ugr | Universidad de Granada

Desarrollo de Sistemas Distribuidos

Práctica 3: RMI

José Manuel Navarro Cuartero

Índice

Ejemplo 1

En este primer ejemplo tenemos una implementación simple en la que se llama a un servidor “Ejemplo” que hereda de la interfaz “Ejemplo_I” con un entero, y si ese entero es 0 se duerme el proceso durante 5 segundos.

```
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");
```

La implementación es simple. Se crea un objeto del tipo del servidor y se incluye en el registro.
El cliente por su parte buscará al servidor y llamará al método que hemos mencionado con el entero que se introduce en la llamada al ejecutable.

```
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]));
```

Resultado:

```
Ejemplo bound
Recibida peticion de proceso: 0
Empezamos a dormir
Terminamos de dormir

Hebra 0
Recibida peticion de proceso: 3

Hebra 3
```

Ejemplo 2

Este ejemplo será muy parecido al anterior, con la salvedad de que ahora se creará un único servidor, pero varias hebras de clientes que accederán a él a la vez, y se dormirán aquellas que sean múltiplo de 10.

```
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 el main del cliente crearemos tantas hebras como se indiquen en la llamada y conforme se creen se harán funcionar, llamando al servidor con su ID de hebra.
Probamos a llamar al cliente con 12 hebras (de la 0 a la 11) y vemos el resultado.

```
Ejemplo bound

Entra Hebra Cliente 11
Sale Hebra Cliente 11

Entra Hebra Cliente 5
Sale Hebra Cliente 5

Entra Hebra Cliente 8
Sale Hebra Cliente 8

Entra Hebra Cliente 0

Entra Hebra Cliente 3
Sale Hebra Cliente 3
Empezamos a dormirCliente 0

Entra Hebra Cliente 6
Sale Hebra Cliente 6
```

```
Entra Hebra Cliente 2
Sale Hebra Cliente 2

Entra Hebra Cliente 7
Sale Hebra Cliente 7

Entra Hebra Cliente 4
Sale Hebra Cliente 4

Entra Hebra Cliente 1

Entra Hebra Cliente 9
Sale Hebra Cliente 9
Sale Hebra Cliente 1

Entra Hebra Cliente 10
Empezamos a dormirCliente 10
Terminamos de dormirCliente 0
Sale Hebra Cliente 0
Terminamos de dormirCliente 10
Sale Hebra Cliente 10
```

Como podemos ver, para las hebras que no son 0 y 10 la ejecución es casi automática, y tal como entran, salen. Las dos hebras múltiplo son las que entran, y se quedan dormidas permitiendo la ejecución de las otras de mientras, y es por esto que son las últimas en salir.

Si en la función del cliente “escribir mensaje” hubiéramos añadido la palabra reservada “synchronized”, hubiéramos bloqueado la ejecución simultánea de dicha función, lo que provocaría que las hebras no puedan entrelazarse, y que, por ejemplo, mientras duerme la hebra 0, ninguna otra hebra podría entrar o salir.

Ejemplo 3

En este último ejemplo se nos presenta una implementación algo distinta. En los casos anteriores el servidor, en su main, creaba una instancia de sí mismo y la unía al registro, y el cliente accedía a la referencia a esta instancia a través del registro.
Sin embargo, en este ejemplo, el servidor se encarga de crear una instancia de un objeto distinto, al que llama “contador”, que es el que hereda de la interfaz, y es el que se une al registro, y al que accede el cliente.

```
Registry reg=LocateRegistry.createRegistry(1099);
contador micontador = new contador();
Naming.rebind("mmicontador", micontador);
```

El contador tendrá 3 funciones muy simples para manejar un entero con el cliente.

```
public interface icontador extends Remote {
    int sumar()
    void sumar (int valor)
    public int incrementar()
}
```

El cliente obtendrá a través del registro la referencia al contador y hará 1000 llamadas al mismo, además de medir cuánto tiempo tarda en cada una de ellas de media.

```
Incrementando...
Media de las RMI realizadas = 0.388 msegs
RMI realizadas = 1000
```

Donaciones

Introducción

Para nuestros servidores replicados de donaciones nos hemos basado en el ejemplo 3, ya que he considerado que a la hora de crear los servidores replicados era más correcto separar la creación del uso de los servidores.
Así pues, dispondremos de 3 clases distintas y 2 interfaces, a saber, Servidor, Cliente y Contador, y ésta última implementará las interfaces “Contador_I” y “Comunicador_I”.
El servidor creará dos réplicas del Contador, que serán con las que interactuarán los clientes, y que intercambiarán información entre ellas.

Interfaz Comunicador_I

Esta es la interfaz que describe el comportamiento relativo al intercambio de información entre los servidores.

```
public interface Comunicador_I extends Remote{
    public void setPareja(String id)
    public int subtotal()
    public boolean tieneCliente(String cliente)
    public int numClientes()
    public void registra(String cliente)
    public String getID()
    public void dona(String cliente, int cantidad)
}
```

Posteriormente describiremos el uso de los métodos, pero en general como podemos ver, son métodos que serán llamados por los propios servidores para hacer operaciones sobre sus datos, o sobre los del otro servidor.

Interfaz Contador_I

Esta interfaz es la que será conocida al cliente, y contiene los 3 métodos de los que éste hará uso.

```
public interface Contador_I extends Remote{
    public int registrar (String nombre)
    public int donar (String nombre, int cantidad)
    public int total_donado ()
}
```

Servidor

Como hemos descrito arriba, nuestra clase “Servidor” simplemente creará las instancias de los contadores, que serán los que actúen como servidores replicados de cara al cliente.

```
Contador_I contador1 = new Contador(ID1);
Contador_I contador2 = new Contador(ID2);

// Registramos los dos servidores contador
Registry registry = LocateRegistry.getRegistry();
registry.rebind(ID1, contador1);
registry.rebind(ID2, contador2);

// Enviamos a cada servidor una referencia al otro,
Comunicador_I com1 = (Comunicador_I)contador1,
    com2 = (Comunicador_I)contador2;
com1.setPareja(ID2);
com2.setPareja(ID1);
```

El servidor creará las instancias deseadas, y las ligará al registro con nombres distintos. En nuestro caso tenemos dos, pero sería trivial tener un vector con tantas réplicas como deseemos. Posteriormente se manda a cada servidor una referencia (casteada al tipo correcto) al otro para que puedan accederse mutuamente.

Contadores

Los contadores serán (en nuestro ejemplo) dos réplicas que actuarán como servidores, recibirán las peticiones de los clientes, y se comunicarán entre ellos. Distinguimos 3 partes en la clase “Contador”, a saber:

Set up

Aquí tenemos el constructor, que inicializará las variables de instancia, y el método en el que se busca en el registro al otro servidor para poder hacer uso de sus métodos.

```
// Métodos para poner el contador en marcha
public Contador(String id) throws RemoteException{
    this.recuento_local = new HashMap<>();
    this.ID=id;
    this.registry = LocateRegistry.getRegistry("localhost");
}
public void setPareja(String id) throws RemoteException{
    try{
        pareja = (Comunicador_I)registry.lookup(id);
    } catch (Exception e) {
        System.err.println("Comunicador_I exception:");
        e.printStackTrace();
    }
}
```

Comunicación con el cliente

Aquí tenemos los 3 métodos que definimos en la interfaz “Contador_I”.
En registrar, comprobamos si el cliente está registrado en el servidor al que se llama o en el otro, y en caso negativo se le registra en el servidor con menor número de clientes.
Para almacenar a los clientes y sus donaciones se usa un map con el nombre del cliente de key y sus donaciones de value. El valor que se retorna es -1 o -2 si el cliente estaba ya registrado en el servidor 1 o 2, respectivamente; o 1 o 2 si se le ha registrado satisfactoriamente.

```
public int registrar (String nombre) throws RemoteException{
    int aux = 0;
    if(this.tieneCliente(nombre)){
        System.out.println(this.ID+"-> "+nombre+" ya estaba registrado en "+this.getID());
        aux=Character.getNumericValue(this.getID().charAt(9))*-1;
    } else if (pareja.tieneCliente(nombre)){
        System.out.println(this.ID+"-> "+nombre+" ya estaba registrado en "+pareja.getID());
        aux=Character.getNumericValue(pareja.getID().charAt(9))*-1;
    } else {
        if(this.numClientes()>pareja.numClientes()){
            pareja.registra(nombre);
            aux=Character.getNumericValue(pareja.getID().charAt(9));
        } else{
            this.registra(nombre);
            aux=Character.getNumericValue(this.getID().charAt(9));
        }
    }
    return aux;
```

De igual forma, en donar se comprueba si el cliente está registrado previamente, y si sí, se almacena su donación en el servidor en el que está registrado.

```
public int donar (String nombre, int cantidad) throws RemoteException{
    int aux = -1;
    if(!this.tieneCliente(nombre) && !pareja.tieneCliente(nombre)){
        System.out.println(this.ID+"-> Para donar hay que estar registrado.");
        aux=0;
    } else if(this.tieneCliente(nombre)){
        this.dona(nombre,cantidad);
        aux=Character.getNumericValue(this.getID().charAt(9));
    } else {
        pareja.dona(nombre,cantidad);
        aux=Character.getNumericValue(pareja.getID().charAt(9));
    }
    return aux;
}
```

Por último, para calcular el total donado se suman los subtotales de ambos servidores.

```
public int total_donado () throws RemoteException{
    int total = 0;
    total+=this.subtotal();
    total+=pareja.subtotal();
    System.out.println(this.ID+"-> Se ha donado un total de "+total);
    return total;
}
```

Comunicación entre los servidores

Las funciones para la comunicación entre los servidores vienen definidas en la interfaz “Comunicacion_I” y en general son más sencillas, solo están diseñadas para habilitar el funcionamiento.
Subtotal y numClientes devuelven datos relativos al map local de donaciones, la suma de las cantidades y el numero de claves.

```
public int subtotal() throws RemoteException{
    int subtotal = 0;
    for (int x : recuento_local.values())
        subtotal+=x;
    System.out.println(this.ID+"-> Subtotal de "+subtotal);
    return subtotal;
}
public int numClientes() throws RemoteException{
    return this.recuento_local.keySet().size();
}
```

De igual manera, tieneCliente comprueba la existencia de una clave, y getID devuelve el string que identifica al servidor de manera inequívoca en el registro.

```
public boolean tieneCliente(String cliente) thr
    return recuento_local.containsKey(cliente);
}
public String getID() throws RemoteException{
    return this.ID;
}
```

Por último, registra guarda a un nuevo cliente (siempre será nuevo), y dona actualiza el value asociado al key cliente.

```
public void registra(String cliente)
    this.recuento_local.put(cliente,0);
    System.out.println(this.ID+"-> "+cliente
    +" se ha registrado en este servidor.");
}
public void dona(String cliente, int cantidad)
    int aux = recuento_local.get(cliente);
    aux+=cantidad;
    recuento_local.put(cliente, aux);
    System.out.println(this.ID+"-> "+cliente
    +" ha donado "+cantidad+" en este servidor");
}
```

Cliente

Para finalizar tenemos a la clase cliente. En nuestro caso la hemos implementado teniendo en mente que se podrían hacer varios accesos paralelos de varios clientes, o incluso con un pequeño cambio muchos clientes en hebras, aunque por comodidad de uso se ha diseñado una pequeña interfaz para que se use desde la consola sin mucho problema.

Main

En el main crearemos referencias a los dos servidores que obtendremos del registro, y luego, de forma aleatoria, escogeremos uno con el que interactuar. Crearemos un objeto cliente con ese servidor, y daremos marcha al loop de uso “normal”.

```
Registry registry = LocateRegistry.getRegistry("localhost");
Contador_I local1 = (Contador_I) registry.lookup(ID1);
Contador_I local2 = (Contador_I) registry.lookup(ID2);
Random random = new Random();
// La asignacion del servidor con el que contacta el cliente es aleatoria
Contador_I servidor = random.nextInt(2)==0?local1:local2;
Cliente cliente = new Cliente(servidor);
cliente.loop();
```

Loop

En el loop normal simplemente se pregunta el nombre al cliente y se le dan las opciones sobre lo que puede hacer. El cliente decidirá qué hacer a través de la consola introduciendo distintas letras.

```
public void loop() throws RemoteException{
    nombreCliente = intro();
    informacion();
    opcionCliente = preguntar();
    while(opcionCliente != 'O'){
        switch (opcionCliente){
            case 'R': this.registrar(); break;
            case 'D': this.donar(); break;
            case 'T': this.total(); break;
            case 'U': this.cambio(); break;
            case 'I': this.informacion(); break;
        }
        opcionCliente = preguntar();
    }
}
```

Funciones auxiliares

En el loop se hace llamada a diversas funciones, como registrar, donar y total. Estas funciones serán las encargadas de llamar a los métodos homónimos del servidor, y de interpretar la respuesta de éste.

```
private void donar() throws RemoteException{
    System.out.println("Cuanto quieres donar?");
    int cantidad = Integer.parseInt(System.console().readLine());
    int aux = servidor.donar(nombreCliente, cantidad);
    switch(aux){
        case 0: System.out.println("Para donar antes hay que estar registrado"); break;
        case 1: System.out.println(cantidad+" donado en el servidor 1."); break;
        case 2: System.out.println(cantidad+" donado en el servidor 2."); break;
        default: System.out.println("Problema en la donacion -> "+aux);
    }
}
```

```
private void registrar() throws RemoteException{
    int aux = this.servidor.registrar(nombreCliente);
    switch(aux){
        case -1: System.out.println("El cliente ya estaba registrado en el servidor 1."); break;
        case -2: System.out.println("El cliente ya estaba registrado en el servidor 2."); break;
        case 1: System.out.println("El cliente ha sido registrado en el servidor 1."); break;
        case 2: System.out.println("El cliente ha sido registrado en el servidor 2."); break;
        default: System.out.println("Problema en el registro -> "+aux);
    }
}
```

```
private void total() throws RemoteException{
    int total = servidor.total_donado();
    System.out.println("Se ha donado un total de "+total);
}
```

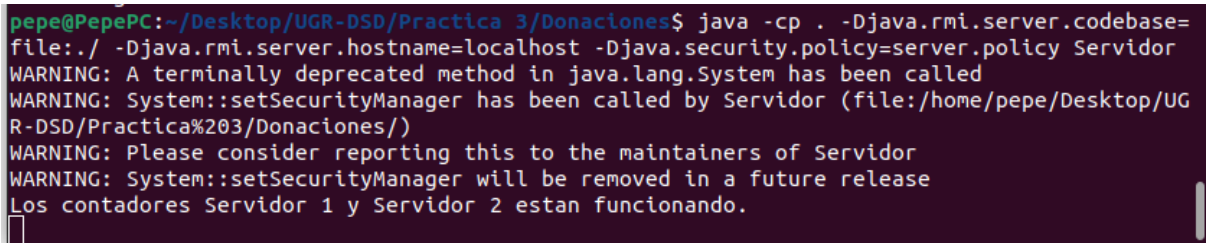
Además, tendremos otras pequeñas funciones que se encargarán de sacar por pantalla información para el usuario, como cambio, preguntar, información e intro.

```
private void informacion(){
    System.out.println("I -> Informacion, R -> Registrarme, D -> Donar");
    System.out.println("T -> Total donado, U -> Cambiar de cliente, O -> Salir");
}
```

```
private char preguntar(){
    System.out.println(nombreCliente+", qué deseas hacer?");
    char c = System.console().readLine().charAt(0);
    c = Character.toUpperCase(c);
    return c;
}
```

Uso

Para usar la aplicación, primero nos aseguraremos de que no hay ningún rmi previo con el comando “pkill rmiregistry”. Posteriormente haremos la llamada con “rmiregistry &”. Los warnings respecto del uso del gestor de seguridad se pueden ignorar. Tras esto compilaremos los archivos con “javac *.java”.
En una terminal llamaremos al servidor para que cree las réplicas con:
java -cp . -Djava.rmi.server.codebase=file:./ -Djava.rmi.server.hostname=localhost -Djava.security.policy=server.policy Servidor



La propia terminal debería informarnos de que los servidores están funcionando. Tras esto, en otra terminal (o en otro PC si así los deseáramos), hacemos:
java -cp . -Djava.security.policy=server.policy Cliente
Y ya podremos usarlo.