Sistemas de Información

Tecnologías de objetos distribuidos: RMI

Agradecimientos: Marisol García Valls, Jesús Villamor Lugo, Simon Pickin de IT/UCIIIM

mcfp@it.uc3m.es

RMI (*Remote Method Invocation*)

Concepto ¿Qué es?

- RMI (Remote Method Invocation). Procedimiento de invocación a métodos remotos
- Permite a un objeto que se está ejecutando en una MV invocar métodos de otro que esté en una MV distinta
- API de java que facilita la creación de aplicaciones de objetos distribuidos proporcionando:
  - Mecanismos de localización,
  - Facilidades de comunicación
  - Una semántica para permitir la invocación de métodos remotos

RMI (***R**emote **M**ethod **I**nvocation*)

Objetivos ¿Para qué sirve?

- Permitir invocación de métodos de un objeto por objetos que residen en diferentes máquinas virtuales (en particular, a través de la red)
  - permitir invocación de métodos remotos por Applets
- Integrar el modelo de objetos distribuidos al lenguaje Java de modo natural, preservando en lo posible la semántica de objetos en Java
  - distinción entre objetos locales y remotos
  - diferentes semánticas en las referencias a objetos remotos:
    - no persistentes (vivas), persistentes, de activación lenta
- Preservar la seguridad de tipos (*type safety*) dada por el ambiente de ejecución Java
- Mantener la seguridad del ambiente dada por los *Security Managers*, en particular, en presencia de carga dinámica de clases
- Facilitar el desarrollo de aplicaciones distribuidas

RMI (***R**emote **M**ethod **I**nvocation*)

Hitos Especificaciones

- **RMI en JDK-1.1**
  - Introducción de las bibliotecas **`java.rmi.*`**
  - Comunicación por *Java Remote Method Protocol* (JRMP)
- **RMI en J2SE SDK 1.2**
  - Referencias de objeto persistentes y objetos activables
  - No hace falta generar skeletons (reflection + *skeletons* genéricos)
- **RMI en J2SE SDK 1.3**
  - Opción de comunicación por IIOP (Internet Inter-ORB Protocol de CORBA)
  - Uso de un fichero de política de seguridad obligatorio con la activación
- **RMI en J2SE SDK 1.4**
  - Mejoras en la *serialization* (secuenciación)
- **RMI en J2SE SDK 1.5 = J2SE SDK 5.0**
  - Introducción de un tipo de invocación dinámica
    - hace uso de reflection + *stubs* genéricos
    - este mecanismo obvia la compilación con **`rmic`**

RMI (*Remote Method Invocation*)

¿Cómo funciona?

- **El servidor**
  - Crea objetos remotos
  - Hace accesibles refs a objetos remotos
  - Espera a que los clientes invoquen a estos objetos remotos o a sus métodos
- **El cliente**
  - Obtiene una referencia de uno o más objetos remotos en el servidor
  - Invoca a sus métodos

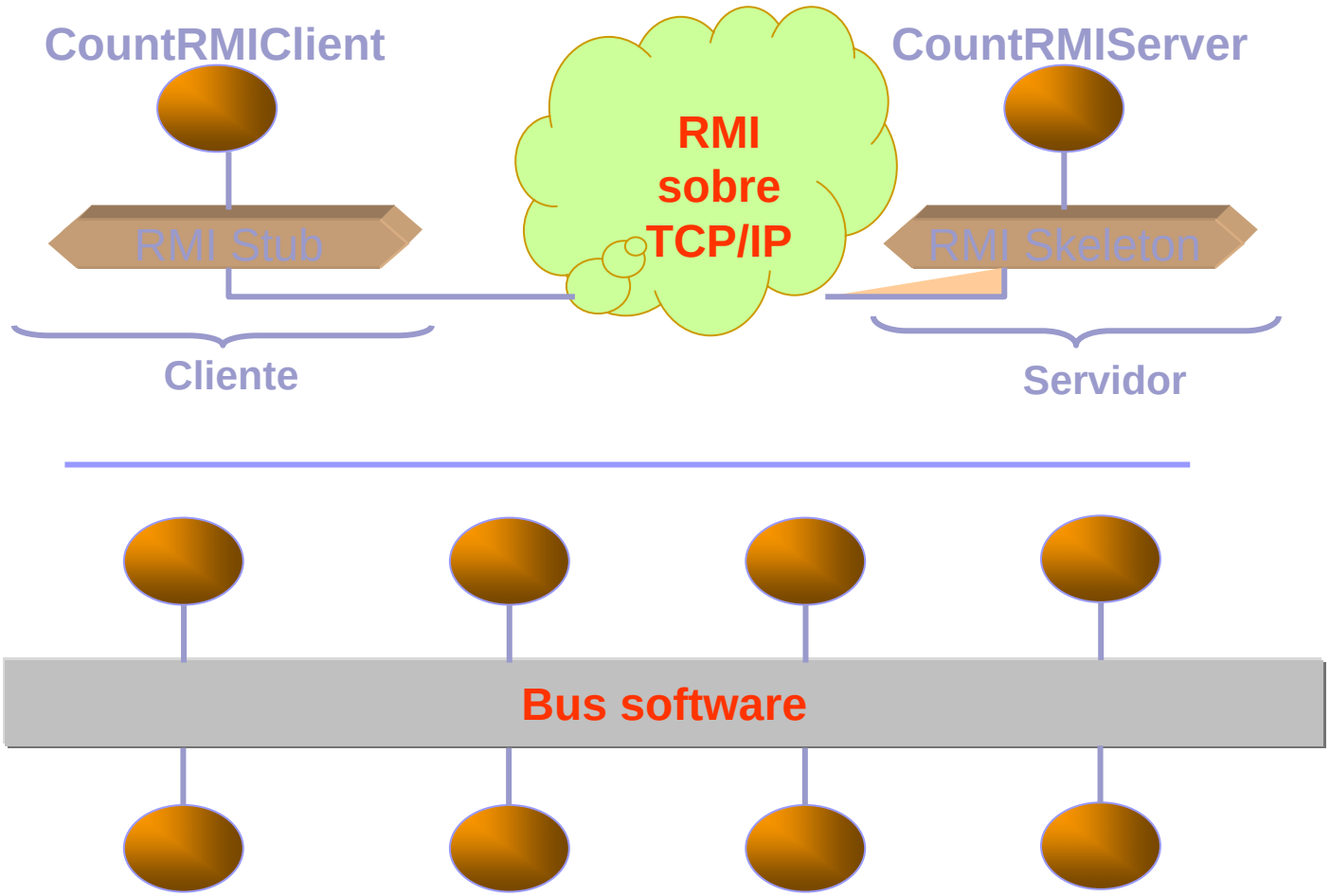
RMI (***R**emote **M**ethod **I**nvocation*)

¿Cómo funciona?

- **RMI proporciona al cliente y servidor**
  - Mecanismos de localización (obtención de refs)
    - Registrar objetos remotos con rmiregistry
    - Pasar y devolver referencias a objetos remotos
  - Mecanismos de comunicación
    - Transparentes para el programador
    - Semántica similar a una llamada estándar a método
  - Mecanismos de carga dinámica de clases
    - Para objetos que se pasan entre C y S bien por parámetro o como tipo de retorno

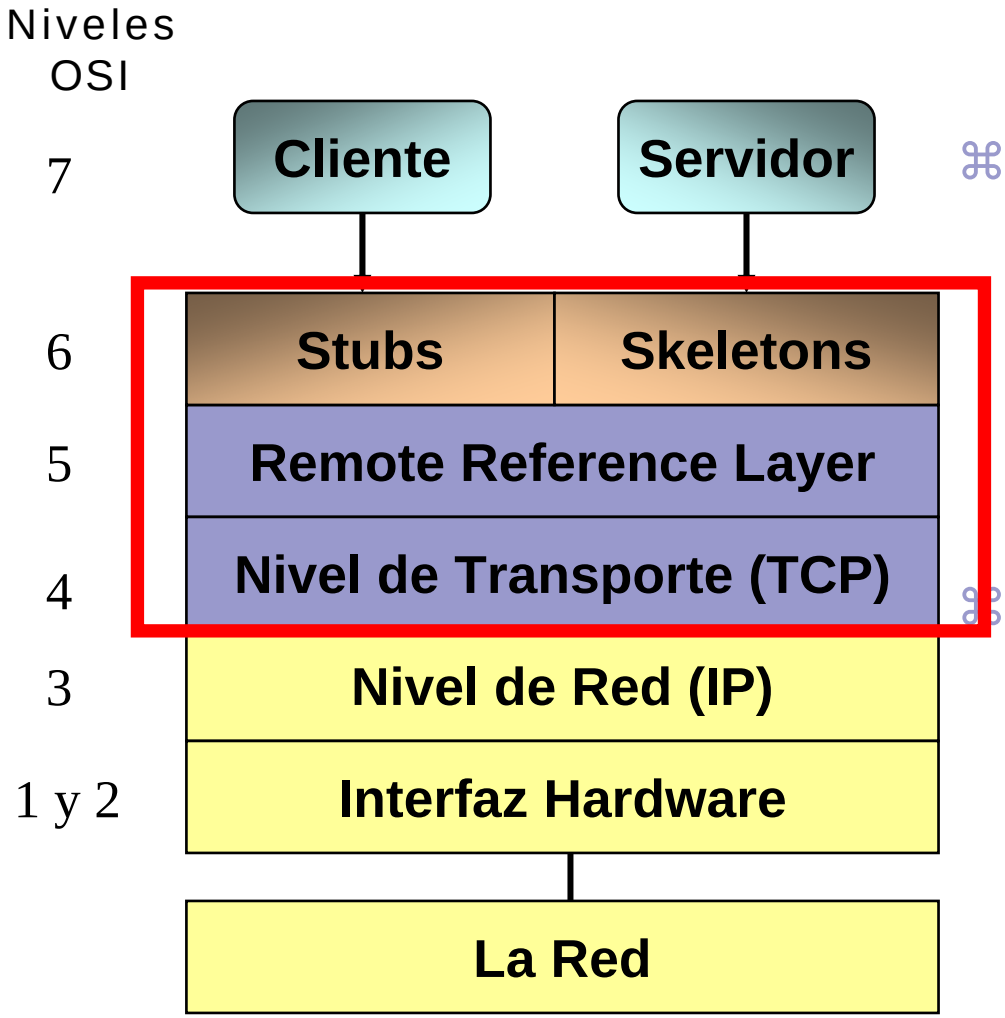
RMI (*Remote Method Invocation*)

¿Cómo funciona?



RMI (*Remote Method Invocation*)

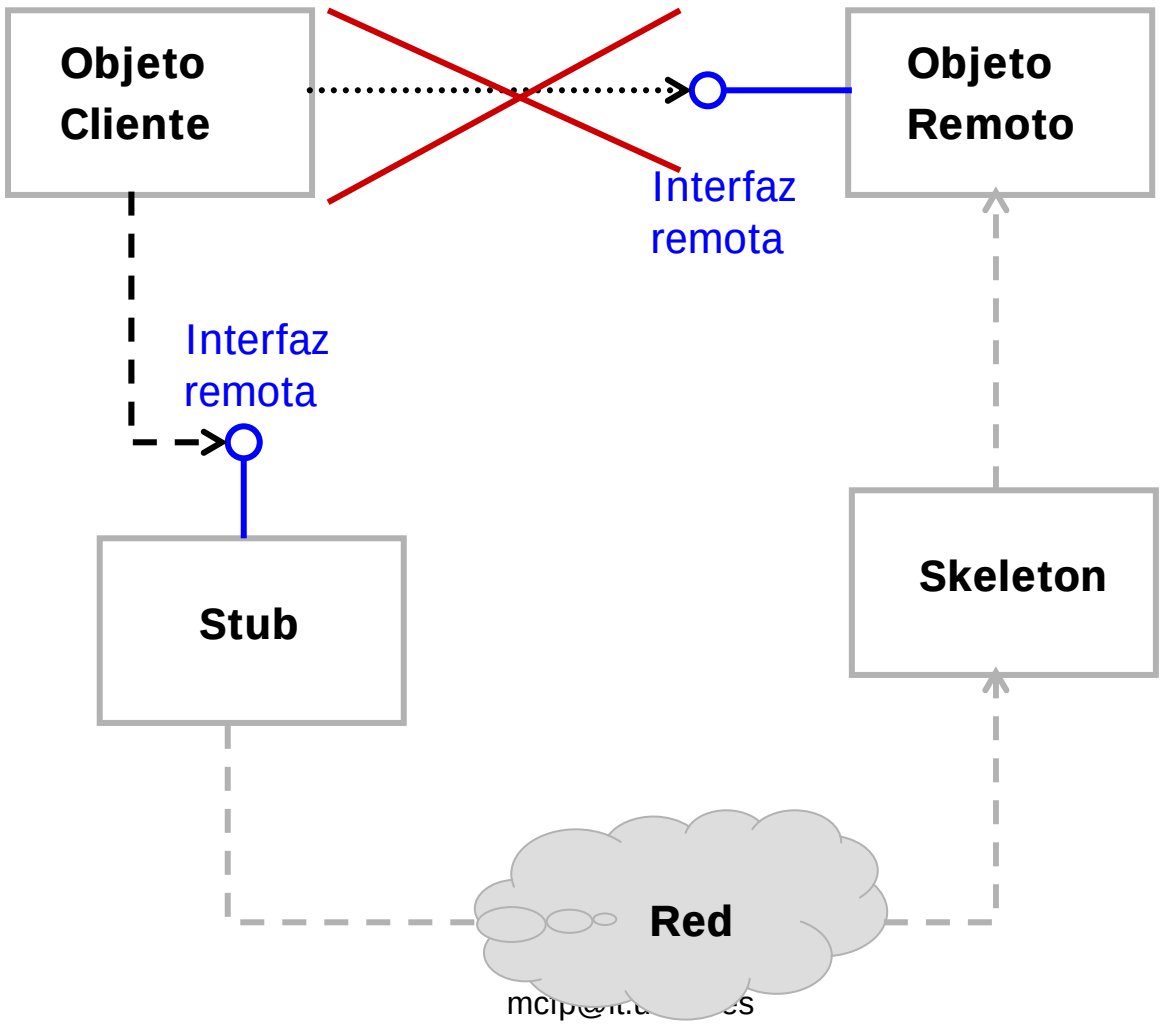
¿Cómo funciona?: Arquitectura



- El Sistema RMI mismo está formado por 3 niveles
  - El nivel de los *Stubs* y *Skeletons*
  - La *Remote Reference Layer*
  - El nivel de Transporte
    - hoy por hoy basado en TCP
- El cliente y el servidor desarrollan sus aplicaciones en paralelo
  - El cliente invoca objetos remotos a través de los *stubs*
  - Los *skeletons* permiten hacer accesibles los objetos servidores al cliente

RMI: ¿Cómo funciona?

Stubs y Skeletons



RMI: ¿Cómo funciona?

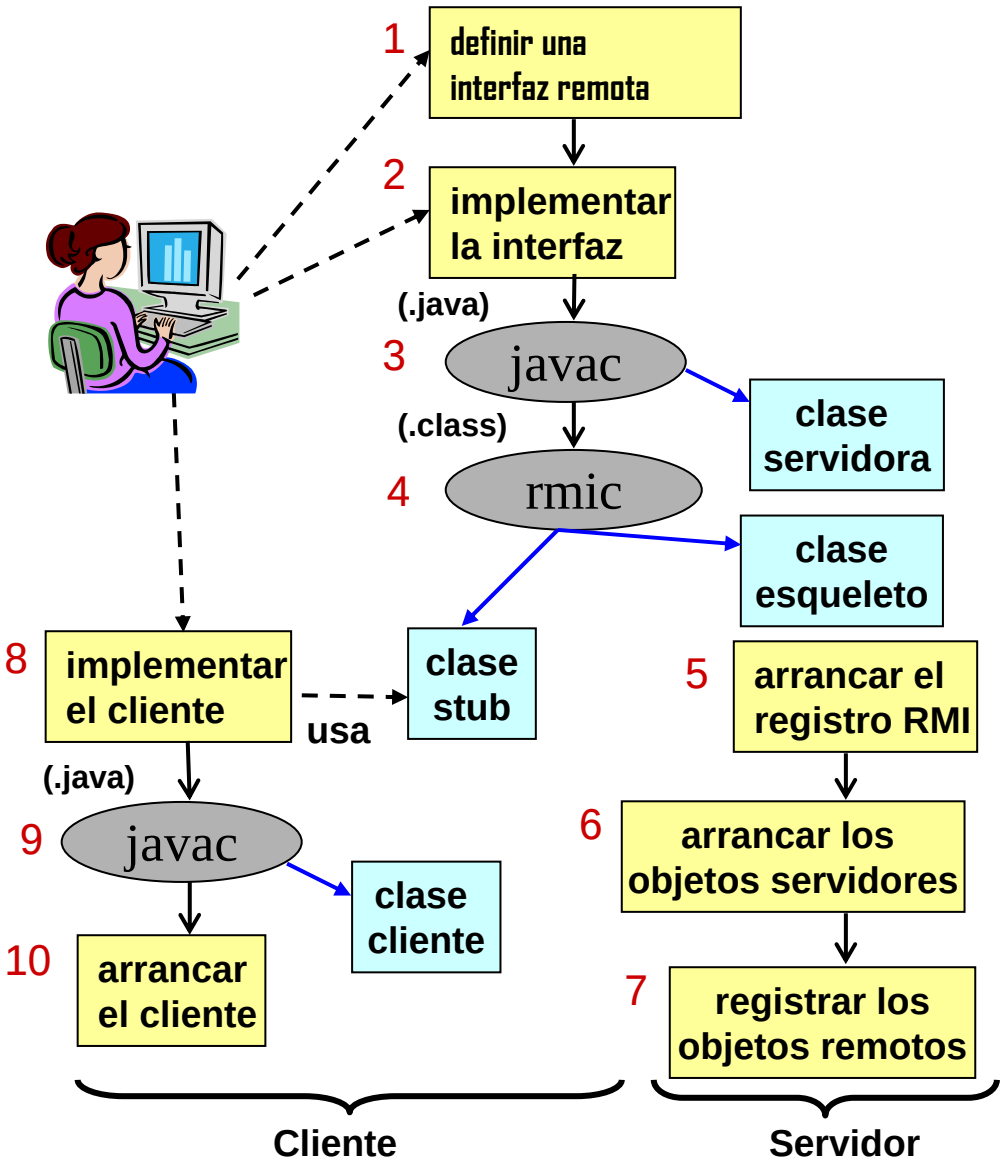
Interfaces remotas y objetos remotos

- Una interfaz remota declara un conjunto de operaciones que podrán invocarse desde otras JVM.
  - debe extender **`java.rmi.Remote`**
  - sus métodos deben declarar que lanzan **`java.rmi.RemoteException`**
- Para que los métodos de una interfaz remota estén disponibles para ser invocadas desde otras JVM ha de implementar la interfaz
- Un objeto remoto es un objeto que implementa una interfaz remota
- Un cliente en otra JVM interactúa con un objeto remoto vía una de sus interfaces remotas, nunca directamente
- Habitualmente, la clase de un objeto remoto extiende
  `java.rmi.server.UnicastRemoteObject`
  pero puede llamar a su método **`exportObject()`** directamente

RMI: ¿Cómo funciona?
Los ***stubs***

- Un *stub* de RMI es un *proxy*
  - es decir, un representante local de un objeto remoto
- Contiene la referencia al objeto remoto
- Permite la invocación de sus métodos como si fuera un objeto local.
- En concreto:
  - recibe las peticiones del llamante
  - realiza el *marshalling* (empaquetado de los parámetros),
  - envía la petición al objeto llamado
  - en el caso de que haya respuesta:
    - realiza el *unmarshalling*
    - devuelve el valor al llamante

RMI: ¿Cómo se usa?



Proceso de Desarrollo

1. Extender **`java.rmi.Remote`**
2. Implementar interfaz, extendiendo **`java.rmi.UnicastRemoteObject`** (o llamando a **`exportObject()`**)
3. Compilar impl. (**`.java`**) con **`javac`**
4. Compilar impl. (**`.class`**) con **`rmic`**
5. Arrancar el **`rmiregistry`**
6. Arrancar los objetos del servidor
7. Registrar los objetos remotos (llamando a métodos del paquete **`java.rmi.Naming`** para asociar un nombre con el objeto remoto)
8. Escribir el código cliente (llamando a métodos del paquete **`java.rmi.Naming`** para localizar el objeto remoto)
9. Compilar el código cliente
10. Arrancar el cliente

RMI Ejemplo 1

Paso 1: Definir interfaz remota

- Importar paquetes adecuados
  - `import java.rmi.Remote;`
  - `import java.rmi.RemoteException;`
- Extender la clase Remote
  - `public interface NombreInt extends Remote{}`
- Lanzar remoteException en todos los métodos
  - `public Tipo NombreMet() throws RemoteException;`

Ejemplo tomado de:

http://www.programacion.com/java/tutorial/rmi/

RMI Ejemplo 1

Paso 1: Definir un interfaz remoto

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
    Object executeTask(Task t) throws RemoteException;
}
```

RMI Ejemplo 1

Paso 1: Definir otros interfaces

- Detectar objetos que puedan viajar por tratarse de:
  - Parámetros de métodos remotos
  - Tipos de retorno de métodos remotos
- Deben extender el interfaz serializable

RMI Ejemplo 1

Paso 1: Definir otros interfaces

```
package compute;
import java.io.Serializable;
public interface Task extends Serializable {
    Object execute();
}
```

RMI Ejemplo 1

Paso 2: Implementar interfaz remota

- Extender la clase UnicastRemoteObject
- Implementar el interfaz remoto definido en el paso 1
- Crear main que realice las siguientes tareas
  - Crear controlador de seguridad
  - Crear objeto de la clase remota
  - Registrar objeto de la case remota

RMI Ejemplo 1

Paso 2: Implementar interfaz remoto

```
package engine;
import java.rmi.*; import java.rmi.server.*; import compute.*;
public class ComputeEngine extends UnicastRemoteObject implements Compute {
    public ComputeEngine() throws RemoteException { super();}
    public Object executeTask(Task t) { return t.execute(); }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }  String name = "//localhost/Compute";
         try {
            Compute engine = new ComputeEngine();
            Naming.rebind(name, engine);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
          System.err.println("ComputeEngine exception: " +  e.getMessage());
           e.printStackTrace();
        }
    }
}
```

RMI Ejemplo 1

Paso 2: Implementar interfaz remoto

```
package engine;
import java.rmi.*; import java.rmi.server.*; import compute.*;
public class ComputeEngine extends UnicastRemoteObject implements Compute{
    public ComputeEngine() throws RemoteException { super();}
    public Object executeTask(Task t) { return t.execute(); }
    public static void main(String[] args) {
```

public class ComputeEngine extends UnicastRemoteObject implements Compute{

```
        try {
           Compute engine = new ComputeEngine();
           Naming.rebind(name, engine);
           System.out.println("ComputeEngine bound");
        } catch (Exception e) {
          System.err.println("ComputeEngine exception: " +  e.getMessage());
          e.printStackTrace();
        }
    }
}
```

RMI Ejemplo 1

Paso 2: Implementar interfaz remoto

```
package engine;
import java.rmi.*; import java.rmi.server.*; import compute.*;
public class ComputeEngine extends UnicastRemoteObject implements Compute {
    public ComputeEngine() throws RemoteException { super();}
    public Object executeTask(Task t) { return t.execute(); }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }  String name = "//localhost/Compute";
         try {
            Compute engine = new ComputeEngine();
            e.printStackTrace();
        }
    }
}
```

```
if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
}
```

RMI Ejemplo 1

Paso 2: Implementar interfaz remoto

```
package engine;
import java.rmi.*; import java.rmi.server.*; import compute.*;
public class ComputeEngine extends UnicastRemoteObject implements Compute {
    public ComputeEngine() throws RemoteException { super();}
    public Object executeTask(Task t) { return t.execute(); }
    pu
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }  String name = "//localhost/Compute";
         try {
            Compute engine = new ComputeEngine();
            Naming.rebind(name, engine);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
          System.err.println("ComputeEngine exception: " +  e.getMessage());
           e.printStackTrace();
        }
    }
}
```

Compute engine = new ComputeEngine();

RMI Ejemplo 1

Paso 2: Implementar interfaz remoto

```
package engine;
import java.rmi.*; import java.rmi.server.*; import compute.*;
public                                                    Compute {
    pub
    pub
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        } String name = "//localhost/Compute";
         try {
            Compute engine = new ComputeEngine();
            Naming.rebind(name, engine);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
          System.err.println("ComputeEngine exception: " +  e.getMessage());
           e.printStackTrace();
        }
    }
}
```

```
String name = "//localhost/Compute";
Naming.rebind(name, engine);
```

RMI Ejemplo 1

Paso 3: Crear programa cliente

- Crear clases que invocan métodos remotos
  - Crear controlador de seguridad
  - Asignar nombre al objeto remoto
  - Invocar métodos del objeto remoto
- Crear resto de las clases

RMI Ejemplo 1

Paso 3: Crear programa cliente

```
package client;
import java.rmi.*; import java.math.*; import compute.*;
public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "//" + args[0] + "/Compute";
        } catch (Exception e) {
            System.err.println("ComputePi exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

```
if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
}
```

RMI Ejemplo 1

Paso 3: Crear programa cliente

```
package client;
import java.rmi.*; import java.math.*; import compute.*;
public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "//" + args[0] + "/Compute";
            Compute comp = (Compute) Naming.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = (BigDecimal) (comp.executeTask(task));
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

String name = "//" + args[0] + "/Compute";
Compute comp = (Compute) Naming.lookup(name);

RMI Ejemplo 1

Paso 3: Crear programa cliente

```
try{
    (…) comp.executeTask(task); (…)
}catch (RemoteException e) {
    (…)
}
```

```
pack
impo                                                    te.*;
pub

                                                        tyManager());
        }
        try {
            String name = "//" + args[0] + "/Compute";
            Compute comp = (Compute) Naming.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = (BigDecimal) (comp.executeTask(task));
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

RMI Ejemplo 1

Paso 3: Crear programa cliente (resto de clases)

```
package client;
import Compute.*; import java.math.*;
public class Pi implements Task {
    private static final BigDecimal ZERO = BigDecimal.valueOf(0);
    private static final BigDecimal  ONE = BigDecimal.valueOf(1);
    private static final BigDecimal FOUR = BigDecimal.valueOf(4);
    private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;
    private int digits;
    public Pi(int digits){ this.digits = digits;}
    public Object execute(){return computePi(digits);}
    public static BigDecimal computePi(int digits) {
        // calcula usando fórmula de Hachin
    }
   public static BigDecimal arctan(int inverseX,  int scale) {
        // Calcula en radianes el  arctan(x) = x - (x^3)/3 + (x^5)/5 -
        // - (x^7)/7 + (x^9)/9 ...
   }
}
```

RMI Ejemplo 1

Paso 4: Compilar la aplicación

- Compilar interfaces
  - javac NombreInterfaz.java
- Compilar la clase servidora
  - javac NombreClase.java
- Generar stubs y skeletons
  - rmic NombreClase
- Compilar la clase cliente
- Hacer accesibles vía web las clases necesarias
  - Interfaces, clases del cliente y del servidor (no todas)

RMI Ejemplo 1

Paso 4: Compilar interfaces

```
Windows.
cd c:\home\waldo\src
javac compute\Compute.java
javac compute\Task.java
jar cvf compute.jar compute\*.class
```

```
UNIX.
cd /home/waldo/src
javac compute/Compute.java
javac compute/Task.java
jar cvf compute.jar compute/*.class
```

RMI Ejemplo 1

Paso 4: Compilar servidor

```
Windows.
cd c:\home\ana\src
javac engine\ComputeEngine.java
rmic -d . engine.ComputeEngine
mkdir c:\home\ana\public_html\classes\engine
cp engine\ComputeEngine_*.class c:\home\ana\public_html\classes\engine
```

```
Unix.
cd /home/ana/src
javac engine/ComputeEngine.java
rmic -d . engine.ComputeEngine
mkdir /home/ana/public_html/classes/engine
cp engine/ComputeEngine_*.class  /home/ana/public_html/classes/engine
```

RMI Ejemplo 1

Paso 4: Compilar cliente

```
Windows:

set CLASSPATH=
    c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar

cd c:\home\jones\src

javac client\ComputePi.java

javac -d c:\home\jones\public_html\classes client\Pi.java
```

```
UNIX.

setenv CLASSPATH
    /home/jones/src:/home/jones/public_html/classes/compute
    .jar

cd /home/jones/src

javac client/ComputePi.java

javac -d /home/jones/public_html/classes client/Pi.java
```

RMI Ejemplo 1

Paso 5: Ejecutar la aplicación

- Arrancar el registro
- Arrancar el servidor
- Ejecutar el cliente

RMI Ejemplo 1

Paso 5: Arrancar el registro

Puerto por defecto

```
Windows (utilizar javaw si no está disponible start).
unset CLASSPATH
start rmiregistry
UNIX.
unsetenv CLASSPATH
rmiregistry &
```

Deshabilitar el classpath

Puerto 2001

```
Windows.
start rmiregistry 2001
UNIX.
rmiregistry 2001 &
```

RMI Ejemplo 1

Paso 5: Arrancar el servidor

```
Windows.
set CLASSPATH=c:\home\ana\src;c:\home\ana\public_html\classes\compute.jar
Unix.
setenv CLASSPATH /home/ana/src:/home/ana/public_html/classes/compute.jar
```

```
Windows.
java -Djava.rmi.server.codebase=file:/c:\home\ana\public_html\classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=java.policy
         engine.ComputeEngine
UNIX.
java -Djava.rmi.server.codebase=http://zaphod/~ana/classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=java.policy
         engine.ComputeEngine
```

RMI Ejemplo 1

Paso 5: Arrancar el s

```
Windows.
set CLASSPATH=c:\home\ana\src;c:\home
Unix.
setenv CLASSPATH /home/ana/src:/home/ana/public_html/classes/compute.jar
```

- Dirección desde donde el servidor distribuye sus clases
- Información de localización que se añadirá cuando se envien clases a otras máquinas virtuales
- Permitirá que el receptor pueda descargar dichas clases

```
Windows.
java -Djava.rmi.server.codebase=file:/c:\home\ana\public_html\classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=java.policy
         engine.ComputeEngine
UNIX.
java -Djava.rmi.server.codebase=http://zaphod/~ana/classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=java.policy
         engine.ComputeEngine
```

RMI Ejemplo 1

Paso 5: Arrancar el servidor

```
Windows.
set CLASSPATH=c:\home\ana\src;c:\home\ana\public_html\classes\compute.jar
Unix.
setenv CLASSPATH /home/ana/src:/home/ana/public_html/classes/compute.jar
```

Nombre del servidor

```
Windows.
java -Djava.rmi.server.codebase=file:/c:\home\ana\public_html\classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=java.policy
         engine.ComputeEngine
UNIX.
java -Djava.rmi.server.codebase=http://zaphod/~ana/classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=java.policy
         engine.ComputeEngine
```

RMI Ejemplo 1

Paso 5: Arrancar el servidor

```
Windows.
set CLASSPATH=c:\home\ana\src;c:\
Unix.
setenv CLASSPATH /home/ana/src:/home/ana/public_html/classes/compute.jar
```

Fichero con política de seguridad
Indica permisos concedidos a las clases del codebase

```
Windows.
java -Djava.rmi.server.codebase=file:/c:\home\ana\public_html\classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=java.policy
         engine.ComputeEngine
UNIX.
java -Djava.rmi.server.codebase=http://zaphod/~ana/classes/
     -Djava.rmi.server.hostname=zaphod.east.sun.com
     -Djava.security.policy=java.policy
         engine.ComputeEngine
```

RMI Ejemplo 1

Paso 5: Arrancar el cliente

```
Windows.
set CLASSPATH
    c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
java -Djava.rmi.server.codebase=file:/c:\home\jones\public_html\classes/
     -Djava.security.policy=java.policy
        client.ComputePi localhost 20
UNIX.
setenv CLASSAPTH
    /home/jones/src:/home/jones/public_html/classes/compute.jar
java -Djava.rmi.server.codebase=http://ford/~jones/classes/
     -Djava.security.policy=java.policy
        client.ComputePi zaphod.east.sun.com 20
```

RMI Ejemplo 1

Paso 5: Arrancar el c

•Dirección desde donde el cliente distribuye sus clases

```
Windows.
set CLASSPATH
    c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
java -Djava.rmi.server.codebase=file:/c:\home\jones\public_html\classes/
     -Djava.security.policy=java.policy
        client.ComputePi localhost 20
UNIX.
setenv CLASSAPTH
    /home/jones/src:/home/jones/public_html/classes/compute.jar
java -Djava.rmi.server.codebase=http://ford/~jones/classes/
     -Djava.security.policy=java.policy
        client.ComputePi zaphod.east.sun.com 20
```

RMI Ejemplo 1

Paso 5: Arrancar el cliente

Fichero que contiene la política de seguridad

```
Windows.
set CLASSPATH
    c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
java -Djava.rmi.server.codebase=file:/c:\home\jones\public_html\classes/
     -Djava.security.policy=java.policy
        client.ComputePi localhost 20
UNIX.
setenv CLASSAPTH
    /home/jones/src:/home/jones/public_html/classes/compute.jar
java -Djava.rmi.server.codebase=http://ford/~jones/classes/
     -Djava.security.policy=java.policy
        client.ComputePi zaphod.east.sun.com 20
```

RMI Ejemplo 1

Paso 5: Arrancar el cliente

Parámetro 1: Nombre del servidor que realizará el cálculo

```
Windows.
set CLASSPATH
    c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar
java -Djava.rmi.server.codebase=file:/c:\home\jones\public_html\classes/
     -Djava.security.policy=java.policy
        client.ComputePi localhost 20
UNIX.
setenv CLASSAPTH
    /home/jones/src:/home/jones/public_html/classes/compute.jar
java -Djava.rmi.server.codebase=http://ford/~jones/classes/
     -Djava.security.policy=java.policy
        client.ComputePi zaphod.east.sun.com 20
```

RMI Ejemplo 1

Paso 5: Arrancar el cliente

```
Windows.
set CLASSPATH
    c:\home\jones\src;c:\home\jones\pu
java -Djava.rmi.server.codebase=file:/c:\home\jones\public_html\classes/
     -Djava.security.policy=java.policy
        client.ComputePi localhost 20
UNIX.
setenv CLASSAPTH
    /home/jones/src:/home/jones/public_html/classes/compute.jar
java -Djava.rmi.server.codebase=http://ford/~jones/classes/
     -Djava.security.policy=java.policy
        client.ComputePi zaphod.east.sun.com 20
```

Parámetro 2: Número de decimales con el que queremos que realice el cálculo

RMI: ¿Cómo funciona?
Servicios utilizados

- Secuenciación de objetos (*serialization*)
- Servicio de nombrado (*naming service*)
  - En particular, *RMI registry*
- Descarga de clases por la red: (*Class Loader*)
- Servicios de seguridad (*Security Manager*)
- Recolección de basura distribuida (*distributed garbage collection*)

RMI: ¿Cómo funciona?
Secuenciación de objetos (*serialization*)

- El paso de parámetros por valor introduce un problema:

Si se pasa un objeto por la red y éste contiene referencias a otros objetos, ¿cómo se resuelven las referencias en la máquina de destino?

- Secuenciar un objeto consiste en convertirlo en una secuencia de bits que representa a ese objeto.
- Para hacer que un objeto sea secuenciable, éste debe implementar el interfaz **`java.io.Serializable`**
  - Es una interfaz de marcado (no contiene métodos): indica que un objeto puede ser secuenciable (*serializable)* y reconstruible (*deserializable)*.
- Es posible implementar una *secuenciación* a medida (*custom serialization*) con **`writeObject()`** y **`readObject()`**.
  - Normalmente. los mecanismos existentes por defecto son suficientemente buenos.

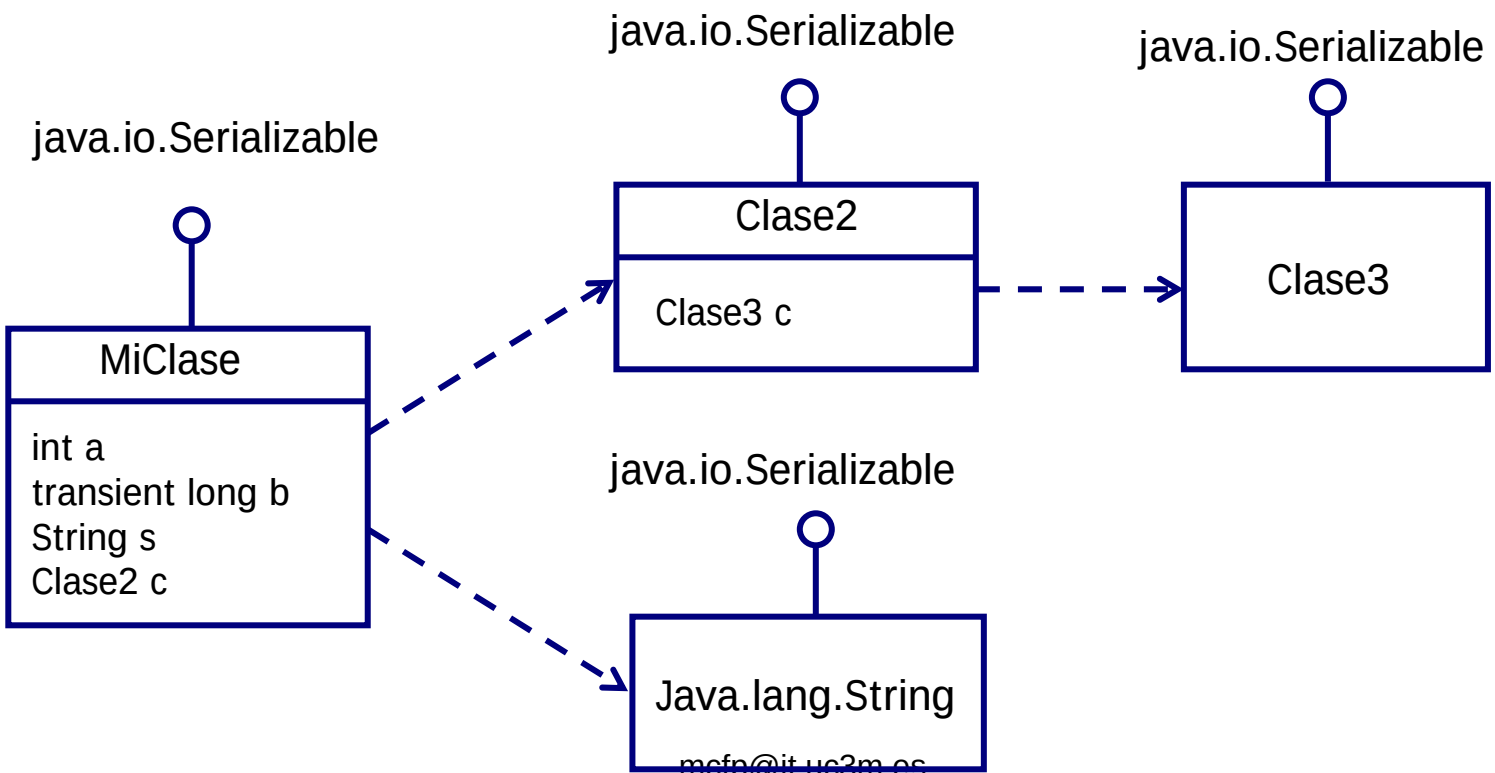
RMI: ¿Cómo funciona?

Reglas para secuenciar objetos

- Cualquier **TIPO PRIMITIVO** (**`int`**, **`char`**, etc.) es secuenciado automáticamente y está disponible en la reconstrucción del objeto al que pertenece.
- Los **OBJETOS** contenidos en el objeto a secuenciar pueden o no ser secuenciados:
  - Si se marcan con la palabra **`transient`**, éstos no son secuenciados con el objeto y no están disponibles en la reconstrucción.
  - Los objetos no marcados con **`transient`** deberán implementar el interfaz **`java.io.Serializable`**.
  - Si no están marcados con **`transient`** ni implementan **`java.io.Serializable`**, se lanza una excepción **`NotSerializable`**.
- Objetos *transient* típicos:
  - objetos muy grandes
  - recursos no reconstruibles en la máquina de destino (sockets o conexiones a bases de datos)
  - información confidencial.

RMI: ¿Cómo funciona?
Secuenciación recursiva

- Cuando se secuencia un objeto, todos sus objetos no *transient*, también serán secuenciados.
- Esto se realiza de forma recursiva para todos los subobjetos.



RMI: ¿Cómo funciona?
Paso de parámetros / valores de retorno

- Dos maneras de pasar parámetros a métodos remotos
  - **por valor**: se inserta una copia "secuenciada" del objeto en el flujo de salida que corresponde al envío de la invocación o el retorno
    - es el objeto remoto el que viaja
  - **por referencia**: se inserta una copia "secuenciada" del *stub* del objeto en el flujo de salida que corresponde al envío de la invocación o el retorno
    - es el *stub* del objeto remoto (instancia de la clase *stub*) el que viaja.
- Tres casos:
  - **tipos primitivos** : se pasan por valor; todos son *serializables*
  - **objetos locales**: se pasan por valor (si no son *serializables,* exception), se crea un nuevo objeto en la máquina virtual que recibe la copia.
  - **objetos remotos**: se pasan por referencia, se crea un nuevo *stub* en la máquina virtual que recibe la copia

RMI: ¿Cómo funciona?
Paso de parámetros / valores de retorno

- Observaciones:
  - Los objetos remotos no viajan, en cambio se envían referencias
  - Un *stub* se debe convertir (*cast*) al tipo de la interfaz remota que implemente la clase del objeto remoto al que corresponde
    - si un objeto remoto implementa varias interfaces remotas un cliente solo puede convertir el *stub* a una de ellas
  - Dos *stubs* que se refieren al mismo objeto remoto en el mismo servidor se consideran iguales bajo la operación **`equals()`**

RMI: ¿Cómo funciona?
Declaración de parámetros

- Los parámetros / valores de retorno que son objetos remotos se pasan por referencia
- Por tanto, la clase de un parámetro de un método de una interfaz remota no puede ser la de un objeto remoto
  - su clase es la de un objeto no remoto o la de una interfaz remota
- Por ejemplo, si **`MyRemote`** es una interfaz remota y **`MyRemoteImpl`** es una clase que la implementa y **`metodo`** es un método de otra interfaz remota, con respecto a la declaración de **`metodo`**:

```
public void metodo(MyRemote remoto)          ¡Bien!
                throws RemoteException

public void metodo(MyRemoteImpl remoto)      ¡Mal!
                throws RemoteException
```

RMI
Objeto remoto vs objeto local

Objeto local

- Instancia de una clase Java
- Se crea una nueva instancia con **`new()`**
- Acceso directo
- Eliminado por el *garbage collector* cuando no hay referencias locales que apuntan a el
- Sus métodos no pueden lanzar **`java.rmi.RemoteException`**

Objeto remoto

- Instancia de una clase Java que implementa una o varias interfaces que extienden **`java.rmi.Remote`**
- En la misma JVM se crea una nueva instancia con **`new()`**; desde otra JVM sólo se puede crear una nueva instancia por activación remota
- Acceso por un *stub*
- Eliminado por el *garbage collector* cuando no hay ni referencias locales ni referencias remotas activas que apuntan a el; una referencia remota se considera activa si no ha sido soltado y se ha realizado un acceso hace poco (con un tiempo configurable)
- Sus métodos pueden lanzar **`java.rmi.RemoteException`**

RMI Registry
Servidor de nombres

- ¿Cómo el cliente encuentra un servicio remoto de RMI?
  - Por medio de un servicio de nombres o de directorios
    - relaciona nombres (cadenas de caracteres) con objetos remotos
  - El servicio de nombres/directorios tiene que estar escuchando
    - en un puerto conocido
    - de una máquina conocida
- RMI puede utilizar distintas servicios de nombres, p.e.:
  - JNDI (*Java Naming and Directory Interface*)
  - RMI Registry: servicio sencillo incluido con RMI
- RMI registry
  - Implementación actual ejecuta en la misma máquina que el objeto remoto
  - Por defecto, escucha en el puerto 1099

RMI Registry
Clases e interfaces

- Interfaz remota **`Registry`**
  - **`lookup()`**, **`bind()`**, **`unbind()`**, **`rebind()`**, **`list()`**
  - Trabaja con nombres que son simples cadenas de caracteres
- Clase **`LocateRegistry`**
  - **`createRegistry()`**: crea un nuevo objeto que implementa la interfaz **`Registry`** y devuelve una referencia a este objeto
  - **`getRegistry()`**: devuelve una referencia a un objeto que implementa la interfaz **`Registry`**
    - en una máquina dada
    - en una máquina dada y un puerto dado
- Clase **`Naming`**
  - Invoca métodos de un objeto remoto que implementa **`Registry`**
  - Trabaja con nombres en forma de URL:

    rmi://maquina:puerto/ruta

RMI Registry
Clase principal: **`Naming`**

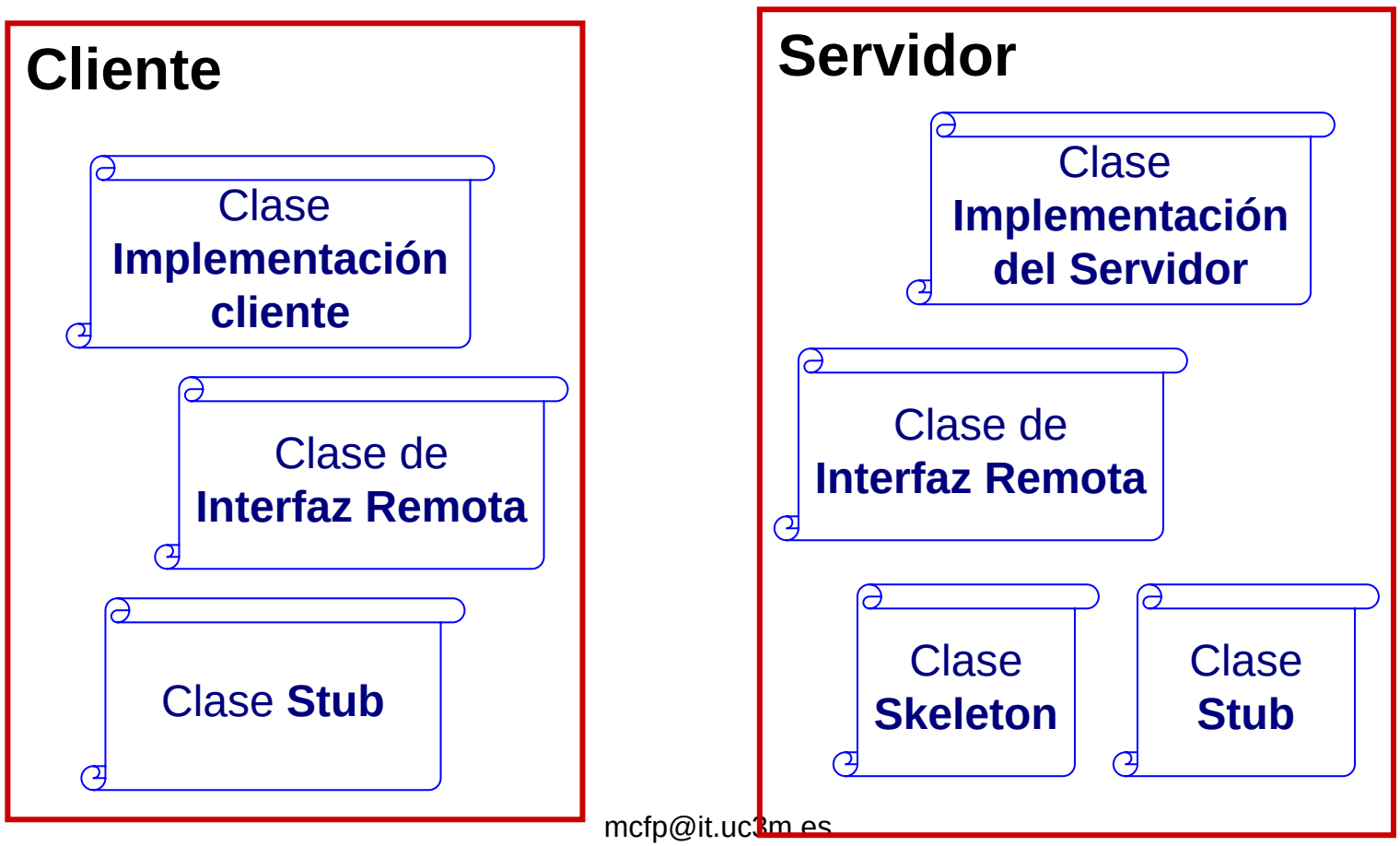
- Método **`lookup(`**URL**`)`**
  - devuelve el objeto remoto (eso es, su referencia) que corresponde al URL
- Método **`bind(`**URL**`,`** objeto remoto**`)`**
  - asocia el URL al objeto remoto (eso es, a su referencia)
- Método **`unbind(`**URL**`)`**
  - desasocia el URL del objeto remoto (eso es, de su referencia)
- Método **`rebind(`**URL**`,`** objeto remoto**`)`**
  - asocia el URL al objeto remoto (eso es, a su referencia) borrando la previa
  - el uso de **`rebind`** evita la posibilidad de una **`AlreadyBoundException`**
- Método **`list()`**
  - Devuelve un array que contiene los URLs actualmente asociados a un objeto remoto en el *registry*

RMI
Retrollamadas (*callbacks*)

- Un servidor puede necesitar hacer una llamada al cliente
- Para lograrlo, el cliente debe comportarse como servidor
- Muchas veces es más práctico llamar a
  `java.rmi.server.UnicastRemoteObject.exportObject()`
  en vez de extender **`UnicastRemoteObject`**

RMI
Ubicación de ficheros

- Sin descarga automática de clases:



RMI
Descarga dinámica de clases

- RMI utiliza serialización para enviar datos entre máquinas distintas
  - En el proceso de **marshaling** se serializan los objetos y se envian junto con información de localización para permitir que la definición de clases se pueda descargar en la máquina que recibe los objetos.
  - En el proceso de **unmarshalling** para convertir de nuevo los datos en objetos activos es necesario resolver la localización de las clases que definen esos objetos. Cuando no se pueden resolver las clases de manera local es necesario descargar la definición de las clases de una máquina remota.
- Cuando un cliente invoca un método de un objeto remoto es posible que tenga que tratar con objetos cuyas definiciones de clase son desconocidas para el classLoader local.
- Para que los objetos remotos estén activos en la MV local es necesario **descargar la definición de las clases** de todos los objetos remotos con los que tenga que tratar:
  - La definición de clase del **stub** que representa al objeto cuyo método se invoca.
  - La definición de clase de los objetos que se pasen como **parámetros**
  - La definición de clase de los objetos que se devuelvan como **tipo de retorno**
  - La definición de clase de los objetos **excepción** que se lancen en el método

RMI
Descarga dinámica de clases

- Durante el *marshalling* , junto con el nombre de una clase, RMI coloca automáticamente su localización (un *codebase*) en forma de URL
  - el URL de la clase se transmite en el flujo de salida
- Si un cliente o un servidor encuentra el nombre de una clase que no reconoce, intenta cargarla como sigue:
  - del CLASSPATH local utilizando el *Class Loader* local, si es posible
  - del URL transmitido junto con el nombre, utilizando el **`RMIClassLoader`** en caso de un *stub* encontrado por un servidor como parámetro de una llamada a un método remoto o por un cliente como resultado de una tal llamada
  - de un URL especificado en la propiedad **`java.rmi.server.codebase`** utilizando el **`RMIClassLoader`** en caso de una clase local
- Especificación del *codebase* que proporciona las clases para objetos (p.e. Stubs) enviados desde este JVM:

```
java -Djava.rmi.server.codebase=http://maquina/ruta/classes ServImpl
```

- Se puede obligar la carga de todas las clases del *codebase* local poniendo la propiedad **`java.rmi.server.useCodebaseOnly`** a **`true`**

RMI, descarga de clases
Seguridad

- El **`RMIClassLoader`** no descarga clases remotas si no hay instalado un **`SecurityManager`**

```
System.setSecurityManager(new RMISecurityManager());
```

- El **`RMISecurityManager`** impide que los *stubs* escuchen en puertos, manipulen hilos fuera de su grupo, enlacen con DLLs, creen procesos, abran descriptores de ficheros, etc.

- Para aplicaciones especiales el programador puede sustituir el cargador de clases (**`RMIClassLoader`**) y el gestor de seguridad (**`RMISecurityManager`**) que vienen con RMI por el suyo propio pero para las aplicaciones estándar no es necesario.

RMI
Activación / desactivación de objetos

- La clase **`Activatable`** y el demonio **`rmid`** permiten
  - la creación y ejecución de objetos remotos bajo demanda
- La clase que implementa la interfaz remota debe extender
  - **`java.rmi.activation.Activatable`**
- Hace falta también una clase **`setup`** (montaje) para
  - instalar un **`SecurityManager`**
  - pasar la información sobre la clase activable al **`rmid`**
  - registrar una referencia y un nombre con el **`rmiregistry`**

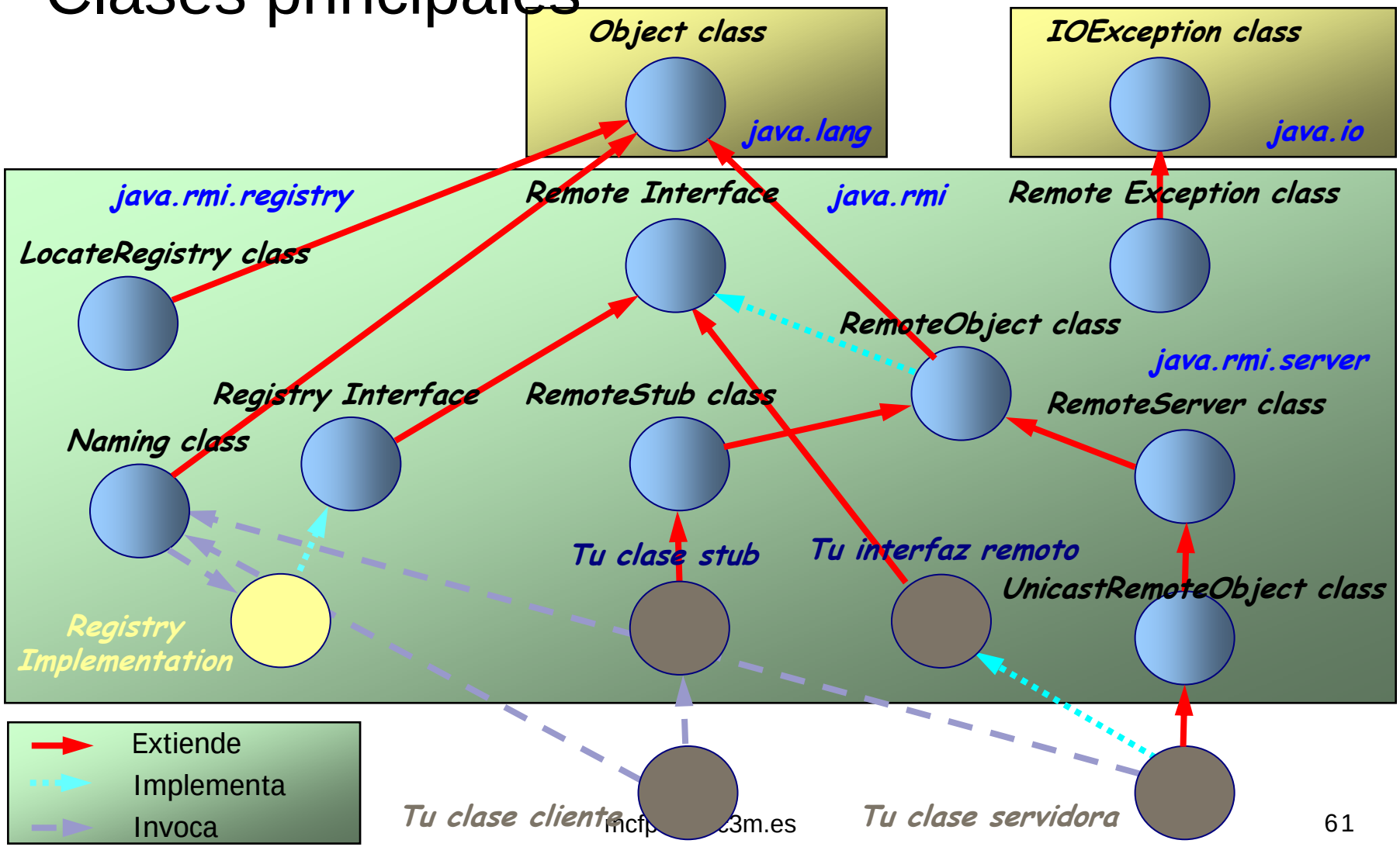
  entre otras cosas
- El **`rmid`** requiere un fichero de política de seguridad (*security policy*)

RMI
Paquetes

- Package **`java.rmi`**
  clientes: para acceder a servicios remotos RMI y para ubicar servicios RMI en máquinas remotas.
- Package **`java.rmi.server`**
  Servidores: para hacer accesible un servicio RMI a peticiones TCP/IP y HTTP proxy.
- Package **`java.rmi.registry`**
  Creación y ubicación de registros de nombres.
- Package **`java.rmi.dgc`**
  Recolección de basura para un entorno distribuido.
- Package **`java.rmi.activation`** (J2SE SDK 1.2+)
  Permite que los servidores sólo sean activados cuando haya una petición real de servicio.

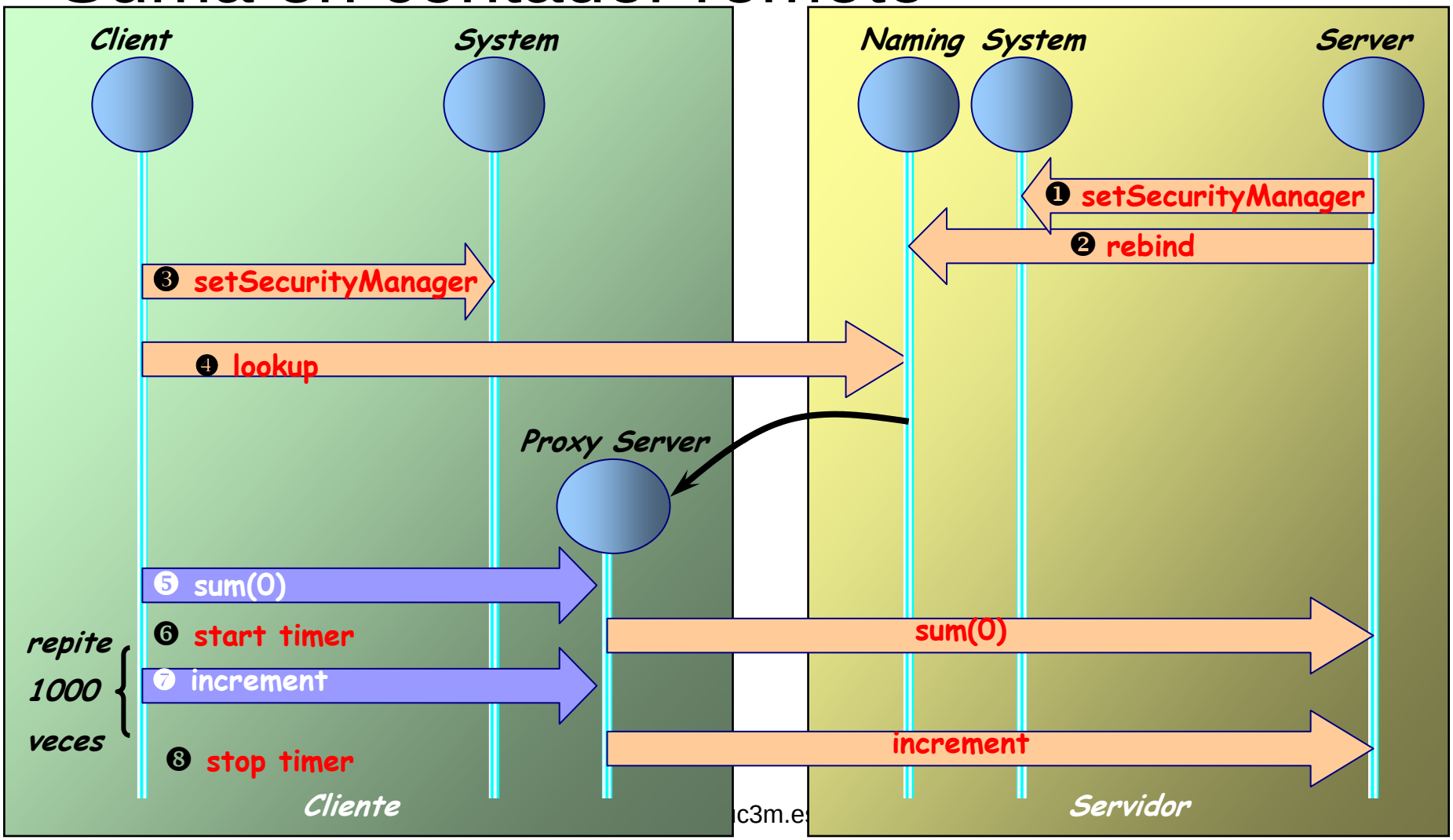
RMI

Clases principales



Escenario básico

Suma en contador remoto



RMI Ejemplo 2

Paso 1: Definir interfaz remota

- Importar paquetes adecuados
  - `import java.rmi.Remote;`
  - `import java.rmi.RemoteException;`
- Extender la clase Remote
  - `public interface NombreInt extends Remote{}`
- Lanzar remoteException en todos los métodos
  - `public Tipo NombreMet() throws RemoteException;`

RMI Ejemplo 2
Definiendo interfaz remota

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface DateServer extends Remote {
    public Date getDate() throws RemoteException;
}
```

Ejemplo tomado de
Java Network Programming
Merlin Hughes, Michael Shoffner, Derek Hamner
Manning Publications

RMI Ejemplo 2

Paso 2: Implementar interfaz remota

- Extender la clase UnicastRemoteObject
- Implementar el interfaz remoto definido en el paso 1
- Crear main que realice las siguientes tareas
  - Crear controlador de seguridad
  - Crear objeto de la clase remota
  - Registrar objeto de la case remota

RMI Ejemplo 2

Objeto remoto (+ servidor)

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class DateServerImpl extends UnicastRemoteObject
                            implements DateServer {

  public DateServerImpl throws RemoteException{}
  public Date getDate() { return new Date(); }

  public static void main(String[] args) throws Exception{
     DateServerImpl dateServer = new DateServerImpl();
     Naming.bind("Date Server", dateServer);
  }
}
```

RMI Ejemplo 2

Paso 3: Compilación de clases e interfaces

Paso 4: Generación de stub y skeletons

- Compilar interfaces
  - javac NombreInterfaz.java
- Compilar la clase servidora
  - javac NombreClase.java
- Generar stubs y skeletons
  - rmic NombreClase

RMI Ejemplo 2

Generación del *stub* y *skeleton*

```
prompt> rmic DateServerImpl
```

Resultado:

DateServerImpl_Stub

DateServerImpl_Skel

RMI Ejemplo 2

Paso 5: Implementación del cliente

- Crear controlador de seguridad
- Asignar nombre al objeto remoto
- Invocar métodos del objeto remoto

RMI Ejemplo 2
Implementación del cliente

```
Import java.rmi.Naming;
Import java.util.Date;

public class DateClient {

  public static void main(String[] args) throws Exception{
    if (args.length != 1)
      throw new IllegalArgumentException (“Syntax: DateClient ”
                                                + “<hostname>”);
      DateServer dateServer = (DateServer)
                Naming.lookup(“rmi://” + args[0] + “/DateServer”);
      Date when = dateServer.getDate();
      System.out.println(when);
    }

  }

}
```

RMI Ejemplo 2
Arranque de la applicación

```
prompt> rmiregistry                    // arrancar el registro

prompt> java DateServerImpl            // arrancar el servidor

prompt> java DateClient localhost      // arrancar el cliente
```

RMI Ejemplo 3

Interfaz remota

```
public interface Calculator extends java.rmi.Remote {
    public long add(long a, long b)
                throws java.rmi.RemoteException;
    public long sub(long a, long b)
                throws java.rmi.RemoteException;
    public long mul(long a, long b)
                throws java.rmi.RemoteException;
    public long div(long a, long b)
                throws java.rmi.RemoteException;
}
```

Ejemplo tomado de
jGuru: Remote Method Invocation: Introduction
http://java.sun.com/developer/onlineTraining/rmi/index.html

RMI Ejemplo 3

Objeto remoto

```
public class CalculatorImpl
                extends java.rmi.server.UnicastRemoteObject
                implements Calculator {

    // Implementations must have an explicit constructor
    // in order to declare the RemoteException exception
    public CalculatorImpl() throws java.rmi.RemoteException {
    super(); }

    public long add(long a, long b) throws
    java.rmi.RemoteException { return a + b; }

    public long sub(long a, long b) throws
    java.rmi.RemoteException { return a - b; }

    public long mul(long a, long b) throws
    java.rmi.RemoteException { return a * b; }

    public long div(long a, long b) throws
    java.rmi.RemoteException { return a / b; }
}
```

RMI Ejemplo 3
Servidor anfitrión

```
import java.rmi.Naming;

public class CalculatorServer {

  public CalculatorServer() {
    try { Calculator c = new CalculatorImpl();
       Naming.rebind("rmi://localhost:1099/CalculatorService", c);
    }
    catch (Exception e) {
       System.out.println("Trouble: " + e);
    }
  }

  public static void main(String args[]) {
    new CalculatorServer();
  }

}
```

RMI Ejemplo 3

Implementación del cliente

```
public class CalculatorClient {

     public static void main(String[] args) {

          try { Calculator c = (Calculator)
                    Naming.lookup( "rmi://localhost /CalculatorService");
                    System.out.println( c.sub(4, 3) );
                    System.out.println( c.add(4, 5) );
                    System.out.println( c.mul(3, 6) );
                    System.out.println( c.div(9, 3) );

          } catch (MalformedURLException murle) {
                    System.out.println( "MalformedURLException " + murle);
          } catch (RemoteException re) {
                    System.out.println( "RemoteException " + re);
          } catch (NotBoundException nbe) {
                    System.out.println( "NotBoundException " + nbe);
          } catch (java.lang.ArithmeticException ae) {
                    System.out.println( "java.lang.ArithmeticException " + ae);
          }

     }
}
```

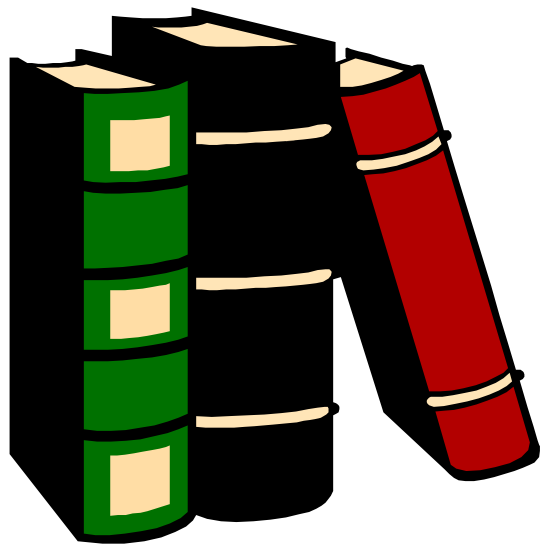
RMI: Algunas referencias

– ***Dan Harkey, Robert Orfali***, ***Client/Server Programming with Java and CORBA, 2nd Edition*** *2nd. Edition (1998) John Wiley & Sons, Inc. ISBN: 0-471-24578-X*
Mirarse el capítulo 13

– ***Cay S. Horstmann, Gary Cornell***
Core Java 2, Volume 2: Advanced Features *4 edition Vol 2 (December 27, 1999) Prentice Hall PTR; ISBN: 0130819344*
Mirarse el capítulo 5

– ***Qusay H. Mahmoud***, ***Distributed Programming with Java*** *(September 1999) Manning Publications Company; ISBN: 1884777651*

– ***Jim Farley***, ***Java: Distributed Computing*** *(January 1998) O'Reilly & Associates; ISBN: 1565922069*



RMI: Referencias web

- Tutorial sencillo con ejemplos

  http://www.programacion.com/java/tutorial/rmi/
- Más referencias: