remotos a través de interfaces. Nunca lo hacen directamente con las clases que implementan estas interfaces.

A diferencia de llamadas locales de Java, una invocación remota pasa los objetos locales en los parámetros por copia, en lugar de por referencia. Por otro lado, ya que se puede acceder a objetos remotos en una red, RMI pasa un objeto remoto por referencia, no copiando la implementación remota real.

RMI proporciona nuevas interfaces y clases que permiten encontrar objetos remotos, cargarlos y después ejecutarlos de forma segura. También incluye un **servicio de denominación** (ligadura) básico y no permanente que permite localizar objetos remotos. Asimismo, RMI proporciona mecanismos de seguridad extra para asegurar un buen comportamiento de estos *stubs*.

Dado que las invocaciones remotas de métodos pueden fallar por muchas más razones que las locales, se han de manejar excepciones adicionales. Por ello, RMI amplía las clases de excepciones de Java para tratar con cuestiones remotas.

Un buen recolector de objetos no utilizados tiene que ser capaz de borrar automáticamente objetos remotos que no se referencian por ningún cliente. RMI utiliza un esquema de recolección basado en contadores de referencias que mantiene la pista de todas las referencias externas existentes a objetos remotos. En este caso, una referencia es justo una conexión cliente-servidor sobre una sesión TCP/IP.

1.2.2 Stubs

Cuando un cliente desea invocar un método en un objeto remoto, realmente llama a un método normal de Java que hay encapsulado en un objeto sustituto llamado *stub*. Java utiliza un mecanismo de serialización de objetos para formar los parámetros en el objeto *stub*. El método que se invoca en el *stub* del cliente construye un bloque de información que consiste en:

- Un identificador del objeto remoto a utilizar.
- Un número de operación que identifica al método que se quiere utilizar.
- Los parámetros formados.

La sintaxis para una llamada remota es la misma que para una llamada local:

```
// cliente.java = Programa cliente
...
      // Llama a la función remota escribir_mensaje()
      instancia_local.escribir_mensaje(Integer.parseInt(args[1]));

...
```

El cliente siempre utiliza variables cuyo tipo es una interfaz para acceder a los objetos remotos. Por ejemplo, asociado a la llamada anterior estaría la interfaz:

```
// Ejemplo_I.java = Interfaz para contador
public interface Ejemplo_I extends Remote {
  public void escribir_mensaje (int id_proceso) throws RemoteException;
}
```

Y la declaración de un objeto para una variable que implementa una interfaz:

```
// cliente.java = Programa cliente
...
// Crea el stub para el cliente
Ejemplo_I stub =
                (Ejemplo_I) UnicastRemoteObject.exportObject(prueba, 0);
...
```

Por supuesto, las interfaces son entidades abstractas que sólo anuncian que métodos pueden ser llamados junto con sus *signatures*. Aquellas variables cuyo tipo sea una interfaz siempre tienen que estar ligadas a un objeto real de algún tipo; en el caso de objetos remotos, son instancias de una clase *stub*. El programa cliente no conoce realmente el tipo de los objetos remotos. Las clases *stubs* y los objetos asociados se crean automáticamente.

1.2.3 Proceso de Desarrollo RMI

A continuación, se muestran y describen los pasos a seguir para crear las clases RMI (cliente y servidor) y ejecutarlas:

1) **Definir la interfaz remota**. El objeto remoto tiene que declarar sus servicios por medio de una interfaz remota. Esto se hace derivando de la interfaz *java.rmi.Remote*. Cada método en una interfaz remota tiene que lanzar una excepción *java.rmi.RemoteException*.
2) **Implementar la interfaz remota**. Se tiene que proporcionar una clase para el objeto remoto que implemente la interfaz anterior.
3) **Compilar la clase del objeto remoto**. Se realiza con *javac*.
4) **Implementar el gestor de seguridad**. Si nuestra interfaz contiene métodos que utilizan argumentos o devuelven resultados en objetos de clases distintas de la API de Java, es necesario implementar un gestor de seguridad para evitar intrusiones de código maligno al invocar el objeto remoto con argumentos dañinos. Para activar el gestor de seguridad utilizamos:

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

   Podemos modificar la política de seguridad de Java con un fichero de permisos del tipo:

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

   Y especificar el fichero al ejecutar el cliente y/o servidor con la opcion:

```
-Djava.security.policy=grantFilePath
```

5) **Lanzar el ligador RMI en el servidor**. RMI define interfaces para un servicio de denominación (ligadura) no permanente llamado *rmiregistry* que permite registrar y localizar objetos remotos utilizando nombres simples. Cada proceso servidor puede tener un ligador propio, o puede haber un único ligador para todos los programas Java que se estén ejecutando en un mismo computador. Por defecto, el ligador tiene asociado el puerto 1099. Si se desea otro, habrá que especificarlo como parámetro en el momento de lanzarlo. El ligador se puede lanzar desde una aplicación Java con *Runtime.getRuntime().exec("rmiregistry")* o con *LocateRegistry.createRegistry (int port).* También puede lanzarse por consola:

```
rmiregistry 1311 &
```

   Un ligador no puede ejecutarse si ya hay otro corriendo en el mismo puerto. Por tanto, sería necesario matar el proceso anterior buscando el número de PID (*ps –ef | grep rmiregistry*).

6) **Lanzar el objeto remoto**. Se tienen que cargar las clases del objeto remoto y después crear las instancias que se deseen. Cuando se desarrolle una aplicación en la cual puedan existir invocaciones remotas a métodos de diferentes objetos que pueden o tienen que formar parte de un único proceso servidor, entonces el diseño a seguir para la aplicación será similar al del ejemplo mostrado más adelante. En éste, el objeto remoto se lanza instanciándolo dentro de un proceso servidor, el cual es un programa normal Java.
7) **Registrar el objeto remoto en el ligador**. Se tienen que registrar todas las instancias de objetos remotos por medio de nombres, que utilizarán los clientes para localizarlos. Para ello, se puede utilizar *java.rmi.Naming* mediante *Naming.rebind(nombre, stub),* o se puede utilizar *java.rmi.registry.Registry* mediante los métodos de la clase *java.rmi.Registry.rebind(nombre, stub)*. Esta clase utiliza el ligador para almacenar los nombres. Una vez ejecutado este paso, el objeto remoto ya está preparado para ser invocado por los clientes.
8) **Escribir el código del cliente**. Normalmente, la implementación del cliente será código normal Java, en el cual hay que utilizar la clase del paso anterior para localizar el objeto remoto.
9) **Compilar el código cliente**. Se utiliza *javac*.
10) **Lanzar el cliente**. Se tienen que cargar las clases del cliente y sus *stubs*.

La siguiente macro se puede utilizar para compilar y ejecutar en una única máquina el ejemplo que se verá más adelante, suponiendo que todos los archivos *.java y el archivo con las políticas de seguridad están en el mismo directorio que la macro.

Nota: la ruta en la que suelen estar los compiladores Java en Linux es /usr/lib/jvm/versión-específica-de-java

```
#!/bin/sh -e
# ejecutar = Macro para compilación y ejecución del programa ejemplo
# en una sola maquina Unix de nombre localhost.

echo
echo "Lanzando el ligador de RMI … "
rmiregistry &

echo
echo "Compilando con javac ..."
javac *.java

sleep 2

echo
echo "Lanzando el servidor"
```

```
java -cp . -Djava.rmi.server.codebase=file:./ -Djava.rmi.server.hostname=localhost -
Djava.security.policy=server.policy Ejemplo &

sleep 2

echo
echo "Lanzando el primer cliente"
echo
java -cp . -Djava.security.policy=server.policy Cliente_Ejemplo localhost 0

sleep 2

echo
echo "Lanzando el segundo cliente"
echo
java -cp . -Djava.security.policy=server.policy Cliente_Ejemplo localhost 3
```

1.2.4 Ejemplo 1

El programa que se muestra a continuación es una pequeña aplicación cliente-servidor. En este ejemplo, el servidor (*Ejemplo.java*) exporta los métodos contenidos en la interfaz *Ejemplo_I.java* del objeto remoto instanciado como *Ejemplo_I* de la clase definida en *Ejemplo.java*.

El programa servidor, cuando recibe una petición de un cliente, imprime el argumento enviado en la llamada. En caso de ser un "0", el argumento recibido espera un tiempo antes de volver a imprimir el mensaje.

El programa cliente es un programa normal Java que realiza las siguientes acciones:

1) Activa el gestor de seguridad
2) Busca el objeto remoto
3) Invoca el método escribir_mensaje con el argumento pasado por consola

```
# Fichero: server.policy


grant codeBase "file:./" {
    permission java.security.AllPermission;
};
```

```
# Fichero: Ejemplo_I.java
# Define la Interfaz remota


import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Ejemplo_I extends Remote {
  public void escribir_mensaje (int id_proceso) throws RemoteException;
}



# Fichero: Ejemplo.java
# Implementa la Interfaz remota


import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.lang.Thread;


public class Ejemplo implements Ejemplo_I {

    public Ejemplo() {
        super();
    }

    public void escribir_mensaje (int id_proceso) {
      System.out.println("Recibida peticion de proceso: "+id_proceso);
      if (id_proceso == 0) {
        try{
          System.out.println("Empezamos a dormir");
          Thread.sleep(5000);
          System.out.println("Terminamos de dormir");
        }
        catch (Exception e) {
          System.err.println("Ejemplo exception:");
          e.printStackTrace();
        }
      }
      System.out.println("\nHebra "+id_proceso);
    }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String nombre_objeto_remoto = "Ejemplo_I";
            Ejemplo_I prueba = new Ejemplo();
            Ejemplo_I stub =
                (Ejemplo_I) UnicastRemoteObject.exportObject(prueba, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(nombre_objeto_remoto, stub);
            System.out.println("Ejemplo bound");
        } catch (Exception e) {
            System.err.println("Ejemplo exception:");
            e.printStackTrace();
        }
    }
}



# Fichero: Cliente_Ejemplo.java
# Código del cliente


import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Cliente_Ejemplo {
    public static void main(String args[]) {
```

```
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String nombre_objeto_remoto = "Ejemplo_I";
            System.out.println("Buscando el objeto remoto");
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Ejemplo_I instancia_local = (Ejemplo_I) registry.lookup(nombre_objeto_remoto);
            System.out.println("Invocando el objeto remoto");
            instancia_local.escribir_mensaje(Integer.parseInt(args[1]));
        } catch (Exception e) {
            System.err.println("Ejemplo_I exception:");
            e.printStackTrace();
        }
    }
}
```

1.2.5 Ejemplo 2: Multihebra

Se trata de un ejemplo similar al anterior pero, en lugar de lanzar varios clientes, creamos múltiples hebras que realizan la misma tarea de imprimir un mensaje remoto accediendo al stub de un objeto remoto. Este ejemplo nos permite ver la gestión de la concurrencia en RMI. En esta ocasión, en vez de pasar al objeto remoto un número entero, pasamos una cadena String.

RMI es **multihebrado**, lo que habilita la gestión concurrente de las peticiones de los clientes. Esto también implica que las implementaciones de los objetos remotos han de ser seguras (*thread-safe*).

```
# Fichero: server.policy


grant codeBase "file:./" {
    permission java.security.AllPermission;
};



# Fichero: Ejemplo_I.java
# Define la Interfaz remota


import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Ejemplo_I extends Remote {
  public void escribir_mensaje (String mensaje) throws RemoteException;
}



# Fichero: Ejemplo.java
# Implementa la Interfaz remota


import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
```

```
public class Ejemplo implements Ejemplo_I {

    public Ejemplo() {
        super();
    }

    public void escribir_mensaje (String mensaje) {
      System.out.println("\nEntra Hebra "+mensaje);

      //Buscamos los procesos 0, 10, 20,...
      if (mensaje.endsWith("0")) {
        try{
          System.out.println("Empezamos a dormir");
          Thread.sleep(5000);
          System.out.println("Terminamos de dormir");
        }
        catch (Exception e) {
          System.err.println("Ejemplo exception:");
          e.printStackTrace();
        }
      }
      System.out.println("Sale Hebra "+mensaje);
    }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String nombre_objeto_remoto = "Ejemplo_I";
            Ejemplo_I prueba = new Ejemplo();
            Ejemplo_I stub =
                (Ejemplo_I) UnicastRemoteObject.exportObject(prueba, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(nombre_objeto_remoto, stub);
            System.out.println("Ejemplo bound");
        } catch (Exception e) {
            System.err.println("Ejemplo exception:");
            e.printStackTrace();
        }
    }
}



# Fichero: Cliente_Ejemplo_Multi_Threaded.java
# Código del cliente


import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;


public class Cliente_Ejemplo_Multi_Threaded
                    implements Runnable {

  public String nombre_objeto_remoto = "Ejemplo_I";
  public String server;

  public Cliente_Ejemplo_Multi_Threaded (String server) {
    //Almacenamos el nombre del host servidor
    this.server = server;
  }

  public void run() {
    System.out.println("Buscando el objeto remoto");
    try {
      Registry registry = LocateRegistry.getRegistry(server);
      Ejemplo_I instancia_local = (Ejemplo_I) registry.lookup(nombre_objeto_remoto);
      System.out.println("Invocando el objeto remoto");
      instancia_local.escribir_mensaje(Thread.currentThread().getName());
    } catch (Exception e) {
      System.err.println("Ejemplo_I exception:");
      e.printStackTrace();
    }
  }
```

```
  public static void main(String args[]) {

    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    int n_hebras = Integer.parseInt(args[1]);

    Cliente_Ejemplo_Multi_Threaded[] v_clientes = new Cliente_Ejemplo_Multi_Threaded[n_hebras];
    Thread[] v_hebras = new Thread[n_hebras];
    for (int i=0;i<n_hebras;i++) {
      //A cada hebra le pasamos el nombre el servidor
      v_clientes [i] = new Cliente_Ejemplo_Multi_Threaded(args[0]);
      v_hebras[i] = new Thread(v_clientes[i],"Cliente "+i);
      v_hebras[i].start();
    }
  }
}
```

En este caso, al lanzar el cliente, el primer parámetro será el nombre del host del servidor y el segundo, el número de hebras que queremos crear. Compruebe qué ocurre con las hebras cuyo nombre acaba en 0 y en qué se diferencian del resto. Preste atención a la impresión de los mensajes y compruebe si estos se entrelazan.

Ahora pruebe a introducir el modificador **synchronized** en el método de la implementación remota: *public **synchronized** void escribir_mensaje (String mensaje) {…}* y trate de entender las diferencias en la ejecución de los programas.

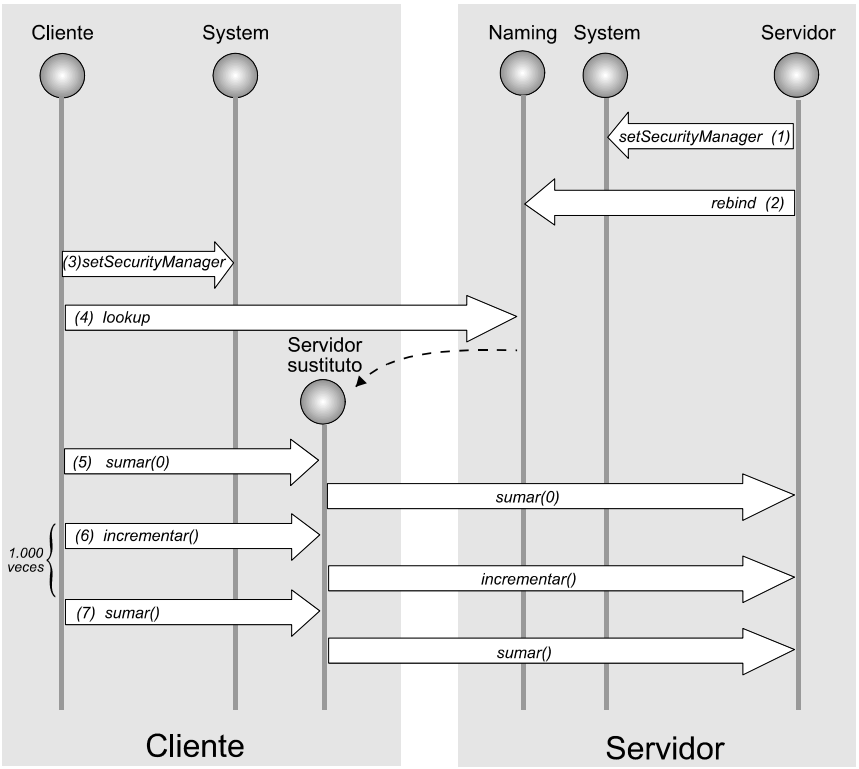
1.2.6 Ejemplo 3

En este ejemplo de contador se crea por una parte el objeto remoto y por otra el servidor. El objeto remoto consta de varias funciones accesibles remotamente. El servidor (*servidor.java*) exporta los métodos contenidos en la interfaz *icontador.java* del objeto remoto instanciado como *micontador* de la clase definida en *contador.java*.

El programa cliente es un aplicación normal de Java que realiza las siguientes acciones:

4) Pone un valor inicial en el contador del servidor.
5) Invoca el método *incrementar* del contador 1.000 veces.
6) Imprime el valor final del contador junto con el tiempo de respuesta medio que se ha calculado a partir de las invocaciones remotas del método *incrementar*.

La siguiente figura muestra las interacciones RMI entre cliente y servidor:



A continuación se da un listado del código fuente del programa completo:

```
# Fichero: server.policy


grant codeBase "file:./" {
    permission java.security.AllPermission;
};



# Fichero: icontador.java
# Define la Interfaz remota


import java.rmi.Remote;
import java.rmi.RemoteException;


public interface icontador extends Remote {
    int sumar() throws RemoteException;
    void sumar (int valor) throws RemoteException;
    public int incrementar() throws RemoteException;

}



# Fichero: contador.java
# Implementa la Interfaz remota


import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.net.MalformedURLException;


public class contador extends UnicastRemoteObject implements icontador {
  private int suma;

  public contador() throws RemoteException{
```

```
  }

  public int sumar() throws RemoteException {
    return suma;
  }


  public  void sumar(int valor) throws RemoteException {
    suma = valor;
  }

  public int incrementar() throws RemoteException {
    suma++;
    return suma;
  }
}



# Fichero: servidor.java
# Código del servidor


import java.net.MalformedURLException;
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.*;
//import contador.contador;


public class servidor {
  public static void main(String[] args) {


// Crea e instala el gestor de seguridad
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
      // Crea una instancia de contador

      //System.setProperty("java.rmi.server.hostname","192.168.1.107");
      Registry reg=LocateRegistry.createRegistry(1099);
      contador micontador = new contador();
      Naming.rebind("mmicontador", micontador);

      //suma = 0;

      System.out.println("Servidor RemoteException | MalformedURLExceptiondor preparado");
    } catch (RemoteException | MalformedURLException e) {
      System.out.println("Exception: " + e.getMessage());
    }
  }
}



# Fichero: cliente.java
# Código del cliente


import java.net.MalformedURLException;
import java.rmi.registry.LocateRegistry;
import java.rmi.*;
import java.rmi.registry.Registry;

public class cliente {
  public static void main(String[] args) {

// Crea e instala el gestor de seguridad
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
```

```
      // Crea el stub para el cliente especificando el nombre del servidor
      Registry mireg = LocateRegistry.getRegistry("127.0.0.1", 1099);

      icontador micontador = (icontador)mireg.lookup("mmicontador");

      // Pone el contador al valor inicial 0
      System.out.println("Poniendo contador a 0");
      micontador.sumar(0);

      // Obtiene hora de comienzo
      long horacomienzo = System.currentTimeMillis();

      // Incrementa 1000 veces
      System.out.println("Incrementando...");
      for (int i = 0; i < 1000; i++ ) {
        micontador.incrementar();
      }

      // Obtiene hora final, realiza e imprime calculos
      long horafin = System.currentTimeMillis();
      System.out.println("Media de las RMI realizadas = "
                         + ((horafin - horacomienzo)/1000f)
                         + " msegs");
      System.out.println("RMI realizadas = " + micontador.sumar());
    } catch(NotBoundException | RemoteException e) {
      System.err.println("Exception del sistema: " + e);
    }
    System.exit(0);
  }
}
```

1.3 Ejercicio

El siguiente ejercicio pretende que el estudiante aprenda a diseñar y programar aplicaciones Clientes-Servidor. Aún siendo un caso sencillo de este tipo de aplicaciones, las interacciones a resolver entre sus componentes permiten abordar los aspectos más importantes (distribución, funcionalidad, concurrencia, etc.) a tener en cuenta en el desarrollo de aplicaciones cliente-servidor de dos etapas.

El ejercicio consiste en desarrollar en RMI un sistema cliente-servidor teniendo en cuenta los siguientes requisitos:

1. El servidor será un servidor replicado (con exactamente 2 replicas), cada replica desplegada en una máquina diferente, y estará encargado de recibir donaciones de entidades (clientes) para una causa humanitaria.

2. El servidor proporcionará dos operaciones, registro de una entidad interesada (cliente) en la causa, y depósito de una donación a la causa. No es posible realizar un depósito (o más) sin haberse registrado como cliente previamente.

3. Cuando una entidad desea registrarse y contacta con cualquiera de las dos réplicas del servidor, entonces el registro del cliente debe ocurrir realmente y de forma transparente en la réplica con menos entidades registradas. Es decir, el cliente sólo se ha dirigido a una réplica aunque esta no haya sido donde realmente ha quedado registrado, pero a

partir de ese momento, el cliente realizará los depósitos en la réplica del servidor donde ha sido registrado.

4. Cada réplica del servidor mantendrá el subtotal de las donaciones realizadas en dicha replica.

5. Un cliente no podrá registrarse más de una vez, ni siquiera en replicas distintas.

6. Los servidores también ofrecerán dos operaciones de consulta: total donado y listado de donantes. Dichas operaciones sólo podrán llevarse a cabo si el cliente previamente se ha registrado y ha realizado al menos un depósito. Cuando un cliente realice alguna de estas consultas, sólo hará la petición a la réplica donde se encuentra registrado, y ésta será la encargada de devolver el resultado después de solicitar la información oportuna a la otra replica.

NOTAS PARA IMPLEMENTARLO EN NETBEANS:

Los ficheros utilizados son similares, la mayor diferencia reside en la importación de los paquetes de las interfaces en los programas cliente y servidor (icontador y contador).

1. Crear un proyecto *Java Aplication* Interface (*icontador*) con el fichero *icontador.java*

```
// icontador.java
package icontador;

import java.rmi.Remote;
import java.rmi.RemoteException;


public interface icontador extends Remote {
    int sumar() throws RemoteException;
    void sumar (int valor) throws RemoteException;
    public int incrementar() throws RemoteException;

}
```

2. Crear la librería (icontador.jar). Botón derecho ratón sobre el proyecto -> *clean and build*

3. Crear el proyecto que implementa el interfaz (*contador*)

```
// contador.java
package contador;

import icontador.icontador;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.net.MalformedURLException;


public class contador extends UnicastRemoteObject implements icontador {
  private int suma;

  public contador() throws RemoteException{

  }

  public int sumar() throws RemoteException {
    return suma;
  }


  public  void sumar(int valor) throws RemoteException {
    suma = valor;
  }

  public int incrementar() throws RemoteException {
    suma++;
    return suma;
  }
}
```

4. Importar la librería (*icontador*) en el proyecto contador. Botón derecho ratón sobre el proyecto -> *properties -> libraries ->Add JAR/Folder* y buscar e incluir el fichero (*icontador/dist/icontador.jar*)

5. Crear la librería (*contador.jar*). Botón derecho ratón sobre el proyecto -> *clean and build*

6. Crear el proyecto servidor, cargando las librerías (i*contador.jar y contador.jar*)

```
// servidor.java = Programa servidor
package servidor;

import contador.contador;
import java.net.MalformedURLException;
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.*;


public class servidor {
  public static void main(String[] args) {

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

    try {
           // Crea una instancia de contador

           Registry reg=LocateRegistry.createRegistry(1099);
           contador micontador = new contador();
           Naming.rebind("mmicontador", micontador);
           System.out.println("Servidor RemoteException | MalformedURLExceptiondor preparado");
    } catch (RemoteException | MalformedURLException e) {
           System.out.println("Exception: " + e.getMessage());
    }
  }
}
```

7. Crear el proyecto cliente cargando las librerías (i*contador.jar y contador.jar*)

```
// cliente.java
package cliente;

import icontador.icontador;
import java.rmi.registry.LocateRegistry;
import java.rmi.*;
import java.rmi.registry.Registry;

public class cliente {
  public static void main(String[] args) {

      String host = "";
      Scanner teclado = new Scanner (System.in);
      System.out.println ("Escriba el nombre o IP del servidor: ");
      host = teclado.nextLine();

    // Crea e instala el gestor de seguridad
if (System.getSecurityManager() == null) {System.setSecurityManager(new SecurityManager());}
    try {
      // Crea el stub para el cliente especificando el nombre del servidor
      Registry mireg = LocateRegistry.getRegistry(host, 1099);

      icontador micontador = (icontador)mireg.lookup("mmicontador");

      // Pone el contador al valor inicial 0
      System.out.println("Poniendo contador a 0");
      micontador.sumar(0);

      // Obtiene hora de comienzo
      long horacomienzo = System.currentTimeMillis();

      // Incrementa 1000 veces
      System.out.println("Incrementando...");
      for (int i = 0 ; i < 1000 ; i++ ) {
```

```
        micontador.incrementar();
      }
      // Obtiene hora final, realiza e imprime calculos
      long horafin = System.currentTimeMillis();
      System.out.println("Media de las RMI realizadas = "+ ((horafin - horacomienzo)/1000f)
                         + " msegs");
      System.out.println("RMI realizadas = " + micontador.sumar());
    } catch(NotBoundException | RemoteException e) {
      System.err.println("Exception del sistema: " + e);
    }
    System.exit(0);
  }
}
```

8. Cambiar las políticas de seguridad del servidor.

    a. Crear el fichero server.policy (en el ejemplo se ha creado en el mismo directorio que la fuente del servidor servidor.java).

```
// server.policy
grant {
    permission java.security.AllPermission;
};
```

    b. Botón derecho ratón sobre el proyecto -> *properties* -> *run* -> *y en VM option poner: -Djava.security.manager -Djava.security.policy=./src/servidor/server.policy*

9. Ejecutar el servidor. Botón derecho ratón -> *run*.

10. Cambiar las políticas de seguridad del cliente.

    a. Crear el fichero client.policy (en el ejemplo se ha creado en el mismo directorio que la fuente del servidor cliente.java).

    b. Botón derecho ratón sobre el proyecto -> *properties* -> *run* -> *y en VM option poner: -Djava.security.manager -Djava.security.policy=./src/cliente/client.policy*

11. Ejecutar el cliente. Botón derecho ratón -> *run*.

NOTAS ADICIONALES:

1. Por defecto *rmiregistry* coge las clases que hay en el classpath por defecto de vuestra máquina virtual de Java. Por tanto, si no modifica nada, tendrá que lanzarlo desde la carpeta donde estén los .class compilados. Si usa javac, como en los ejemplos, habrá de hacerlo desde el directorio donde ha compilado, tal y como hace el script de compilación. Por contra, si usa NetBeans, los .class se crean en la carpeta build/classes y tendrá que lanzarlo desde esa ubicación.

2. Por esa razón, los ficheros .policy y los argumentos de la VM son ligeramente distintos entre su ejecución por línea de órdenes y NetBeans.

3. En Netbeans puede crear el cliente y el servidor en el mismo proyecto, aunque en la vida real se hace en proyectos separados, que comparten una interfaz en un proyecto aparte.

4. En NetBeans debe crear varias *configuraciones*: una para el cliente, con sus parámetros de la VM y los argumentos del programa, y una para cada servidor replicado (con sus argumentos específicos al servidor y los parámetros de la VM comunes).

5. Si compila desde línea de órdenes, no olvide el "-cp .".

6. Escriba bien los argumentos de ejecución. Si no encuentra el fichero, o está mal escrito, la ejecución no le avisa.