



Ingeniería del Software II  
Curso 2008/2009

Sergio Ilarri Artigas  
silarri@unizar.es



# Índice

---

- Introducción
- *Stubs y Skeletons (rmic)*
- Objetos Remotos y Objetos Serializables:
  - Interfaz Remota: ejemplo
  - Objeto Remoto: ejemplo
- El Servicio de Nombres (*rmiregistry*)
- Ejemplo de Servidor y de Cliente
- Intercambio de Referencias Remotas
- Anexo: Algunos Conceptos Más Avanzados



# Reflexión

---

¿Tienen los *sockets* alguna pega?

Necesita protocolos de aplicación



# Introducción

---

- En lugar de trabajar directamente con *sockets*, RMI:
  - Permite invocar métodos de objetos remotos de forma transparente
  - Asume un entorno Java homogéneo
  - Por tanto, podemos beneficiarnos de su modelo de objetos



# Reflexión

---

¿Cómo podríamos invocar objetos remotos de forma transparente?

Que un objeto local se haga pasar por el objeto remoto y se encargue él de las comunicaciones



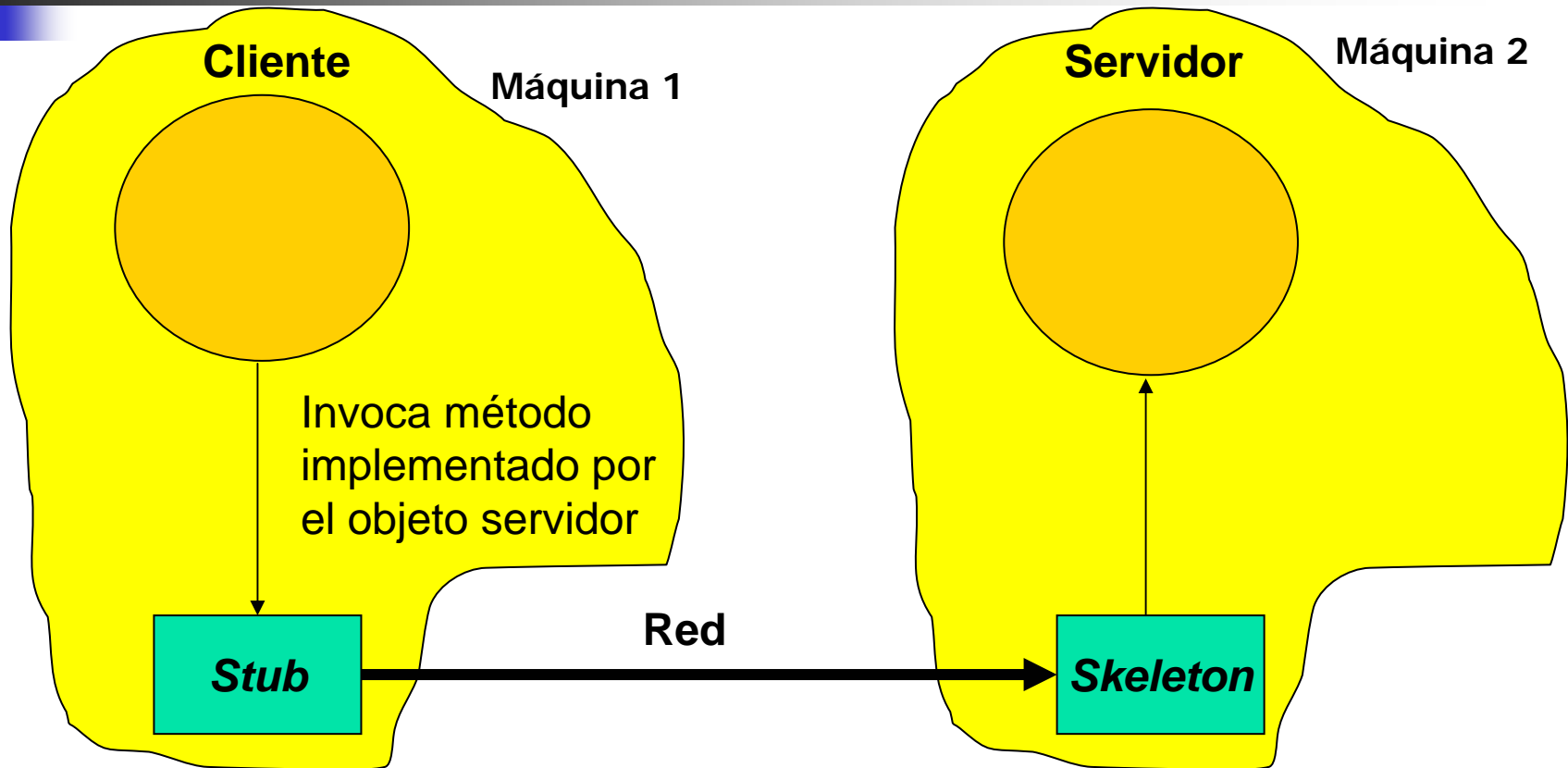
# Reflexión

---

¿Cómo podríamos programar un objeto que trate las invocaciones remotas como si fueran locales?

Que otro objeto local se encargue de las comunicaciones y delegue en él la ejecución de las operaciones invocadas

# Stubs y Skeletons



Codifica (*marshalling/serialization*) y envía los datos y la invocación por la red

Decodifica (*unmarshalling/deserialization*) los datos, invoca al método del servidor, devuelve resultados o excepciones



# *Stub*

---

- Cada clase de objetos remota tiene una clase ***stub*** asociada que implementa la interfaz remota:
  - Usada por el **cliente** en sustitución de la clase remota
  - Las invocaciones remotas del cliente en realidad se dirigen al *stub*
  - En la implementación de cada operación, se envía un mensaje con los parámetros de invocación *serializados* a la máquina virtual que ejecuta el objeto remoto





# Reflexión

---

¿A qué patrón suena esto?



Visto en clase...

Un *stub* es un *Proxy*



# *Skeleton*

---

- Clase usada por el servidor:
  - Recibe los mensajes remotos
  - *Deserializa* los parámetros de invocación
  - Invoca el método del objeto que implementa la interfaz remota
  - *Serializa* los resultados y los devuelve al llamador, así como posibles excepciones



# Reflexión

---

¿A qué patrón suena esto?

Un *skeleton* es un *Adapter/Wrapper*

# Objetos Remotos y Serializables

## ■ Objetos remotos:

- Reciben invocaciones remotas
- Se pasan por referencia (remota)
- Tienen localización fija
- Implementa la *interfaz remota*

*implements  
Remote*

## ■ Objetos serializables:

- Encapsulan datos
- Se pasan por valor
- Pueden transferirse a otra máquina virtual

*implements  
Serializable*

Todo argumento  
o valor de retorno  
en RMI



# Interfaz Remota (I)

---

- Define el protocolo de alto nivel entre cliente y servidor
- Es una interfaz Java normal pero que extiende ***java.rmi.Remote***
- En él el servidor declara los métodos disponibles remotamente para los clientes
- Todos esos métodos deben lanzar ***java.rmi.RemoteException***
- Todos los argumentos y resultados deben ser ***serializables***



# Ejemplo de Interfaz Remota

---

MessageWriter.java

```
import java.rmi.* ;  
public interface MessageWriter extends Remote  
{  
    void writeMessage(String s) throws RemoteException ;  
}
```



# Reflexión

---

¿Cómo conseguimos evitar que un método de un objeto remoto pueda ser invocado remotamente?

No definiéndolo en una interfaz que extienda  
*java.rmi.Remote*



# Reflexión

---

¿Puede un cliente acceder remotamente a métodos estáticos?

No, pues no pueden definirse en una interfaz Java (igual que los atributos no constantes)





# Reflexión

---

```
public interface ClockWithMillis extends Clock  
{  
    ...  
}
```

No aparece *extends Remote*... Pero os digo que es una interfaz remota... ¿Cómo es posible?

Será que *Clock* sí extiende la interfaz *Remote*



# Interfaz Remota (II)

---

- La interfaz *java.rmi.Remote*:
  - No declara ni métodos ni atributos
  - Su único propósito es “marcar” que cualquier interfaz que la extienda debe ser tratada por el sistema RMI como una interfaz remota
  - El compilador de RMI *rmic* genera clases *stub* y *skeleton* para aquellas clases que implementan interfaces remotas



# Reflexión

---

```
public interface Clock
```

```
{
```

```
    public long getTime();
```

```
}
```

---

```
public interface ClockWithMillis extends Clock, Remote
```

```
{...}
```

¿Por qué el ejemplo anterior no compilaría?

Porque *getTime* no declara que lanza  
*RemoteException*



# *RemoteException*

---

- Todos los métodos remotos deben declarar que pueden lanzar *java.rmi.RemoteException* (superclase de todas la excepciones RMI)
- Invocaciones remotas:
  - Sintácticamente como las locales
  - Pero pueden producirse fallos en la red, al serializar o deserializar argumentos, etc.
  - Se obliga al programador a tratar esos fallos
  - Las excepciones en el objeto “servidor” también se encapsulan y se propagan al “cliente”

*Exception chaining*  
(Java 1.4)



# El Objeto Remoto

---

- Instancia de una clase que implementa una interfaz remota
- Esta clase debe extender normalmente ***java.rmi.server.UnicastRemoteObject***

El constructor de la clase implementada debe lanzar *RemoteException*

- Constructor (sin argumentos) que exporta el objeto al sistema RMI
- El programador normalmente no usa esta clase explícitamente, simplemente se extiende
- Convención de nombrado:  
*<nombreInterfazRemota>Impl*



# Ejemplo de Objeto Remoto

MessageWriterImpl.java

```
import java.rmi.* ;  
import java.rmi.server.* ;  
public class MessageWriterImpl extends UnicastRemoteObject  
    implements MessageWriter {  
  
    public MessageWriterImpl() throws RemoteException {  
    }  
  
    public void writeMessage(String s) throws RemoteException {  
        System.out.println(s) ;  
    }  
  
}
```



# Compilador de RMI

---

- Utilizado para “compilar” clases que implementan *Remote* (clases de implementación remotas)
- ***rmic*** (parte del JDK):
  - *rmic ClassName* genera las clases:
    - *ClassName\_Stub* (implementa la interfaz remota)
    - *ClassName\_Skel* (no con la versión del protocolo stub 1.2)
  - *Con -keep*, mantiene el código fuente intermedio del *stub* y del *skeleton*



# Reflexión

---

Un “cliente” de una máquina desea comunicar con un “servidor”.

¿Cómo sabe dónde está?

- La localización se codifica en el código cliente
- El usuario indica la localización
- Nivel de indirección: servicio de nombres





# Servicio de Nombres (I)

---

- Funciona como un listín telefónico:
  - Permite al cliente saber dónde se está ejecutando el objeto con el que desea contactar
  - Objetivo: independencia de localización

*java.rmi.Naming*



# Servicio de Nombres (II)

---

En *java.rmi.Naming* tenemos los siguientes métodos estáticos:

```
public static void bind(String name, Remote object)  
public static void rebind(String name, Remote object)  
public static void unbind(String name)  
public static String[] list(String name)  
public static Remote lookup(String name)
```

Un parámetro nombre (*name*) es una URL con el formato:

*//registryHost:port/logical\_name*

Por defecto, el host (opcional) es localhost

Por defecto, el puerto (opcional) es 1099



# Servicio de Nombres (III)

## Bind y rebind

Se abre un socket con el *registry*  
(servidor de nombres)

*bind()*,  
*rebind()*,  
*unbind()*

Se serializa el *stub* que implementa el  
objeto remoto

Se liga dicho *stub* con un nombre

El servidor asocia un nombre lógico al objeto



# Servicio de Nombres (IV)

---

- *lookup()*: devuelve el *stub* ligado al nombre dado

El cliente obtiene una referencia al objeto a partir del nombre

*list()*,  
*lookup()*

- *list()*: devuelve los nombres de todos los objetos remotos registrados en ese servidor de nombres



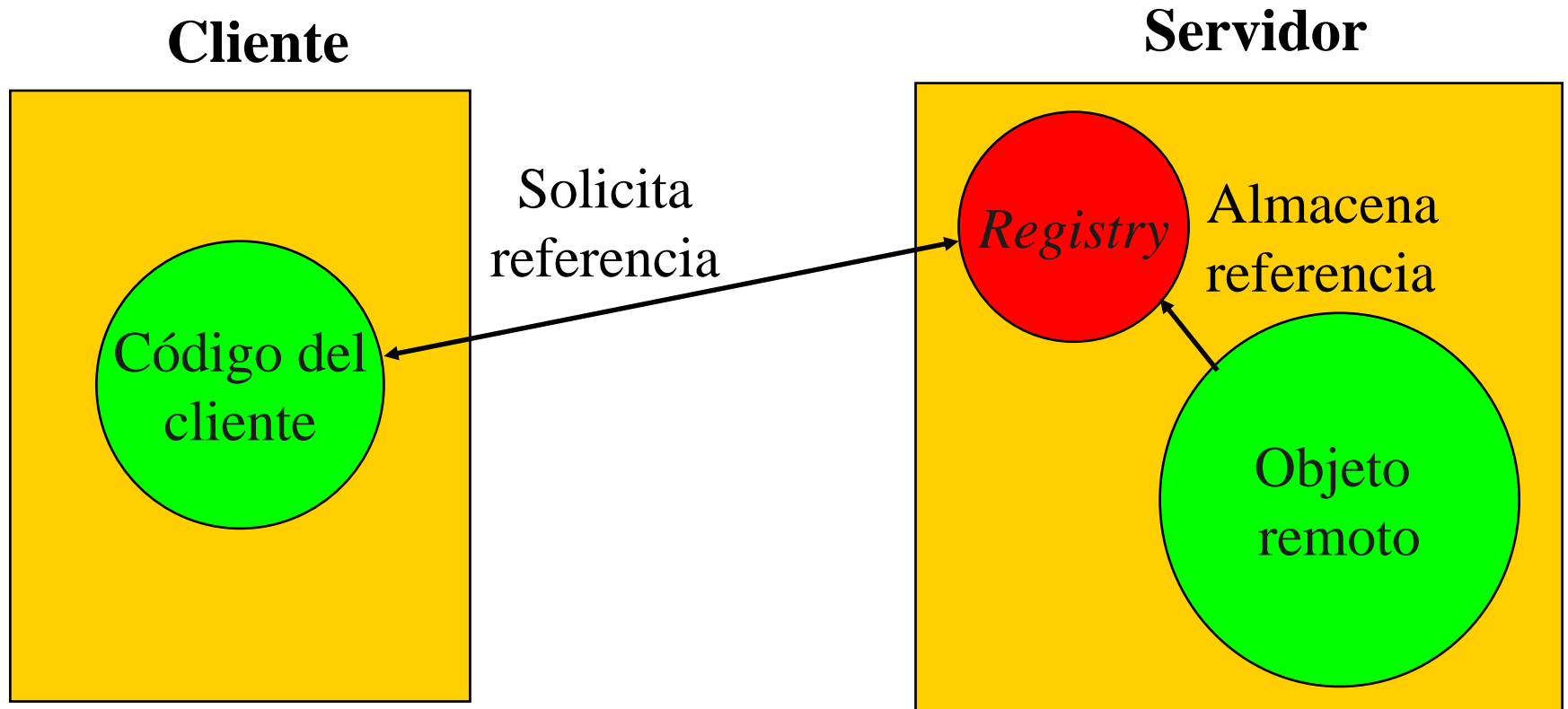
# Servicio de Nombres (V)

---

- *java.rmi.Naming* es una clase de utilidad que registra/contacta objetos a partir de una URL:
  - A partir de la URL, determina un objeto remoto *registry*, que implementa la interfaz *java.rmi.registry.Registry*, utilizando la clase de utilidad *java.rmi.registry.LocateRegistry*
  - Y ahora registra o contacta el objeto invocando un método de instancia, análogo al estático usado en *java.rmi.Naming*, sobre el objeto *registry*

# Servicio de Nombres (VI)

Normalmente, el *registry* está en el servidor donde se aloja el objeto remoto





# Servicio de Nombres (VII)

---

- Por tanto, distinguimos 3 pasos:
  1. Se llama a un método estático de *Naming* con una cierta URL
  2. Se analiza la URL y la información de máquina y puerto se utiliza para, a través de la clase *LocateRegistry*, obtenerse el stub del servidor de nombres (*registry*) correspondiente
  3. *Naming* usa el stub para invocar el método apropiado del servidor de nombres



# Servicio de Nombres (VIII)

---

- Aplicación *rmiregistry*:
  - Contiene un objeto que implementa *java.rmi.registry.Registry*
  - No es persistente
  - Por seguridad, no permite (de)registrar objetos desde otra máquina
- Cómo lanzar el servidor de nombres:
  - *rmiregistry 50500*
  - Desde código, en *LocateRegistry*:  
*public static createRegistry(int port)*





# Servicio de Nombres (IX)

---

- Algunos inconvenientes:
  - No es independiente de la localización:
    - La URL del objeto lleva la dirección y puerto del *rmiregistry* correspondiente
    - Si el *rmiregistry* cambia de máquina, hay que cambiar los “clientes” o configurarlos para que “lean” la localización actual de algún sitio
  - Es una estructura de nombres “plana”
    - No se permiten jerarquías (ej.: */is2/MessageWriter*)
    - No es escalable



# Ejemplo de Servidor

HelloServer.java

```
import java.rmi.* ;  
public class HelloServer {  
    public static void main(String [] args) throws Exception {  
        MessageWriter server = new MessageWriterImpl() ;  
        Naming.rebind("messageservice", server) ;  
    }  
}
```

Este servidor crea un objeto remoto



# Ejemplo de Cliente

HelloClient.java

```
import java.rmi.* ;  
public class HelloClient {  
    public static void main(String [] args) throws Exception {  
        MessageWriter server = (MessageWriter) Naming.lookup(  
            "rmi://hendrix.cps.unizar.es/messageservice") ;  
        server.sendMessage("Hello, world") ;  
    }  
}
```



# Reflexión

---

A la vista de los ejemplo, ¿cuándo tenemos que referenciar los *stubs* y *skeletons* en el código?

Se manejan transparentemente

# Intercambio de Referencias Remotas

- Se pueden pasar como argumentos, y devolver como resultados, referencias a objetos remotos

```
public interface Printer extends Remote {  
    void print(String document) throws RemoteException ;  
}
```

Ejemplo: *callbacks*

objeto  
remoto

```
public interface PrinterHub extends Remote {  
    Printer getPrinter(int dpi, boolean isColor)  
        throws RemoteException ;  
}
```



# Reflexión

---

¿Qué pasa si un objeto remoto recibe peticiones simultáneas de varios clientes?

Política multithread



# Política *Multithread*

---

- Invocaciones de diferentes clientes pueden ejecutarse concurrentemente en diferentes *threads* (ej., un *thread* por petición)
- Por tanto, la clase que implementa la interfaz remota tiene que ser *thread-safe*
- Uso del modificador *synchronized*
  - Serializa las peticiones: cuidado con los *deadlocks*



# Resumen Clases Java (I)

---

- *java.rmi.Remote*
- *java.rmi.RemoteException*
- *java.rmi.server.UnicastRemoteObject*
- *java.rmi.activation.Activatable*
- *java.rmi.server.RemoteRef*





# Resumen Clases Java (II)

---

- *java.rmi.Naming*
- *java.rmi.registry.Registry*
- *java.rmi.registry.LocateRegistry*
- *java.rmi.UnmarshalException*
- *java.rmi.RMI SecurityManager*



# Resumen Clases Java (III)

---

- *java.rmi.server.RMIConnectionFactory*
- *java.rmi.server.RMIServerConnectionFactory*
- *java.rmi.server.RMIClientConnectionFactory*



# Agradecimientos

---

Algunas de las transparencias e ideas de esta presentación están inspiradas o han sido adaptadas de trabajos de:

Celsina Bignoli (<http://www.smccd.edu/accounts/bignolic/>)

Bryan Carpenter (<http://www.hpjava.org/courses/arl>)



# Referencias

---

- RMI Trail (The Java Tutorial, Sun):  
<http://java.sun.com/docs/books/tutorial/rmi/index.html>
- Remote Method Invocation Home (Sun):  
<http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>
- RMI Specification (Sun):  
<http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html>



Fin

---

Gracias por vuestra atención