

# Programación Distribuida con Java-RMI

## Tema 7 - Programación Concurrente y de Tiempo Real

<sup>1</sup>Departamento de Ingeniería Informática  
Universidad de Cádiz

PCTR, 2021

1. Conceptos de Programación Distribuida
2. Remote Method Invocation (RMI) en Java
3. El Nivel de Resguardos (Stubs)
4. La Responsable de que Todo Funcione: la interfaz
5. Generación de Resguardos
6. Arquitectura Completa de una Aplicación
7. Evoluciones Reseñables

# Conceptos de Programación Distribuida

- ▶ No existe memoria común
- ▶ Comunicaciones
- ▶ No existe un estado global del sistema
- ▶ Grado de Transparencia
- ▶ Escalables
- ▶ Reconfigurables

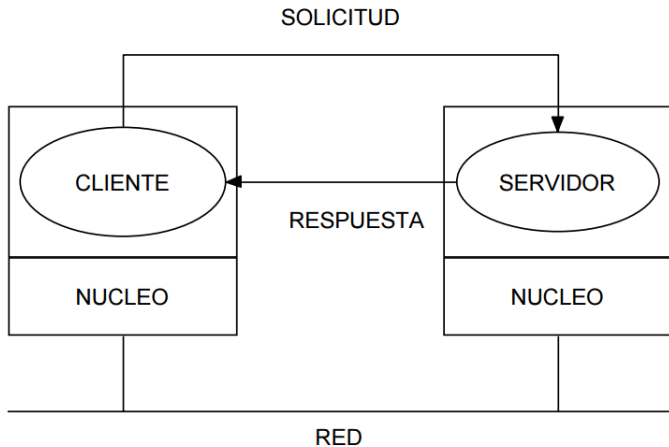
# Modelos de Programación Distribuida

- ▶ Modelo de paso de mensajes
  - ▶ Operaciones send y receive
- ▶ Modelo RPC (Remote Procedure Call)
  - ▶ *Stubs* de cliente y servidor
  - ▶ Visión en Java es RMI, incorporando orientación a objetos
- ▶ Modelos de Objetos Distribuidos
  - ▶ DCOM de Microsoft
  - ▶ Jini de SUN
  - ▶ CORBA de OMG

# Modelo de Paso de Mensajes

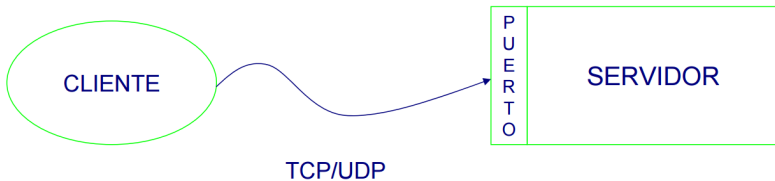
- ▶ Mecanismo para comunicar y sincronizar entidades concurrentes que no pueden o no quieren compartir memoria
- ▶ Llamadas *send* y *receive*
- ▶ Tipología de las llamadas:
  - ▶ Bloqueadas - No bloqueadas
  - ▶ Síncrona - Asíncronas
  - ▶ Fiables - No Fiables
- ▶ En Java:
  - ▶ Sockets: Clases `Socket` y `ServerSocket`
  - ▶ Canales: CTJ (Communicating Threads for Java), JCSP (Java Communicating Sequential Processes)
  - ▶ Implementaciones de MPI (por ejemplo, MPJ-Express)

# Modelo General



# Puertos (Breve Repaso) I

- El protocolo TCP (y también UDP) utilizan los puertos para hacer llegar los datos de entrada a un proceso concreto que se ejecuta en una máquina



# Puertos (Breve Repaso) II

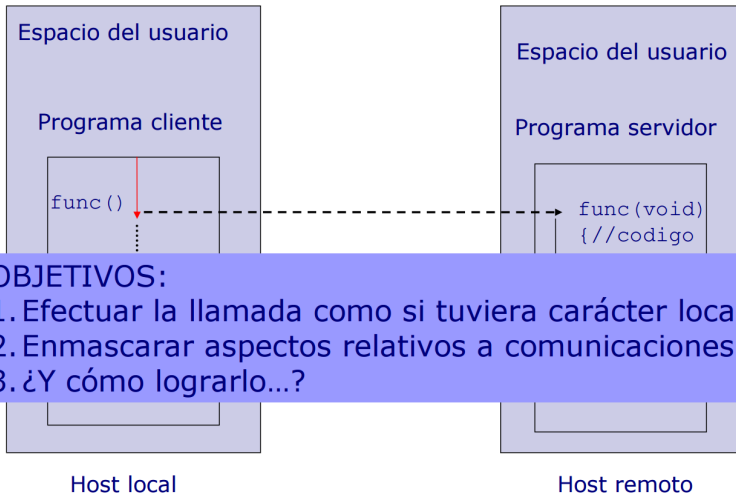




# Modelo Remote Procedure Call (RPC)

- ▶ Mayor nivel de abstracción
- ▶ Llamadas a procedimiento local o remoto indistintamente
- ▶ Necesita de resguardos stubs/skeletons (¡OCULTAMIENTO!)
- ▶ En Unix:
  - ▶ Biblioteca `rpc.h`
  - ▶ Representación estándar de red: XDR
  - ▶ Automatización parcial: especificación-compilador `rpcgen` para generación de resguardos
- ▶ En Java:
  - ▶ Objetos remotos. Serialización
  - ▶ Paquete `java.rmi`
  - ▶ Automatización parcial: interfaz-compilador `rmic` para generación de resguardos
  - ▶ En versiones recientes del JDK, son generados de forma automática

# Llamando a un Procedimiento Remoto



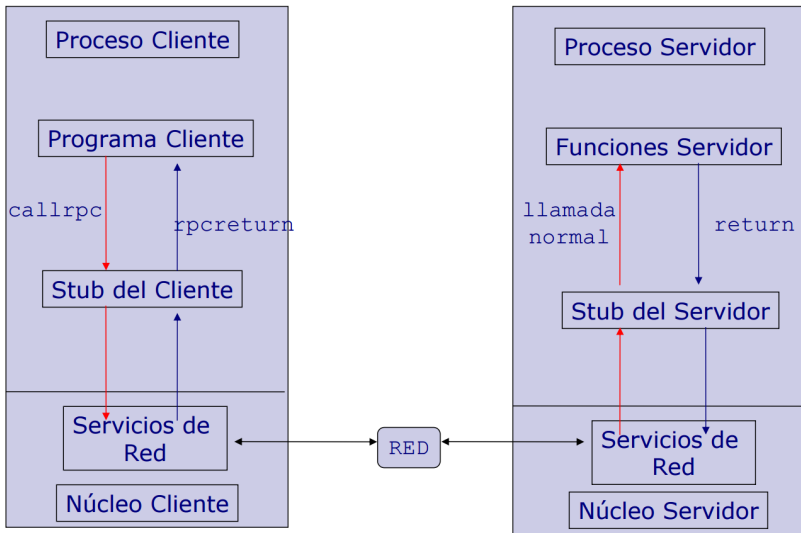
## OBJETIVOS:

1. Efectuar la llamada como si tuviera carácter local
2. Enmascarar aspectos relativos a comunicaciones
3. ¿Y cómo lograrlo...?

# Introducción del Nivel de Stubs

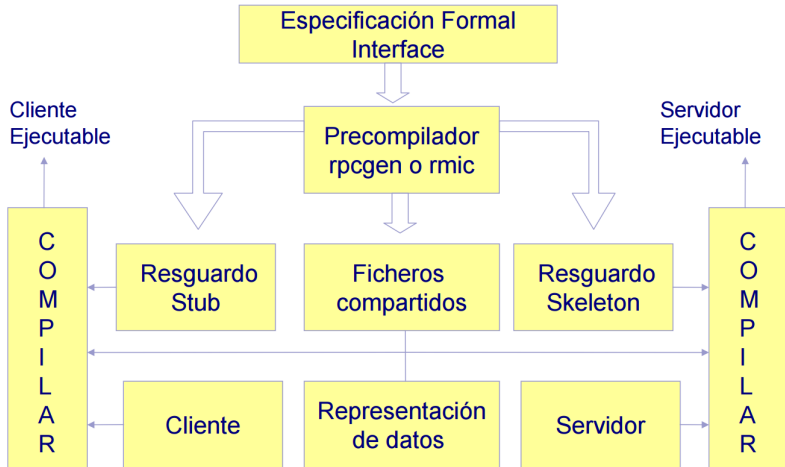
- ▶ Efectúan el marshalling/unmarshalling de parámetros.
- ▶ Bloquean al cliente a la espera del resultado.
- ▶ Transparencia de ejecución remota
- ▶ Interface con el nivel de red
  - ▶ TCP
  - ▶ UDP

# Esquema RPC con Stubs

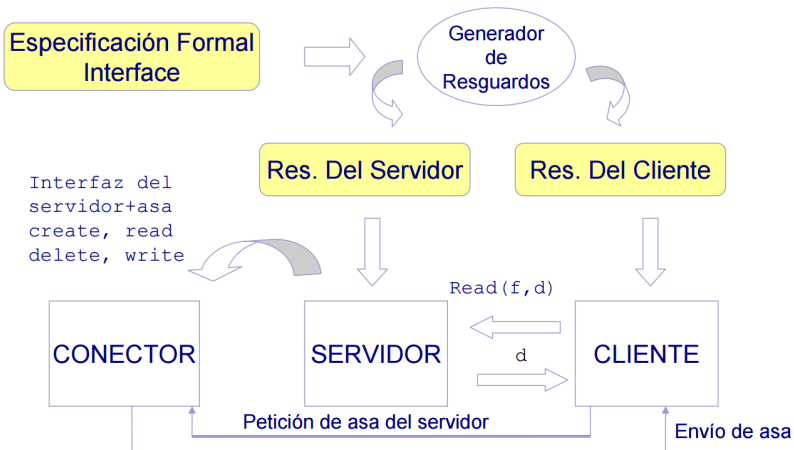


- ▶ De forma manual
- ▶ De forma automática:
  - ▶ Especificación formal de interfaz
  - ▶ Software específico
  - ▶ rpcgen (C-UNIX) / rmic (Java)

# Generación Automática de Stubs



# Resolución de Llamadas Remotas



- Especificación de la interfaz del servidor remoto en un fichero de especificación .x

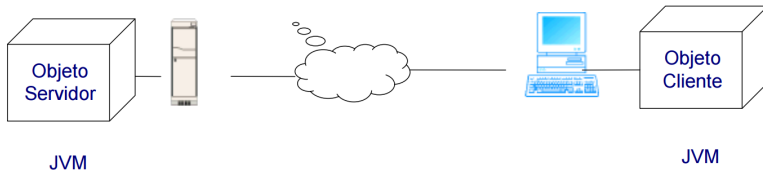
```
/*rand.x*/  
program RAND_PROG{  
    version RAND_VER{  
        void inicia_aleatorio(long)=1;  
        double obtener_aleat(void)=2;  
    }=1;  
}=0x3111111;
```

- Generación automática de resguardos: `rpcgen rand.x`
- Distribuir y compilar ficheros entre las máquinas implicadas
- Necesario utilizar representación XDR. Biblioteca `xdr.h`



# RMI (Remote Method Invocation) en Java

- ▶ Permite disponer de objetos distribuidos utilizando Java
- ▶ Un objeto distribuido se ejecuta en una JVM diferente o remota
- ▶ **Objetivo:** lograr una referencia al objeto remoto que permita utilizarlo como si el objeto se estuviera ejecutando sobre la JVM local
- ▶ Es similar a RPC, si bien el nivel de abstracción es más alto
- ▶ Se puede generalizar a otros lenguajes utilizando JNI
- ▶ RMI pasa los parámetros y valores de retorno utilizando serialización de objetos.



# RPC vs. RMI

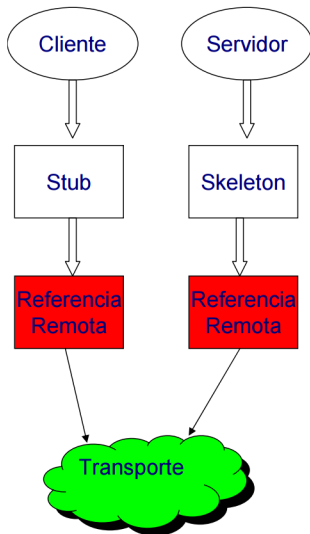
RPC	RMI
Carácter y Estructura de diseño procedimental	Carácter y Estructura de diseño orientada a objetos
Es dependiente del lenguaje	Es dependiente del lenguaje
Utiliza la representación externa de datos XDR	Utiliza la serialización de objetos en Java
El uso de punteros requiere el manejo explícito de los mismos	El uso de referencias a objetos locales y remotos es automático
NO hay movilidad de código	El código es móvil, mediante el uso de bytecodes.

# Llamadas Locales Vs. Llamadas Remotas

- ▶ Un objeto remoto es aquél que se ejecuta en una JVM diferente, situada potencialmente en un host distinto.
- ▶ RMI es la acción de invocar a un método de la interfaz de un objeto remoto.

```
1 //ejemplo de llamada a metodo local
2 int dato;
3 dato = Suma (x,y);
4 //ejemplo de llamada a metodo remoto (no completo)
5 IEjemploRMI1 ORemoto =
6 (IEjemploRMI1)Naming.lookup("//sargo.uca.es:2005/EjemploRMI1");
7 ORemoto.Suma(x,y);
```

# Arquitectura RMI I



- ▶ El servidor debe extender `RemoteObject`
- ▶ El servidor debe implementar una interfaz diseñada previamente
- ▶ El servidor debe tener como mínimo un constructor (nulo) que lanzará la excepción `RemoteException`
- ▶ El método `main` del servidor crea los objetos remotos
- ▶ El compilador de RMI (`rmic`) genera el stub y el skeleton. En desuso (`deprecated`)
- ▶ Los clientes de objetos remotos se comunican vía interfaces remotas
- ▶ Paso de parámetros en RMI:
- ▶ Los tipos primitivos se pasan por valor.

- ▶ Los parámetros referidos a objetos son pasados<sup>1</sup> por referencia o por valor, dependiendo de si la clase a que pertenecen los métodos que los emplean implementan la interfaz Remote o Serializable.

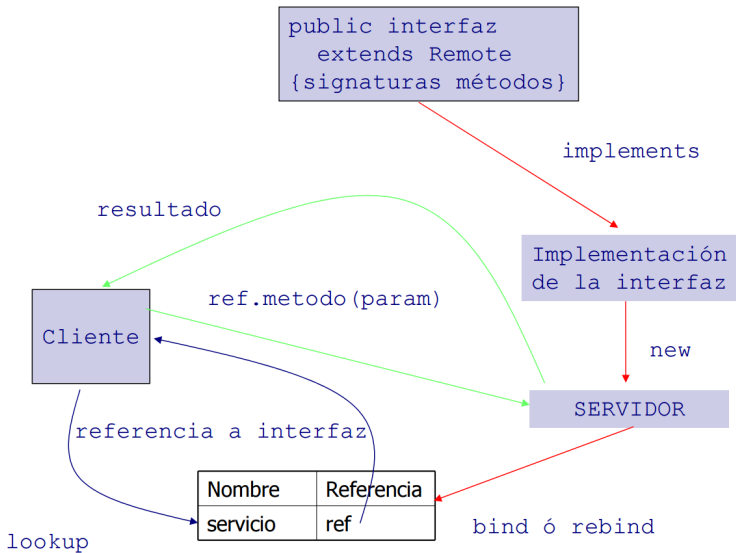
---

<sup>1</sup>Capel, M. Sistemas Concurrentes y Distribuidos, Copicentro, 2012.

# La Clase java.rmi.Naming

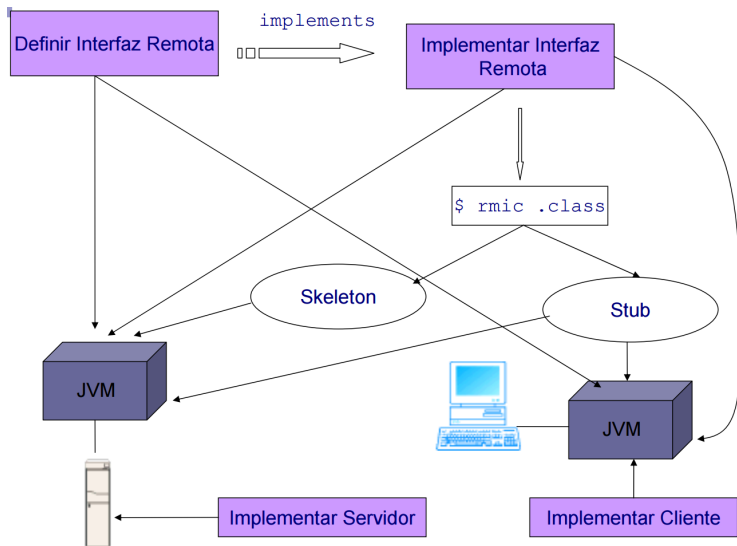
```
1 public static void bind(String name, Remote obj)
2 public static String[] list(String name)
3 public static Remote lookup(String name)
4 public static void rebind(String name, Remote obj)
5 public static void unbind(String name)
```

# ¿Cómo Lograrlo? I





# ¿Cómo Lograrlo? II



# Fases de Diseño de RMI (1 - Interface)

- ▶ Escribir el fichero de la interfaz remota.
- ▶ Debe ser `public` y extender a `Remote`.
- ▶ Declara todos los métodos que el servidor remoto ofrece, pero NO los implementa.
- ▶ Se indica nombre, parámetros y tipo de retorno.
- ▶ Todos los métodos de la interfaz remota lanzan obligatoriamente la excepción `RemoteException`.
- ▶ El propósito de la interface es ocultar la implementación de los aspectos relativos a los métodos remotos.

# Fases de Diseño de RMI (1 - Ejemplo de Interfaz)

Código: codigos\_t7/IEjemploRMI1.java

```
1  //se importan las clases del paquete rmi
2  import java.rmi.*;
3
4  //toda interface remota debe extender la clase Remote
5  public interface IEjemploRMI1 extends Remote{
6
7      //todo metodo de la interfaz remota debe declarar la
          excepcion RemoteException
8      int Suma(int x, int y)throws RemoteException;
9      int Resta(int x, int y)throws RemoteException;
10     int Producto(int x, int y)throws RemoteException;
11     float Cociente(int x, int y)throws RemoteException;
12 }
```

## Fases de Diseño de RMI (2 - Implementación de Interfaz-Escritura del Servidor)

- ▶ Se implementa la interfaz remota en un clase que extiende a `UnicastRemoteObject`.
- ▶ Se escribe código de servidor que cree objetos potencialmente exportables.
- ▶ Los objetos a exportar son registrados en un servicio de DNS mediante el método `Naming.bind`.
- ▶ El servidor queda a la espera de recibir peticiones de servicio desde los clientes.

# Fases de Diseño de RMI (2-Ejemplo de Implementación de Interfaz y Escritura de Servidor I

Código: codigos\_t7/EjemploRMI1.java

```
1 //se importan los paquetes necesarios
2 import java.rmi.*;
3 import java.rmi.server.*;
4 import java.rmi.registry.*;
5 import java.net.*;
6
7 public class EjemploRMI1
8 extends UnicastRemoteObject //el servidor debe siempre extender
    esta clase
9 implements IEjemploRMI1 //el servidor debe siempre implementar
    la interfaz
10 //remota definida con caracter previo
11 {
12     public int Suma(int x, int y)
13     throws RemoteException {
14         return x + y;
15     }
```

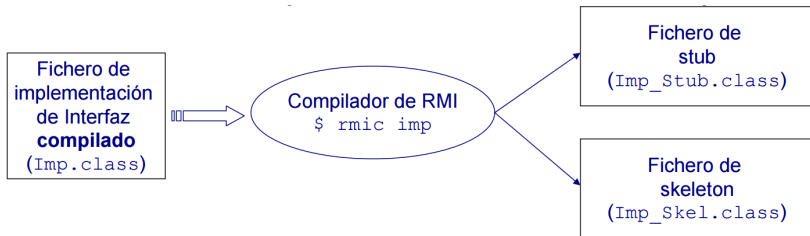
# Fases de Diseño de RMI (2-Ejemplo de Implementación de Interfaz y Escritura de Servidor II)

```
16
17 public int Resta(int x, int y)
18     throws RemoteException {
19     return x - y;
20 }
21
22 public int Producto(int x, int y)
23     throws RemoteException {
24     return x * y;
25 }
26
27 public float Cociente(int x, int y)
28     throws RemoteException {
29     if (y == 0) return 1f;
30     else return x / y;
31 }
32
33 //es necesario que haya un constructor (nulo) como minimo, ya
    que debe
34 //lanzar RemoteException
```

## Fases de Diseño de RMI (2-Ejemplo de Implementación de Interfaz y Escritura de Servidor III)

```
35  public EjemploRMI1() throws RemoteException{}
36
37  //el metodo main siguiente realiza el registro del servicio
38
39  public static void main(String[] args) throws Exception {
40
41      //Se crea el objeto remoto. Podriamos crear mas si interesa.
42      IEjemploRMI1 ORemoto = new EjemploRMI1();
43
44      //Se registra el servicio
45      Naming.bind("Servidor", ORemoto);
46
47      System.out.println("Servidor Remoto Preparado");
48  }
49 }
```

## Fases de Diseño RMI (3 - Generación de STUB y SKELETON) I



```
rmic -vcompat EjemploRMI1
```

Nota: en versiones recientes del JDK esta fase no es necesaria.



## Fases de Diseño RMI (3 - Generación de STUB y SKELETON) II

- ▶ Es necesario que en la máquina remota donde se aloje el servidor se sitúen también los ficheros de stub y skeleton. (Cambios a partir de 1.5)
- ▶ Con todo ello disponible, se lanza el servidor llamando a JVM del host remoto, previo registro en un DNS.
- ▶ El JDK proporciona un DNS básico mediante el binario `rmiregistry`. Otros DNS alternativos son posibles.
- ▶ En nuestro caso, el servidor remoto se activó en `sargo.uca.es` sobre el puerto 2005.

```
java EjemploRMI1 &
```

## Fases de Diseño RMI (4 - Registro)

- ▶ Método `Naming.bind("Servidor",ORemoto);` para registrar el objeto remoto creado requiere que el servidor de nombres esté activo.
- ▶ Dicho servidor se activa con `start rmiregistry` en Win32
- ▶ Dicho servidor se activa con `rmiregistry &` en Unix.
- ▶ Se puede indicar como parámetro el puerto que escucha.
- ▶ Si no se indica, por defecto es el puerto 1099.
- ▶ El parámetro puede ser el nombre de un host como en `Naming.bind("//sargo.uca.es:2005/Servidor",ORemoto);`

## Fases de Diseño RMI (5 - Cliente) I

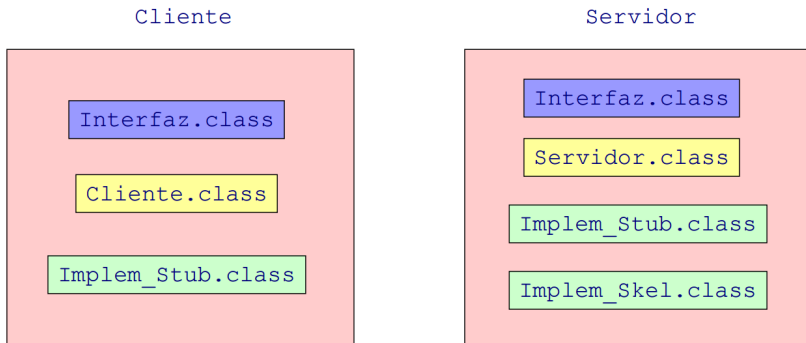
- ▶ El objeto cliente procede siempre creando un objeto de interfaz remota.
- ▶ Posteriormente efectúa una llamada al método `Naming.lookup` cuyo parámetro es el nombre y puerto del host remoto junto con el nombre del servidor.
- ▶ `Naming.lookup` devuelve una referencia que se convierte a una referencia a la interfaz remota.
- ▶ A partir de aquí, a través de esa referencia el programador puede invocar todos los métodos de esa interfaz remota como si fueran referencias a objetos en la JVM local (aunque la semántica de la resolución es profundamente diferente).

# Fases de Diseño RMI (5 - Cliente) II

Código: codigos\_t7/ClienteEjemploRMI1.java

```
1  import java.rmi.*;
2  import java.rmi.registry.*;
3
4  public class ClienteEjemploRMI1 {
5      public static void main(String[] args)
6          throws Exception {
7          int a = 10;
8          int b = -10;
9
10         IEjemploRMI1 RefObRemoto =
11             (IEjemploRMI1) Naming.lookup("//localhost/Servidor");
12
13         System.out.println(RefObRemoto.Suma(a, b));
14         System.out.println(RefObRemoto.Resta(a, b));
15         System.out.println(RefObRemoto.Producto(a, b));
16         System.out.println(RefObRemoto.Cociente(a, b));
17
18     }
19 }
```

# Distribución de Archivos



**Figura:** A partir de la JDK 1.2 las clases SKELETON no son necesarias

# Ejercicio I

- ▶ Descargue los ficheros IEjemploRMI1.java, EjemploRMI1.java y ClienteEjemploRMI1.java
- ▶ Distribuya ahora la aplicación con un compañero
- ▶ Escriba una interfaz de operaciones para números complejos, utilizando tipos primitivos, llamada IComplejo.java
- ▶ Impleméntela en Complejo.java
- ▶ Escriba código de servidor y otro de cliente que hagan uso de la misma. Guárdelos en ServerComplejo.java y ClientComplejo.java
- ▶ Distribuya nuevamente la aplicación
- ▶ Acuerden una interfaz llamada interfaz\_grupo\_x.java
- ▶ Un miembro del grupo escribirá el cliente\_grupo.java y el otro el servidor\_grupo\_x.java de manera independiente. Una vez hecho, lancen la arquitectura rmi.

- ▶ A partir del JDK 1.2 la implementación de RMI no necesita skeletons (con ese nombre tenían poco futuro). El stub estará en ambos lados.
- ▶ A partir del JDK 1.5 se incorpora Generación Dinámica de Resguardos.
- ▶ Recompile los ejemplos anteriores sin el flag `-vcompat`.

- ▶ Serialización de Objetos
- ▶ *Callback* de Cliente
- ▶ Descarga Dinámica de Clases



- ▶ Serializar un objeto es convertirlo en una cadena de bits, posteriormente restaurada en el objeto original.
- ▶ Es útil para enviar objetos relativamente complejos en RMI.
- ▶ Tipos primitivos se serializan automáticamente.
- ▶ Clases contenedoras (que implementan `Serializable`) también.
- ▶ Objetos complejos deben implementar la interfaz `Serializable` (no tiene métodos, actúa como un flag) para poder ser serializados
- ▶ Pero aún así, si tenemos objetos que referencian a objetos que referencian a objetos... la cosa se complica, y puede llegar a ser necesario escribir un código *ad-hoc* para efectuar la serialización.

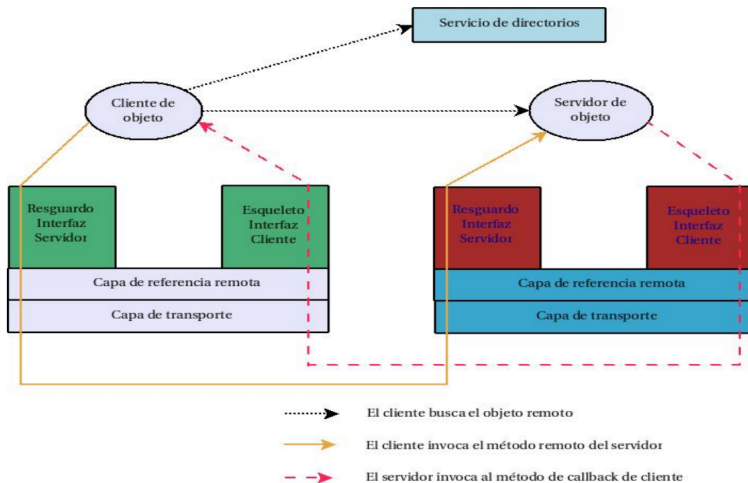
- ▶ Si el servidor de RMI debe notificar eventos a clientes (por ejemplo esto es necesario en algo tan simple como un *chat*, la arquitectura RMI básica se torna inapropiada
- ▶ La alternativa estándar es el sondeo continuo (*polling*), donde cliente:

```
1  Iservidor RefRemota= (IServidor)
2      Naming.lookup (URL);
3  while (!(Ref.Remota.Evento.Esperado())){}
```

# Características del CallBack

- ▶ Clientes interesados **se registran** en un objeto servidor para que les sea notificado un evento
- ▶ Cuando el evento se produce, el objeto servidor notifica al cliente su ocurrencia
- ▶ ¿Qué necesitamos para que funcione?

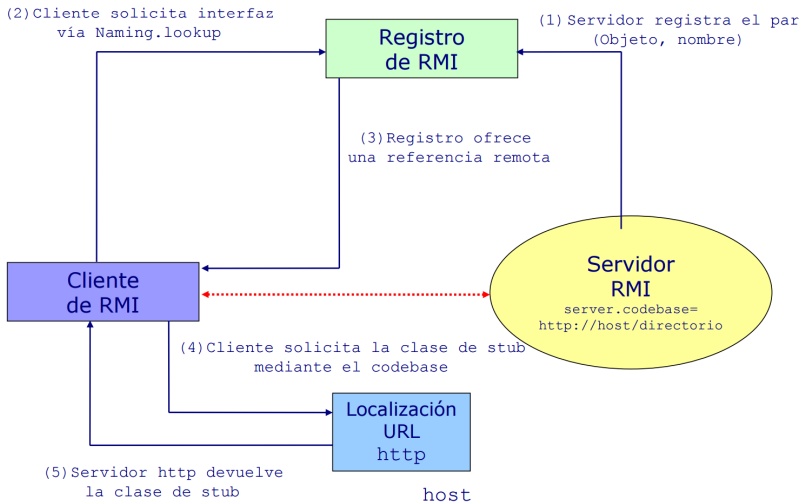
# Arquitectura RMI para Callback



- ▶ Hay que duplicar la arquitectura
- ▶ Se requieren dos interfaces
- ▶ Habrá dos niveles de *stubs*
  - ▶ Cliente (*stub+skel*) y servidor (*stub+skel*)
- ▶ Es necesario proveer en el servidor medios de almacenamiento para que los clientes registren sus peticiones de callback.

- ▶ Permite cambios libres en el servidor
- ▶ Elimina la necesidad de distribuir (resguardos) si los hay y nuevas clases
- ▶ Dinámicamente, el cliente descarga todas las clases que necesita para desarrollar una RMI válida
- ▶ Mediante HTTP o incluso FTP

# Descarga Dinámica de Clases II



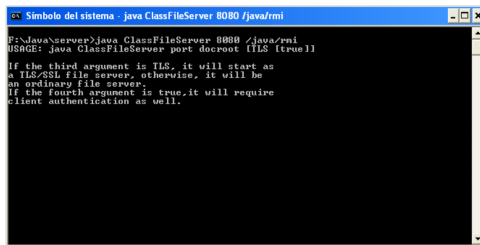
# Servidor HTTP para Descarga Dinámica I

- ▶ Mini-servidor de HTTP de Sun
  - ▶ Clases ClassServer y ClassFileServer
  - ▶ Se compilan conjuntamente

```
java ClassFileServer 8080 /rmi/clases
```
- ▶ Servidor HTTP estándar
  - ▶ Apache
  - ▶ Otros



# Servidor HTTP para Descarga Dinámica II



```
Símbolo del sistema - java ClassFileServer 8080 /java/rmi
F:\Java\server>java ClassFileServer 8080 /java/rmi
USAGE: java ClassFileServer port docroot [TLS {true}]

If the third argument is TLS, it will start as
a TLS/SSL file server, otherwise, it will be
an ordinary file server.
If the fourth argument is true, it will require
client authentication as well.
```

Figura: Ejemplo de servidor básico de Sun activo en Windows

- ▶ Colocar resguardos y clases para carga dinámica en /java/rmi
- ▶ Ajustar la propiedad `java.rmi.server.codebase`

# Bibliografía

-  Capel, M.  
*Sistemas Concurrentes y Distribuidos, Tomo I*  
Copicentro, 2012
-  Eckel, B.  
*Thinking in Java*  
Prentice Hall, 2006
-  Hilderink et al.  
*Communicating Threads for Java*  
Draft
-  Vinoski, S.  
*CORBA: Integrating Diverse...*  
IEEE Communications, 1997
-  Grosso, W.  
*Java RMI*  
O'Reilly, 2001