Desarrollo de Aplicaciones Distribuidas

AUTORES:
Alejandro Calderón Mateos
Javier García Blas
David Expósito Singh
Laura Prada Camacho

Departamento de Informática
Universidad Carlos III de Madrid
Julio de 2012

JAVA RMI:
ESTRUCTURA INTERNA RMI

Contenidos

1. Factory Pattern para RMI

2. *Comentarios Dudas-Frecuentes*

3. *Bootstrap Problem*

FACTORY PATTERN PARA RMI

Factory Pattern para RMI

- Concepto introducido en: Design Patterns, Elements of Reusable Object-Oriented Software.
- Objeto que controla la creación y/o el acceso a otros objetos.
- En el contexto de Java RMI: reducción en el número de objetos que es necesario registrar.

Factory Pattern para RMI

- Ejemplo práctico: la biblioteca.

1. Registro de nuevos usuarios para entrar en la biblioteca.
2. Identificación tarjeta de usuario.
3. Control de préstamo de libros.
4. Reserva de libros para cada usuario.

- *El bibliotecario representa una factoría dado que gestiona el acceso al contenido de la biblioteca.*

Factory Pattern para RMI

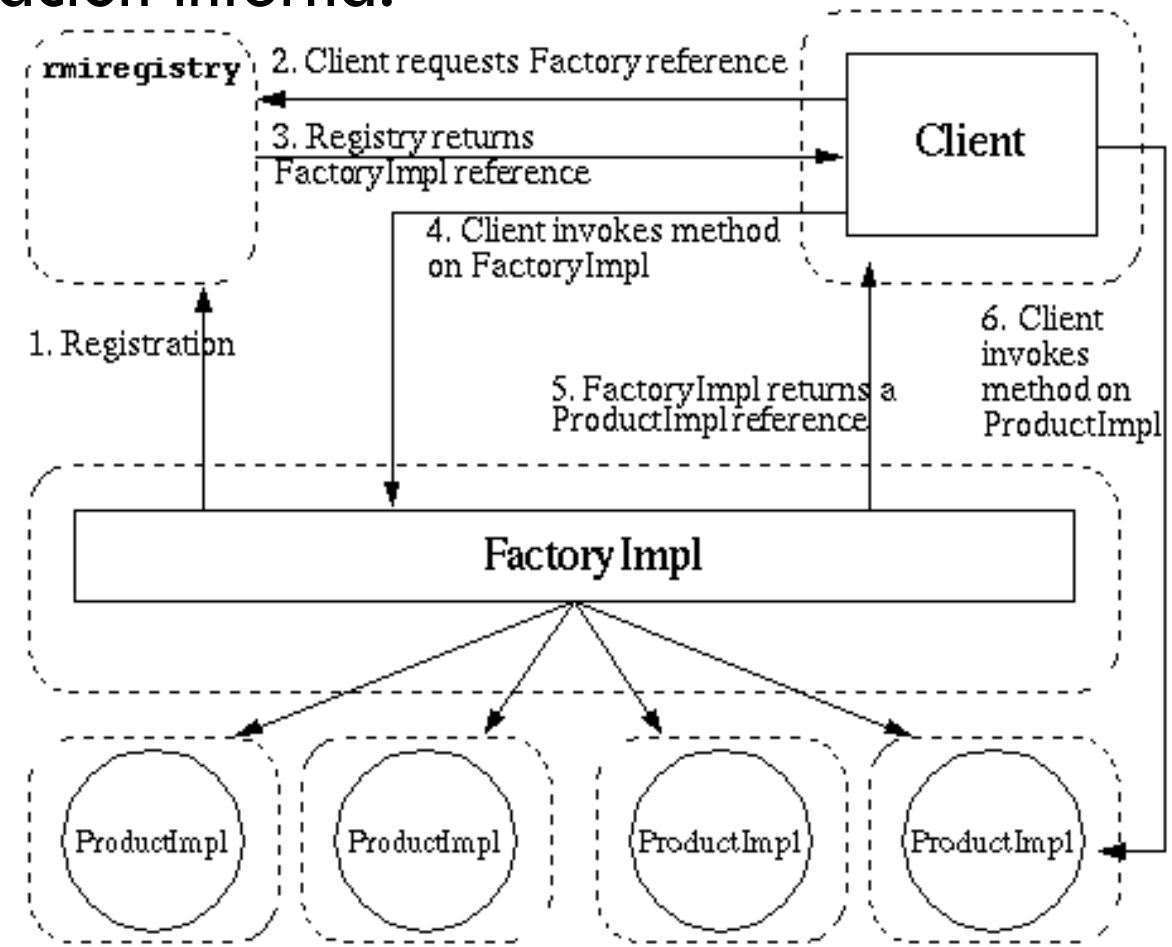
- Organización interna:
  - Hay dos únicas interfaces remotas que entiende el cliente: *Factory* y *Product*.
  - *FactoryImpl* implementa *Factory interface*.
  - *ProductImpl* implementa *Product interface*.

Factory Pattern para RMI

- Organización interna:
  - *FactoryImpl* se registra en el registro RMI.
  - El cliente solicita una referencia a *Factory*.
  - *Rmiregistry* devuelve una referencia remota a *FactoryImpl*.
  - El cliente invoca un método remoto de *FactoryImpl* para obtener una referencia remota de *ProductImpl*.
  - *FactoryImpl* devuelve una referencia remota de un objeto *ProductImpl* existente o bien uno que acaba de ser creado en función de la petición del cliente.
  - El cliente invoca un método remoto de *ProductImpl*.

Factory Pattern para RMI

- Organización interna:



Factory Pattern para RMI

- El bibliotecario representa un *interface* remoto (factoría de objetos) con uno o más métodos que devuelven objetos que implementan el *interface* *LibraryCard* (productos).

- El bibliotecario oferta métodos para acceder a los libros.

- La única implementación registrada en el registro RMI es la clase bibliotecario.

COMENTARIOS-DUDAS FRECUENTES

Comentarios y dudas frecuentes

Diferencias entre:

```
registry.rebind("rmi://localhost/FlightServices:1099",new RemoteConnection()");
```

Y

```
Naming.rebind("rmi://localhost/FlightServices:1099",new RemoteConnection()");
```

Estructura de Naming.rebind:

```
public static void rebind(String name, Remote obj)
    throws RemoteException, java.net.MalformedURLException
    {
    ParsedNamingURL parsed = parseURL(name);
    Registry registry = getRegistry(parsed);if (obj == null)
    throw new NullPointerException("cannot bind to null");
registry.rebind(parsed.name, obj);
    }
```

Comentarios y dudas frecuentes

RemoteStub rstub=UnicastRemoteObject.exportObject(RemoteObject MyClass)

Llamada al constructor de objeto remoto MyClass implica:

- Crear objeto MyClass (sólo accesible vía *skeleton*).
- Crear objeto Myclass_Skel y activar escucha a las peticiones del cliente.
- Devolver una referencia a objeto (asociado al *skeleton*) que es el objeto RemoteStub object.

- Objetos no persistentes.

BOOTSTRAP PROBLEM

Estructura interna de RMI

- Buscar respuesta a:
  - ¿Quién realmente crea el objeto de *stub*? ¿El servidor, el cliente o el registro?
  - ¿A qué puerto está escuchando el servidor?
  - ¿Acaso el servidor escucha en el puerto 1099? (puerto por defecto del registro RMI)
  - ¿Cómo sabe el cliente en qué puerto escucha el servidor?
  - ¿Es necesario el registro RMI para hacer funcionar el sistema?
  - ¿Se puede utilizar RMI sin el *rmiregistry*?

Estructura interna de RMI

- Planteamiento inicial: *ignorar el registro RMI*.
  - Tenemos un servidor y un cliente en máquinas distintas.
  - El servidor extiende *java.rmi.server.UnicastRemoteObject*.
  - El cliente quiere ejecutar un método remoto del servidor.

- Comunicaciones internas mediante *sockets*.

- *Stubs* y *Skeletons* contienen toda la información concerniente a las comunicaciones.

  Cliente ↔ *stub* ↔ [RED] ↔ *skeleton* ↔ Servidor

Estructura interna de RMI

- Funcionamiento a nivel de sockets:
  1. El servidor escucha un puerto.
  2. El cliente no sabe en qué maquina y puerto está escuchando el servidor. **Pero tiene un objeto *stub* que tiene esa información.**
  3. El cliente invoca la función del ***stub***.
  4. El *stub* se conecta al puerto del servidor y envía los parámetros siguiendo los siguientes pasos:
     1. El cliente se conecta al puerto de escucha del servidor.
     2. El servidor acepta la conexión entrante y crea un nuevo *socket* para gestionar esta única conexión.
     3. El antiguo puerto de escucha permanece aguardando posteriores peticiones de otros clientes.

Estructura interna de RMI

4. La comunicación cliente-servidor se realiza utilizando el nuevo *socket*.

5. Se realiza el intercambio de los parámetros del método siguiendo un protocolo preestablecido.

6. El protocolo puede ser JRMP (*Java Remote Method protocol*) o CORBA-compatible RMI-IIOP (*Internet Inter-ORB Protocol*).

5. El método es ejecutado en el servidor y el servidor es devuelto al cliente vía *stub*.

7. El *stub* devuelve el resultado al cliente como si hubiera ejecutado la función localmente.

Estructura interna de RMI

- Punto2: *El cliente no sabe en qué maquina y puerto está escuchando el servidor.* ***Pero tiene un objeto stub que tiene esa información.***
- Pero… **el puerto de escucha asociado a cada objeto remoto del servidor es dinámico.**
- **Este puerto se crea en el momento de ejecutar el servidor.**
- Pregunta: ¿Cómo sabe el cliente qué puerto tiene asociado el servidor si éste es dinámico?

Bootstrap Problem

Estructura interna de RMI

- Es necesario informar al cliente acerca del estado del servidor.
- Registro RMI:
  - *Hashmap* de dupletes {etiqueta, *Stub_object*}.
  - Clase *java.rmi.Naming*.
  - Puerto preestablecido (1099 por defecto).

Estructura interna de RMI

Ejemplo estructura de un *stub*:

```
public final class CalcImpl_Stub
    extends java.rmi.server.RemoteStub
    implements Calc, java.rmi.Remote
{
    private static final long serialVersionUID = 2;
    private static java.lang.reflect.Method $method_add_0;
    static {
        try {
            $method_add_0 = Calc.class.getMethod("add",
                new java.lang.Class[] {int.class, int.class});
        } catch (java.lang.NoSuchMethodException e) {
            throw new java.lang.NoSuchMethodError(
                "stub class initialization failed");
        }
    }

    // constructors
    public CalcImpl_Stub(java.rmi.server.RemoteRef ref) {
        super(ref);
    }
    // methods from remote interfaces
```

Estructura interna de RMI

```
// implementation of add(int, int)
   public int add(int $param_int_1, int $param_int_2)
      throws java.rmi.RemoteException
      try {
         Object $result = ref.invoke(this, $method_add_0,
                          new java.lang.Object[]
                          {new java.lang.Integer($param_int_1),
                          new java.lang.Integer($param_int_2)},
                          -7734458262622125146L);
      return ((java.lang.Integer) $result).intValue();
      } catch (java.lang.RuntimeException e) {
         throw e;
      } catch (java.rmi.RemoteException e) {
         throw e;
      } catch (java.lang.Exception e) {
         throw new java.rmi.UnexpectedException("undeclared checked
                                                exception", e);
      }
   }
}
```

Estructura interna de RMI

1. *RMIRegistry* se ejecuta en el servidor y es un objeto remoto con un **puerto conocido**.

2. El servidor se exporta a un puerto anónimo de la máquina servidora. Este puerto es desconocido a los clientes.

3. Cuando se invoca *Naming.rebind()*, se le pasa la referencia de la implementación del objeto creado. La clase *Naming* construye un objeto *stub* y lo asocia al objeto remoto del registro. Los pasos de creación del *stub* son:

    1. Carga la clase *CalcImpl_Stub* en la JVM.

    2. Toma *RemoteRef obj* de *c*. *RemoteRef ref=c.getRef();*Esta referencia encapsula todos los detalles del servidor, como nombre, dirección y número de puerto asociado.

Estructura interna de RMI

3. Utiliza este objeto *RemoteRef* para construir el *stub*:
   CalcImpl_Stub stub=new CalcImpl_Stub(ref);

4. Pasa este objeto *stub* al *RMIRegistry* para asociarlo con la etiqueta.

5. *RMIRegistry* almacena internamente la etiqueta y el objeto *stub* en un *hashmap*.

4. Cuando el cliente ejecuta *Naming.lookup()*, para la etiqueta como parámetro. El *RMIRegistry* devuelve el *stub* de vuelta al cliente.

5. Ahora el cliente conoce el nombre del servidor y el puerto de escucha asociado al objeto remoto. El cliente puede invocar al método remoto de su *stub* para ejecutarlo.

Estructura interna de RMI

- El registro RMI únicamente se emplea para realizar *bootstrapping*.