Práctica 4. Objetos remotos. Java RMI

SCS, 2010/11

11 de enero de 2011

Índice

1. Ejercicio 1: Compilación y uso de objetos remotos en Java RMI

Se mostrará el un ejemplo simple de creación de objetos remotos en Java RMI.

Se usará el código de ejemplo que implementa una *"calculadora remota"* comentado en los apuntes de la asignatura.

Descarga: rmi1.tar.gz rmi1.zip

El código de partida incluye:

- `Calculadora.java`: Interfaz remoto a implementar por las calculadoras remotas
- `servidor/CalculadoraImpl.java`: Clase de implementación del interfaz remoto que hereda de `UnicastRemoteObject`
  Se incluye también una implementación que no hereda de esa clase (`CalculadoraImpl2.java`)
- `servidor/EjemploServidor.java`: Servidor simple que crea y exporta una instancia de la calculadora remota y la registra en `rmiregistry` con el nombre "Calculadora"
  Se incluye también un servidor para usar la implementación del interfaz remoto que no hereda de `UnicastRemoteObject` (`EjemploServidor2.java`, realiza exportación explícita)
- `cliente/EjempoCliente.java`: Cliente simple que obtiene del `rmiregistry` una referencia a una calculadora remota y ejecutas sus métodos

1.1. Compilación

1. Descomprimir el paquete.

```
$ tar xzvf rmi1.tar.gz
$ cd rmi1
```

2. Compilar el interfaz remoto

   ```
   $ javac Calculadora.java
   ```

3. Compilar la implementación y el cliente

   Nota: es necesario copiar el fichero `.class` del interfaz al directorio `cliente`

   ```
   $ cp Calculadora.class cliente
   $ cd cliente
   $ javac EjemploCliente.java
   ```

4. Compilar el servidor

   Nota: es necesario copiar el fichero `.class` del interfaz al directorio `servidor`

   ```
   $ cd ..
   $ cp Calculadora.class
   $ cd  servidor
   $ javac CalculadaraImpl.java
   $ javac EjemploServidor.java
   ```

1.2. Ejecución

Desde tres terminales distintos

1. Lanzar el servicio de nombres `rmiregistry`

   ```
   $ rmiregistry &
   ```

   Queda a la espera en el puerto 1099

2. Lanzar el servidor (queda a la espera)

   ```
   $ cd servidor
   $ java -Djava.rmi.server.codebase=file:///home/alumno/<...>/rmi1/servidor/  EjemploServidor
     [también puede usarse EjemploServidor2]
   ```

   Se debe especificar la propiedad de la JVM `java.rmi.server.codebase` para que el *rmiregistry* y los clientes sepan de donde descargar los *bytecodes* (ficheros .class) del *stub*

   En este caso se indica un directorio local (*file://....*)

3. Ejecutar el cliente

   ```
   $ cd cliente
   $ java EjemploCliente
   ```

1.3. Comentarios

Si el cliente hubiera sido compilado y ejecutado en una versión de Java (compilador + máquina virtual) anterior a la 1.5, hubiera sido necesario generar la clase `DiccionarioImpl_Stub` con el compilador **rmic**

Se puede ejecutar **rmic** para comprobar cómo sería el código de ese *stub*

```
$ cd servidor
$ rmic -keep CalculadoraImpl
```

La opción `-keep` fuerza que se genere el fichero `.java`

2. Ejercicio 2: Uso de "factorias" de objetos remotos

En ocasiones no es necesario (ni razonable) que todo objeto remoto de una aplicación esté registrado junto con su nombre en el servidor *remiregistry*. Normalmente esto es así en aplicaciones donde se maneje gran número de objetos (objetos *Cliente* y *Artículo* de un sistema empresarial, objeto *CuentaCorriente* en un sistema bancario)

En esos casos es habitual aplicar el patrón de diseño *Factoria*

- La creación de cierto tipo de objetos remotos se deje en manos de métodos específicos de otro objeto.
- Ese objeto "factoria" está especializado en crearlos (recuperando los datos que sean pertinentes de una B.D. por ejemplo) y exportarlos en el sistema RMI para hacerlos disponibles al proceso cliente.
- Ese objeto factoria si que estará registrado con un nombre en el servidor *remiregistry*.
- Los clientes recuperarán una referencia remota suya mediante consultas al registro y le pedirán que cree los objetos remotos que puedan necesitar.

En este ejercicio se verá una versión simplificada del patrón *Factoria* que gestiona la creación de objetos remotos para acceso a diccionarios, similares a los usados en la práctica 2.

Se manejarán 2 interfaces remotos distintos, junto con una clase de implementación para cada uno de ellos.

- Interfaces remotos
  - `DiccionarioRemoto`: métodos de consulta y modificación de diccionarios
  - `GestorDiccionarios`: métodos para crear y recuperar referencias a diccionarios remotos
- Clases de implementación
  - `DiccionarioRemotoImpl`: implementa los diccionarios
    - Cada diccionario tiene asociado un nombre *xxxx*
    - Físicamente recupera y guarda los datos en un fichero de texto de nombre *xxxx.dic*
    - Durante la ejecución usa una tabla hash con pares (palabra, defínción) cargados desde el fichero `xxxx.dic` para implementar las funciones de búsqueda en inserción
    - Sus métodos serán llamadas por los clientes y ejecutados en el servidor (donde residen los ficheros `xxxx.dic`)
  - `GestorDiccionariosImpl`: implementación del gestor de diccionarios
    - Ofrece acceso al diccionario genérico "general.dic", enviando referencias remotas (*stubs*) al objeto DiccionarioImpl que lo gestiona
    - Ofrece dos métodos factoria que devuelven referencias remotas *stubs* a diccionarios remotos
      - ◇ Método `crearDiccionario(nombre)`: creación de diccionarios nuevos (vacíos)
      - ◇ Método `obtenerDiccionario(nombre)`: carga desde fichero y hace disponibles diccionarios ya existentes
    - Ene sta implementación se mantiene una lista con diccionarios remotos gestionados (almacenada en una tabla hash) de modo que ante peticiones repetidas no sea necesario volver a crear/cargar los diccionarios

Se parte de un código de ejemplo con una implementación de los diccionarios remotos y una implementación parcial del gestor de diccionarios.

Descarga: rmi2.tar.gz rmi2.zip

2.1. Pasos a seguir

1. Descomprimir el paquete.

```
$ tar xzvf rmi2.tar.gz
$ cd rpc2
```

2. Contenido

   - `DiccionarioRemoto.java, DiccionarioRemoto.java`: Interfaces remotas (ya compiladas)
   - `servidor/DiccionarioRemotoImpl.java`: Implementación del diccionario
   - `servidor/GestorDiccionariosImpl.java`: Implementación del gestor de diccionarios
   - `servidor/Servidor.java`: Crea el gestor de diccionarios, lo registra en *rmiregistry* con el nombre "Gestor-Diccionarios" y se queda a la espera
   - `servidor/general.dic`: Fichero con el diccionario general de prueba
   - `cliente/Cliente`: Ejemplo de llamadas a los métodos remotos exportados
     - Obtiene del *rmiregistry* un *stub* para el gestor de diccionarios
     - Llama a los métodos factoria para obtener *stubs* de diccionarios
     - Invoca los métodos de esos diccionarios remotos
   - `cliente/nueva.policy`: Ejemplo de fichero de política de seguridad para permitir la descarga de *byetecodes* desde localizaciones remotas
     - Define una política donde todo está permitido
     - **Aviso**: este fichero no es útil en aplicaciones reales

2.2. Tareas a realizar

Se pide implementar los métodos factoria de `GestorDiccionariosImpl` del fichero `servidor/GestorDiccionarios.java` (ver el código para más detalles)

- `DiccionarioRemoto crearDiccionario(String nombre)`
  - Si el diccionario ya existe en la tabla hash de diccionarios gestionados o existe el correspondiente fichero `xxxxx.dic` devuelve *null* para indicar que no se puede crear un diccionario ya existente
  - En otro caso, se crea un diccionario vacio, se exporta al sistema RMI, se le pasa su *stub* al cliente y se toma nota en la tabla hash de diccionarios gestionados
- `DiccionarioRemoto obtenerDiccionario(String nombre)`
  - Si el diccionario ya existe en la tabla hash de diccionarios gestionados se devuelve el *stub* almacenado
  - Si está en esa tabla pero existe el correspondiente fichero `xxxxx.dic` se crea un diccionario, se carga sus palabras desde ese fichero, se exporta al sistema RMI, se le pasa su *stub* al cliente y se toma nota en la tabla hash de diccionarios gestionados
  - En otro caso se devuelve *null* para indicar que el diccionario no existe

Pasos para compilación (una vez implementado):

1. Compilar interfaces (ya está hecho)

```
$ javac DiccionarioRemoto.java
$ javac GestorDiccionarios.java
```

2. Compilar implementaciones y compilar el servidor

```
$ cp DiccionarioRemoto.class servidor
$ cp GestorDiccionarios.class servidor
$ cd servidor
$ javac DiccionarioRemotoImpl.java
$ javac GestorDiccionariosImpl.java
$ javac Servidor.java
```

3. Compilar el cliente

```
$ cp DiccionarioRemoto.class cliente
$ cp GestorDiccionarios.class servidor
$ cd cliente
$ javac Cliente.java
```

Pasos para ejecución (una vez compilado) [usar 3 terminales]: Desde tres terminales distintos

1. Lanzar el servicio de nombres `rmiregistry` [si no estaba lanzado]

   ```
   $ rmiregistry &
   ```

   Queda a la espera en el puerto 1099

2. Lanzar el servidor (queda a la espera)

   ```
   $ cd servidor
   $ java -Djava.rmi.server.codebase=file:///home/alumno/<...>/rmi2/servidor/  Servidor
   ```

3. Ejecutar el cliente (indicando el uso del fichero de políticas de seguridad `nueva.policy`)

   ```
   $ cd cliente
   $ java -Djava.security.policy=nueva.policy Cliente localhost
   ```

3. Documentación a entregar

- Para el **ejercicio 2** se entregará el framento de código fuente donde se implementan las funciones `DiccionarioRemoto crearDiccionario(String nombre)` y `DiccionarioRemoto obtenerDiccionario(String nombre)`

  Nota: Comentar los problemas y dificultades encontrados, si los hubo.

- El esquema seguido con el gestor de diccionarios es razonablemente eficiente en cuanto al uso de recursos, pero esta implementación concreta tiene un grave problema de cara a su utilización en un sistema distribuido real, donde múltiple cliente accederán a un conjunto de diccionarios compartidos.

  Describir el tipo de situaciones en las que la implementación del ejemplo se comportaría de forma incorrecta ante mútliples accesos/usos concurrentes.