

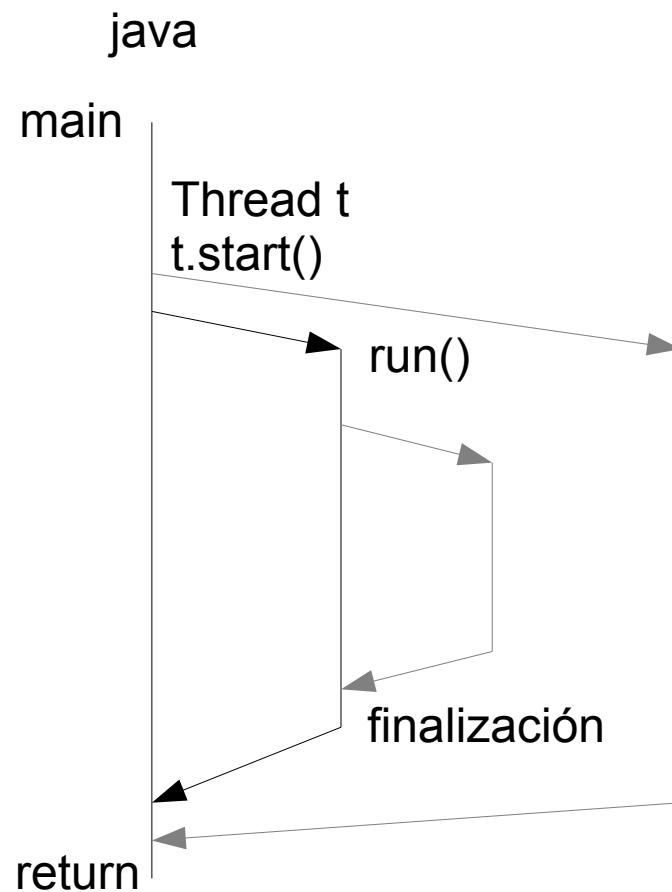
Hilos en Java



Rodrigo Santamaría

Hilos

- Un hilo (Thread) es un proceso en ejecución dentro de un programa



- La finalización depende del hilo (`Thread.suspend`, `stop` están depreciados)
- Los hilos implementan prioridad y mecanismos de sincronización
- Cualquier clase se puede hacer hilo
 - implements `Runnable`

Hilos

```
public class PingPong extends Thread  
{  
    private String word;  
    public PingPong(String s) {word=s;}  
  
    public void run()  
    {  
        for (int i=0;i<3000;i++)  
        {System.out.print(word);  
         System.out.flush();}  
    }  
  
    public static void main(String[] args)  
    {Thread tP=new PingPong("P");  
     Thread tp=new PingPong("p");  
     //tP.setPriority(Thread.MAX_PRIORITY);  
     //tp.setPriority(Thread.MIN_PRIORITY);  
     tp.start();  
     tP.start();  
    }  
}
```

- Clase Thread
 - Implementa Runnable
 - start() → run()
 - stop(), suspend()
 - setPriority()
 - sleep()
- Hereda de Object
 - wait(), notify()

Sincronización

- Los hilos se comunican generalmente a través de campos y los objetos que tienen esos campos
 - Es una forma de comunicación eficiente
 - Pero puede plantear errores de interferencias entre hilos
- La sincronización es la herramienta para evitar este tipo de problemas, definiendo órdenes estrictos de ejecución

Interferencia entre hilos

```
class Counter  
{  
    private int c = 0;  
    public void increment() { c++; }  
    public void decrement() { c--; }  
    public int value() { return c; }  
}
```

c++ está compuesto de:

1. Obtener el valor de c
2. Incrementar c en 1
3. Almacenar el valor de c

c++ está compuesto de:

4. Obtener el valor de c
5. Decrementar c en 1
6. Almacenar el valor de c

- Dos hilos A y B pueden estropearlo:
 - Hilo A: recuperar c (0)
 - Hilo B: recuperar c (0)
 - Hilo A: incrementar c (1)
 - Hilo B: decrementar c (-1)
 - Hilo A: almacenar c (1)
 - Hilo B: almacenar c (-1)

Métodos sincronizados

- Convertir un método en sincronizado tiene dos efectos:
 - Evita que dos invocaciones de métodos sincronizados del mismo objeto se mezclen.
Cuando un hilo ejecuta un método sincronizado de un objeto, todos los hilos que invoquen métodos sincronizados del objeto se bloquearán hasta que el primer hilo termine con el objeto.
 - Al terminar un método sincronizado, se garantiza que todos los hilos verán los cambios realizados sobre el objeto.

Bloqueo intrínseco

- Cuando un hilo invoca un método sincronizado, adquiere el *bloqueo intrínseco* del objeto correspondiente.
- Si invoca un método *estático* sincronizado, adquiere el bloqueo intrínseco de la *clase*, independiente de los de sus objetos

Métodos sincronizados

Hilo 1

Hilo 2

Hilo 3

objeto

métodos

...

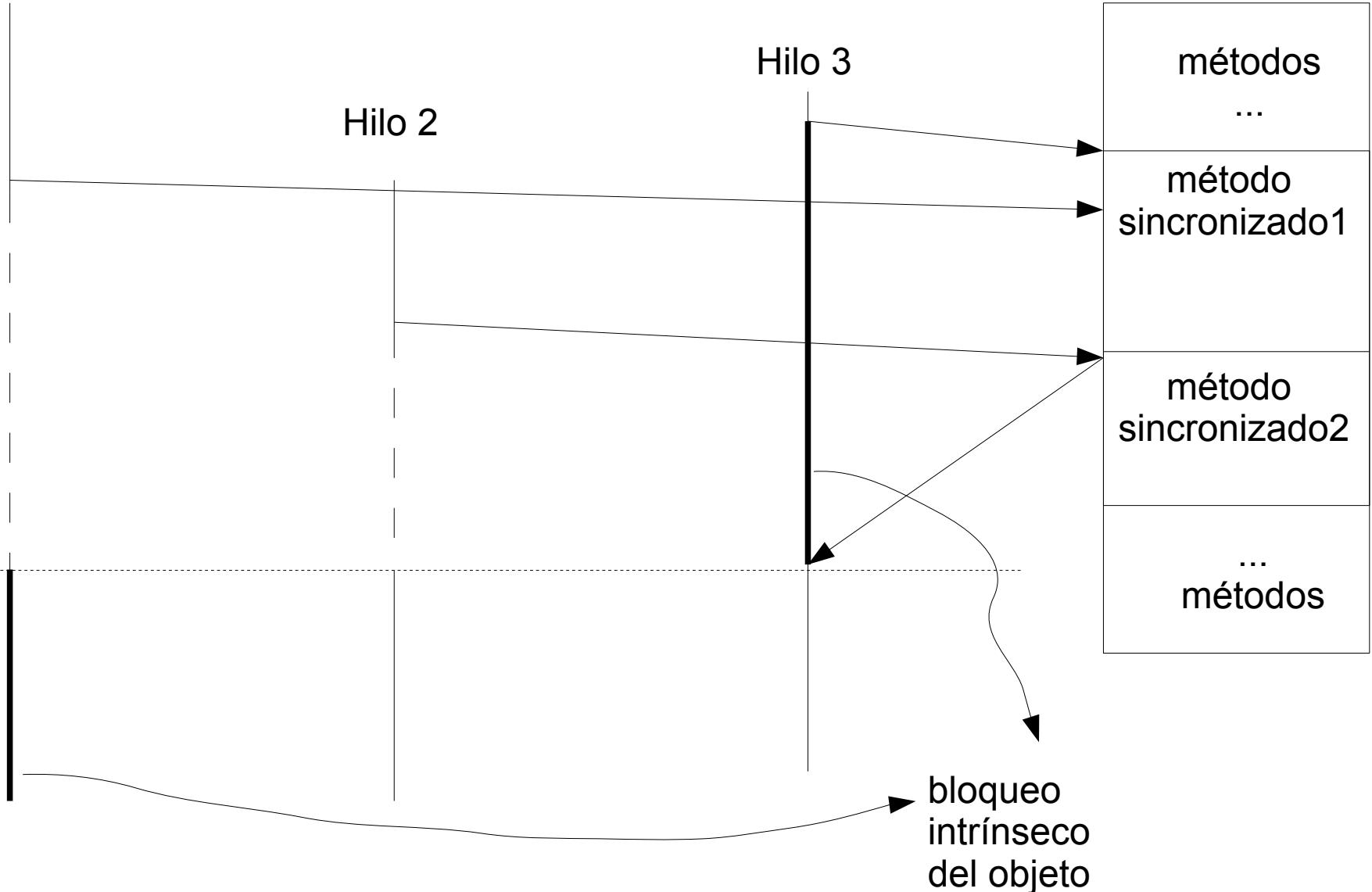
método
sincronizado1

método
sincronizado2

...

métodos

bloqueo
intrínseco
del objeto



Métodos sincronizados

```
class Counter

{
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }
    public int value() { return c; }
}
```

Código sincronizado

- En vez de sincronizar todo un método podemos sincronizar una porción de código
 - Debemos especificar sobre qué quieren el bloqueo intrínseco
 -

```
public void addName(String name)
{
    synchronized(this)
    {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Código sincronizado

```
public class MsLunch
{
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1()
    {
        synchronized(lock1)
        {
            c1++;
        }
    }

    public void inc2()
    {
        synchronized(lock2)
        {
            c2++;
        }
    }
}
```

- Grado más fino de sincronización
- Si la sincronización es sobre un atributo estático, se bloquea *la clase*.

Problemas con múltiples procesos

- **Espera ocupada:** un proceso espera por un recurso, pero la espera consume CPU
 - `while(!recurso) ; wait()`
- **Interbloqueo (deadlock):** varios procesos compiten por los mismos recursos pero ninguno los consigue
- **Inanición:** un proceso nunca obtiene los recursos que solicita, aunque no esté interbloqueado
- **Autobloqueo:** un proceso espera por recursos que ya posee → **sincronización reentrante**

Sincronización reentrante

- **Autobloqueo:** un proceso tiene el bloqueo intrínseco de un objeto
 - Mientras lo tiene, vuelve a pedirlo
- La sincronización reentrante evita que un proceso se bloquee a sí mismo en una situación de autobloqueo
 - Implementada en el núcleo de Java.

wait() y notify()

- **wait()** suspende el hilo hasta que se recibe una notificación del objeto sobre el que espera
 - Solución para *espera ocupada*
- El proceso que espera debe tener el bloqueo intrínseco del objeto que invoca al wait
 - Si no, da un error (`IllegalMonitorStateException`)
 - Una vez invocado, el proceso suspende su ejecución y libera el bloqueo
 - `wait(int time)` espera sólo durante un tiempo
- **notify()/notifyAll()** informan a uno/todos los procesos esperando por el objeto que lo invoca de que pueden continuar

Sincronización

```
public class SynchronizedPingPong extends Thread  
{  
    private String word;  
    public SynchronizedPingPong(String s) {word=s;}
```

```
public void run()  
{  
    synchronized(getClass())  
    {  
        for (int i=0;i<3000;i++)  
        {  
            System.out.print(word); Ejecuto una iteración  
            System.out.flush();  
            getClass().notifyAll(); Aviso de que he terminado  
            try  
            {getClass().wait();} Espero un aviso  
            catch (java.lang.InterruptedIOException e) {}  
        }  
        getClass().notifyAll();  
    }  
}
```

```
public static void main(String[] args)  
{  
    SynchronizedPingPong tP=new SynchronizedPingPong("P");  
    SynchronizedPingPong tp=new SynchronizedPingPong("p");  
    tp.start();  
    tP.start();  
}
```

“Para entrar por aquí tenemos que conseguir el bloqueo intrínseco de la clase SynchronizedPingPong”

wait()

- **this.wait()** → espera por esta instancia de la clase
- **this.getClass().wait()** o **getClass().wait()** → espera por la clase de esta instancia
- **objeto.wait()** → espera por el objeto que sea.
- **objetoEstático.wait()** → espera por la clase del objeto estático que sea
- En cualquier caso, dentro de **synchronized()** sobre el objeto/clase a la que espera
- Para liberar, **notify/notifyAll** sobre el objeto/clase sobre la que se espere

Depuración de interbloqueos

```
import java.lang.management.*;
```

```
...
```

```
ThreadMXBean tmx = ManagementFactory.getThreadMXBean();
long[] ids = tmx.findDeadlockedThreads();
if (ids != null)
{
    ThreadInfo[] infos = tmx.getThreadInfo(ids, true, true);
    System.out.println("The following threads are deadlocked:");
    for (ThreadInfo ti : infos)
        System.out.println(ti);
}
```

Carrera 4x100

- Implementar una carrera por relevos, similarmente al ejercicio anterior:
 - Tenemos 4 Atletas dispuestos a correr
 - Tenemos una clase principal Carrera
 - Tenemos un objeto estático testigo
 - Todos los atletas empiezan parados, uno comienza a correr (tarda entre 9 y 11s) y termina su carrera e inmediatamente comienza otro
 - Pistas:
 - Thread.sleep y Math.random para la carrera
 - synchronized, wait y notify para el paso de testigos
 - System.currentTimeMillis o Calendar para ver tiempos

La carrera amable

- Tenemos ocho atletas en una carrera de 100m
 - Cada atleta llega a meta tras entre 9 y 11s
 - Cuando uno llega a la meta, espera por el siguiente que llegue, y cruza la meta. El nuevo que ha llegado espera por el siguiente, y así sucesivamente
 - Tras cruzarla, imprime su nombre y el tiempo actual
- Programar dicho esquema mediante una clase Atleta que implemente Runnable
 - Pistas:
 - Thread.sleep y Math.random para la carrera
 - synchronized, wait y notify para la meta
 - System.currentTimeMillis o Calendar para ver tiempos

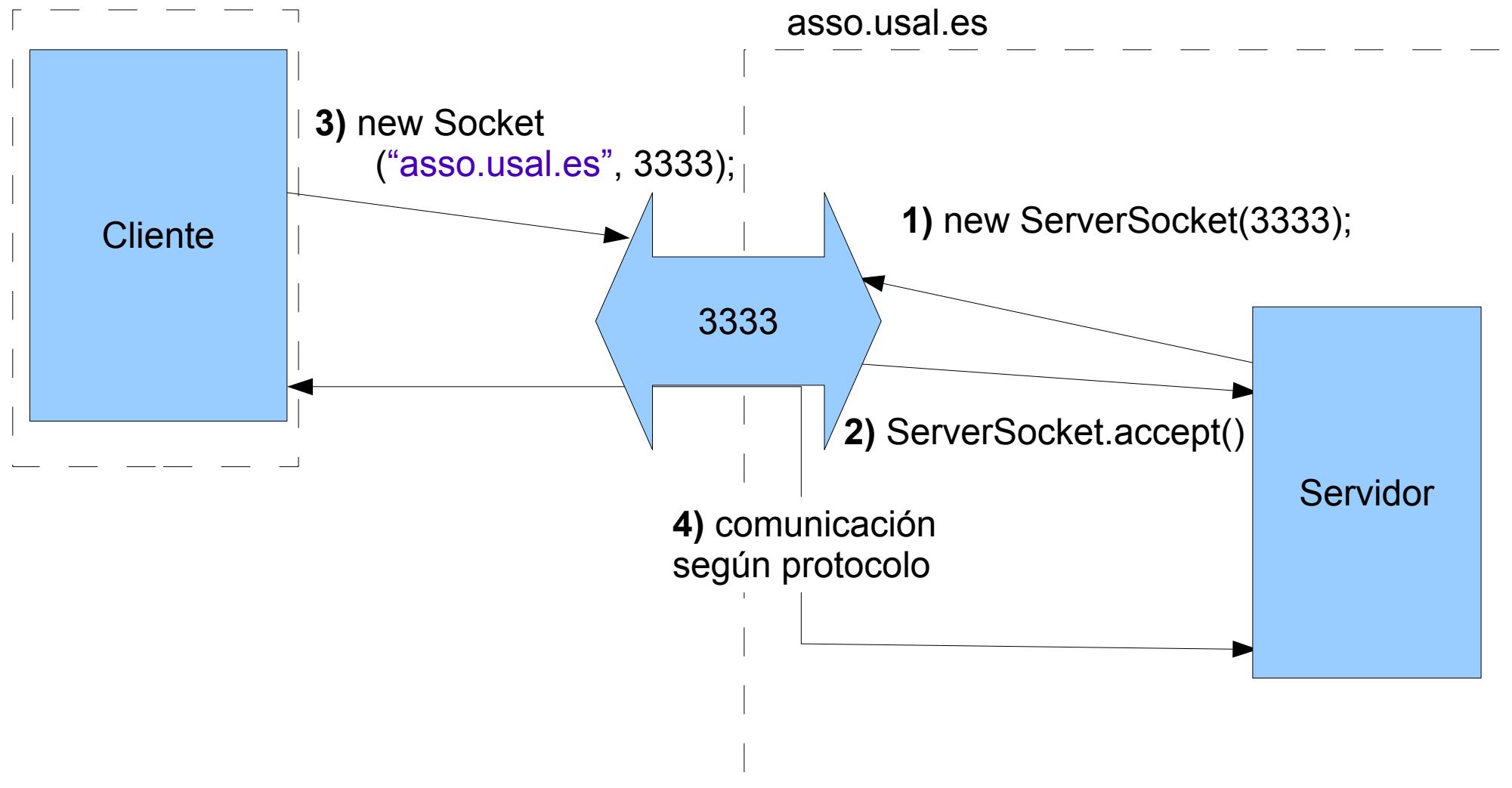
Sockets
en Java



Sockets

- Socket = enchufe
 - Punto final del enlace de comunicación entre dos procesos (cliente y servidor)
 - Clases Socket y ServerSocket
- Protocolo normal de comunicación
 - 1) Abrir el socket
 - 2) Abrir flujos de entrada y salida al socket
 - 3) Leer y escribir en los flujos según el protocolo del servidor
 - 4) Cerrar los flujos
 - 5) Cerrar el socket

Sockets



Socket

```
public class EchoClient {  
    public static void main(String[] args) throws IOException {  
        Socket echoSocket = null;  
        PrintWriter out = null;  
        BufferedReader in = null;  
        String ip="carpex.fis.usal.es";  
  
        try {  
            echoSocket = new Socket(ip, 80); //nuevo socket conectado a IP y puerto  
            out = new PrintWriter(echoSocket.getOutputStream(), true); //salida al socket  
            in = new BufferedReader(new InputStreamReader( //entrada al socket  
                echoSocket.getInputStream()));  
        }  
        catch (UnknownHostException e)  
        { System.err.println("Don't know about host: "+ip); System.exit(1); }  
        catch (IOException e)  
        { System.err.println("Couldn't get I/O for the connection to: "+ip); System.exit(1); }  
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));  
        String userInput;  
  
        while ((userInput = stdIn.readLine()) != null) {  
            out.println(userInput);  
            System.out.println("echo: " + in.readLine());  
        }  
        stdIn.close();  
        out.close();  
        in.close();  
        echoSocket.close();  
    }  
}
```

Abrir
socket y
flujos

Leer y escribir según el
protocolo

Cerrar flujos y
socket

ServerSocket

1) Establecer el servicio

```
ServerSocket server = new ServerSocket(3333);
```

2) Esperar por peticiones

```
Socket client = server.accept();
```

3) Procesar peticiones

```
in = new BufferedReader(new  
                    InputStreamReader(client.getInputStream()));  
out = new PrintWriter(client.getOutputStream(), true);  
while(true)  
{  
    line = in.readLine();  
    out.println(line); //Servicio de echo  
}
```

4) Terminar el servicio

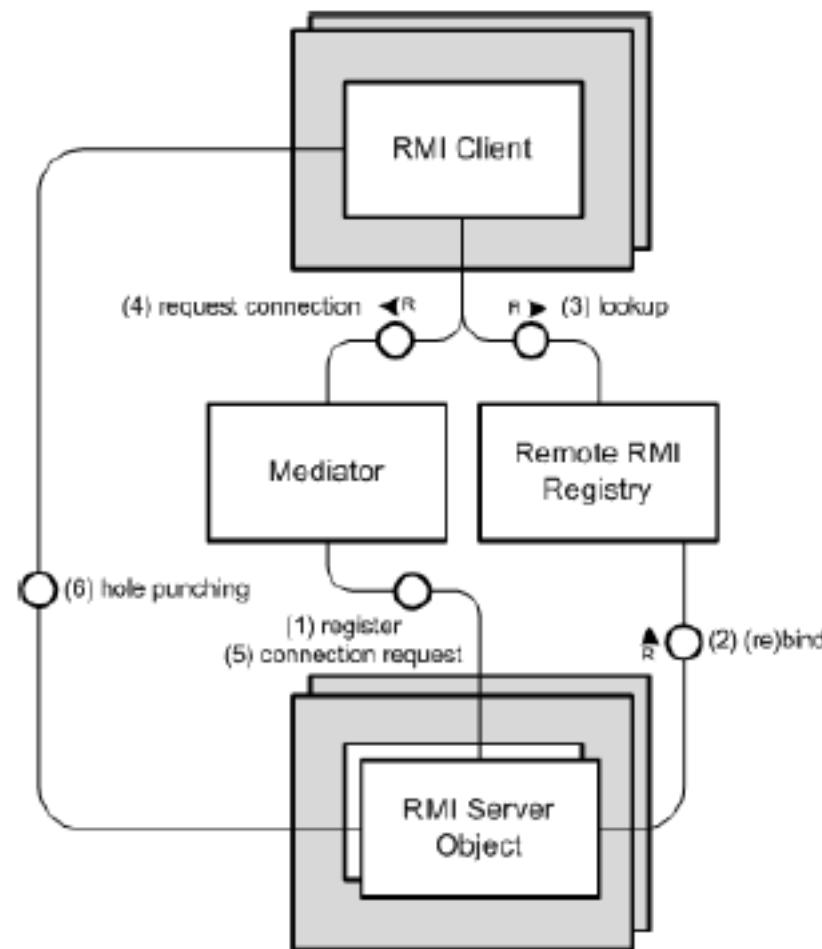
```
in.close();    out.close();    server.close();
```

Documentación y ejercicios

- API de Java: clases Socket y ServerSocket
- Tutorial
<http://download.oracle.com/javase/tutorial/networking/sockets>
- Descargar y ejecutar las clases para implementar el servicio *KnockKnock* de página del tutorial (“Writing a Client/Server Pair”)
 - En local
 - En remoto con otros compañeros

RMI

Remote Method Invocation

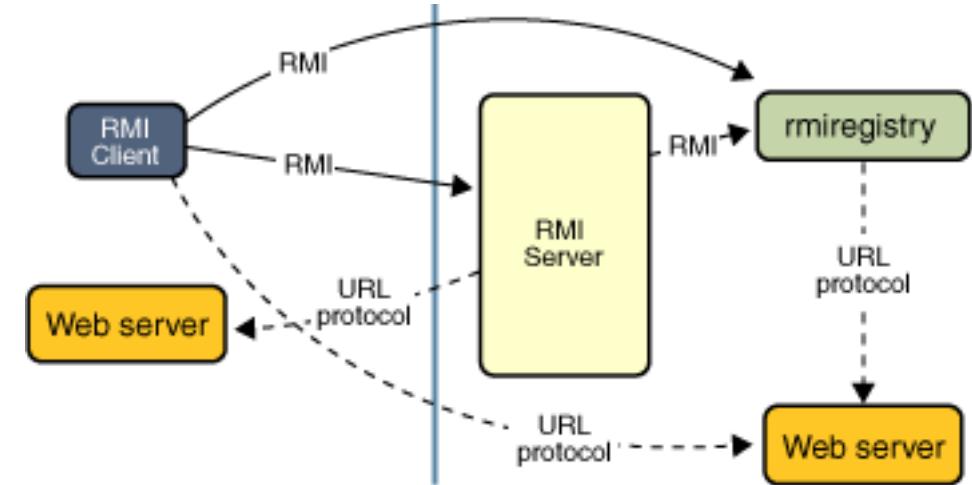


RMI

- **Remote Method Invocation:** permite que un objeto corriendo en una JVM use métodos de otro corriendo en otra JVM (local o *remota*)
 - Un paso más allá de los sockets
- RMI introducción:
 - <http://download.oracle.com/javase/tutorial/rmi/overview.html>
- Tutorial de implementación de un servicio:
 - <http://download.oracle.com/javase/tutorial/networking/sockets/clientServer.html>

Aplicación de Objetos Distribuidos

- Aplicación basada en RMI
- Dos fases fundamentales
 - Localizar objetos remotos
 - Registrados mediante el registro RMI
 - Pasados por referencia en invocaciones remotas
 - Comunicarse con objetos remotos
 - Gestionado por el servidor RMI, para el usuario es como llamar a métodos locales



Implementación

1. Definir **interfaz** con los métodos remotos
 - Será conocida por cliente y servidor
2. Implementar la clase que dará el servicio de la interfaz (**el servidor**)
3. Implementar el **cliente** que usará el servicio
4. Instanciar el servidor y registrarlo mediante un **stub**:
 - Referencia remota al servidor generada por RMI para el uso de los clientes

Interfaz

```
/**  
 * Interfaz para un servicio RMI de corredores de bolsa  
 * Debe heredar de java.rmi.Remote  
 * Debe manejar RemoteException en sus métodos  
 */  
public interface Corredor extends Remote  
{  
    String listarTitulos() throws RemoteException;  
    void comprar(String nombre, int cantidad) throws RemoteException;  
    void vender(String nombre, int cantidad) throws RemoteException;  
}
```

Sólo los métodos del objeto que se encuentren en una interfaz Remote serán accesibles vía RMI

La misma interfaz (idejalmente ya compilada y en un .jar) debe importarse en cliente y servidor

Servidor y Stub

```
/**  
 * Servidor de bolsa que implementa Corredor  
 * Debe implementar una interfaz de java.rmi.Remote  
 *  
 */  
public class Bolsa implements Corredor  
{  
    String listarTitulos() throws RemoteException  
    {...}  
    void comprar(String nombre, int cantidad) throws RemoteException;  
    {...}  
    void vender(String nombre, int cantidad) throws RemoteException;  
    {...}  
    //Cualquier otra función interna que sea necesaria  
}
```

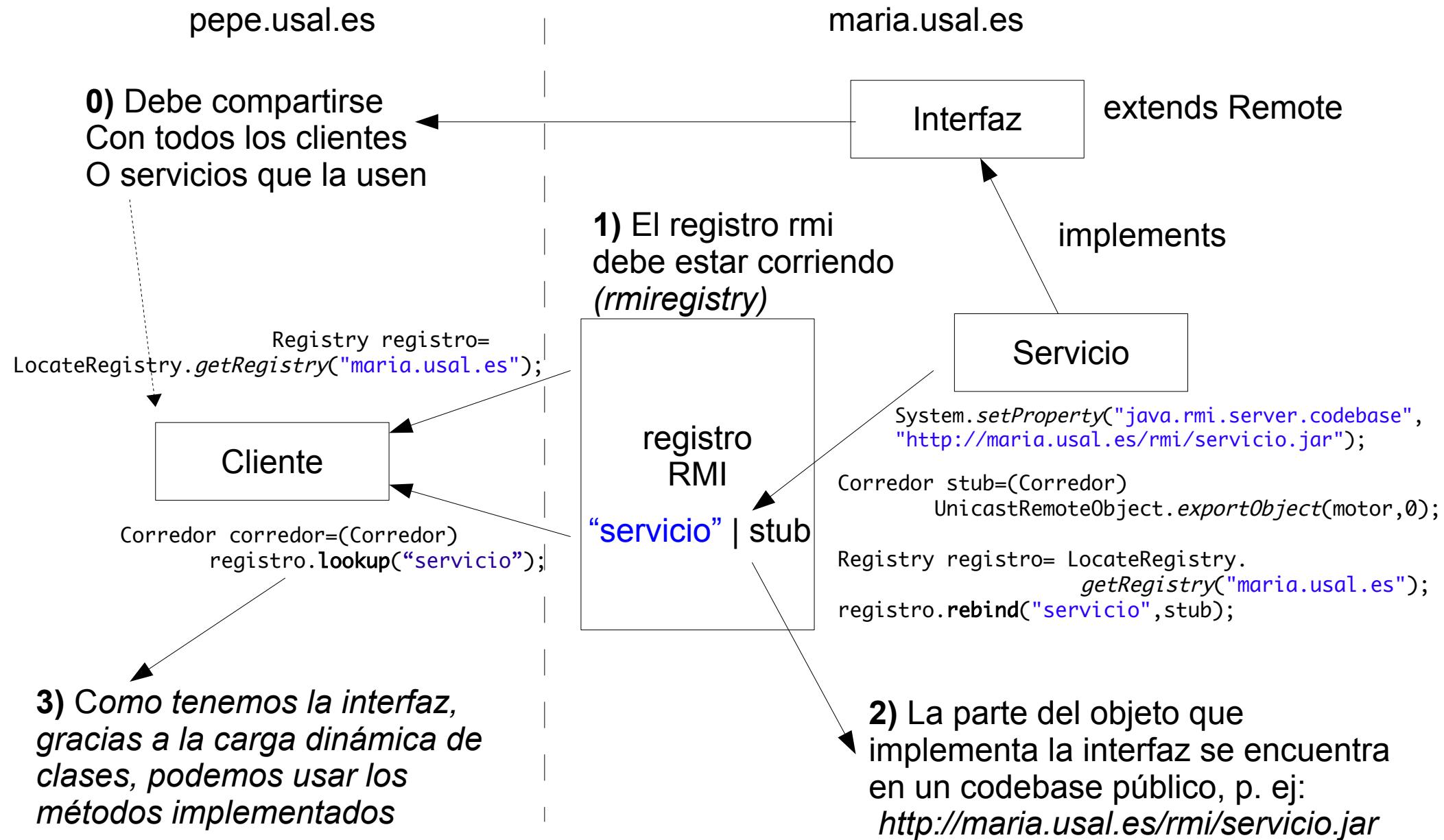
```
//Registramos un objeto Bolsa de nombre “LaBolsa”  
String nombre="LaBolsa";  
Corredor motor=new Bolsa();  
→ Corredor stub=(Corredor) UnicastRemoteObject.exportObject(motor,0);  
Registry registro=LocateRegistry.getRegistry();  
registro.rebind(nombre,stub);
```

Cliente

```
public class Cliente
{
    public static void main(String args[])
    {
        try
        {
            String nombre="LaBolsa";
            //Instanciar el registro RMI
            Registry registro=LocateRegistry.getRegistry(args[0]);
            //Instanciar un objeto de la clase del servidor
            Corredor corredor=(Corredor) registro.lookup(nombre);
            //Uso del servicio

            ...
        }
        catch (Exception e)
        {
            System.err.println("Excepción en el cliente de la bolsa:");
            e.printStackTrace();
        }
    }
}
```

Esquema



Servidor y seguridad

- Si queremos cargar o enviar clases *distintas* del API de Java, hay que implementar un **gestor de seguridad**

- Para evitar intrusiones de código maligno
- Para activar el gestor:

```
if (System.getSecurityManager()==null)  
    System.setSecurityManager(new SecurityManager());
```

- Podemos modificar la política de seguridad de Java con un fichero de permisos con líneas como:

```
grant { permission java.security.AllPermission; };
```

- Especificar el fichero con la opción de la JVM
 - -Djava.security.policy=grantFilePath

Codebase

- El **codebase** es un lugar desde el que se cargan clases en la JVM
- El CLASSPATH es un codebase local
- El codebase “remoto” se usa para acceder a clases desde applets o RMI, a través de urls
- Se puede modificar con el argumento de la JVM -Djava.rmi.codebase=”url”
- En RMI, nuestro codebase debe apuntar a las clases compiladas que se comparten
 - La interfaz o interfaces y cualquier clase que no esté en la API y sea argumento de las interfaces

Naming

- Podemos simplificar el uso de codebases en el registro RMI mediante la clase Naming
 - `Naming.lookup()` y `Naming.rebind()` funcionan como sus homónimos de `Registry`, pero no necesitamos obtener previamente el registro

Errores frecuentes

- En el servidor:
 - No haber iniciado el registro RMI
 - No tener definida el path al codebase de RMI
- En servidor y cliente
 - No utilizar la misma clases compiladas para la interfaz (generalmente lo mejor es usar un .jar)
 - Si enviamos clases no definidas en el API:
 - No tener implementada una política de seguridad

Procedimiento: interfaz

```
import java.rmi.*;  
  
public interface Corredor extends Remote  
{  
    String listarTitulos() throws RemoteException;  
    void comprar(String nombre, int cantidad) throws  
RemoteException;  
    void vender(String nombre, int cantidad) throws  
RemoteException;  
}
```

- La interfaz compilada debe ser **accesible al cliente y servidor**
 - Bien como las clases tal cual (.java)
 - O (mejor) como las clases compiladas (.class o .jar)

Procedimiento: interfaz

- Compilar y construir .jar

```
cd /home/usuario/workspace/rmiInterface/src  
javac rmiInterface/Corredor.java  
jar cvf bolsaInterface.jar rmiInterface/*.class
```

- Distribuir el .jar entre los servidores/clientes para que quieran implementar/usar el servicio
- O distribuir la carpeta con los .class o .java

Procedimiento: servidor

```
public class Bolsa implements Corredor
{
    public java.util.TreeMap<String, Integer> listado;

    String listarTitulos() throws RemoteException {...}
    void comprar(String nombre, int cantidad) throws RemoteException {...}
    void vender(String nombre, int cantidad) throws RemoteException {...}

    public static void main(String[] args)
    {
        try{
            System.setProperty("java.rmi.server.codebase",
                               "file:///PathAClass0JarDeInterfaz");
            Corredor motor=new Bolsa();
            Corredor stub=(Corredor) UnicastRemoteObject.exportObject(motor,0);

            Naming.rebind("//servidor:puertoRMI/nombreServicio", stub);
            System.out.println("La bolsa está en marcha.");
        } catch (Exception e) {e.printStackTrace();}
    }
}
```

Procedimiento: servidor

- Debe importar la **interfaz** a través del .jar
- Debe implementar los métodos de la **interfaz**
- El **registro RMI** debe estar corriendo antes de hacer llamadas a `rebind()`
- Debe hacer referencia al **codebase** para la interfaz (.jar o carpeta con binarios) a través de una url a fichero local (`file://`) o remoto (`http://`)
- Si la interfaz implica enviar o recibir como argumentos clases no incluidas API, debe implementar la **política de seguridad**

Procedimiento: cliente

```
public class Cliente
{
    public static void main(String args[])
    {
        try
        {
            Corredor corredor=(Corredor)
                Naming.lookup("rmi://servidor:puertoRMI/nombreServicio");
            System.out.println("Listado de titulos:");
            System.out.println(corredor.listarTitulos());
            System.out.println("Ahora vendo 10 de Danone.");
            corredor.vender("Danone",10);
        }
        catch (Exception e)    {e.printStackTrace();}
    }
}
```

Procedimiento: cliente

- Debe importar el .jar o las clases compiladas de la **interfaz**
- Hace referencia a la **máquina** donde está registrado el objeto remoto y al **nombre** con el que se ha registrado
- Si la interfaz implica enviar o recibir como argumentos clases no incluidas API, debe implementar la **política de seguridad**

Ejercicio

- Basándonos en los ejemplos anteriores, crear un sistema de bolsas
 - Cada pareja dará un servicio de bolsa, por ejemplo con nombres
 - IBEX, NASDAQ, Wall Street, Tokyo, Londres, Frankfurt
 - El resto seréis especuladores que se matan en el parqué a comprar y vender
 - El servicio de Bolsa, da igual cuál, debe proveer los servicios del Corredor de bolsa (ver arriba):
 - listarTitulos: devuelve una lista de los títulos de esa bolsa y la cantidad de acciones disponibles en venta
 - comprar y vender: de un título, la cantidad especificada

FAQ

- ¿Dónde puedo encontrar información adicional?
 - Generalmente, los tutoriales de Java sobre RMI son muy recomendables:
 - Sobre codebase:
<http://download.oracle.com/javase/1.4.2/docs/guide/rmi/codebase.html>
 - Sobre RMI
 - Sobre

FAQ

- En LocateRegistry.getRegistry(), obtengo el error:
 - java.rmi.ConnectException: Connection refused to host
 - Esto suele ocurrir si el registro RMI no se ha iniciado.
 - Se puede iniciar desde un terminal con
 - rmiregistry [port]
 - O desde java con
 - Runtime.getRuntime().exec("rmiregistry");
 - O, mejor, desde java con
 - LocateRegistry.createRegistry(int port);
 - Ojo con iniciar un registro cuando ya hay otro corriendo

FAQ

- En `registro.rebind()` tengo el error:
 - `java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:`
`java.lang.ClassNotFoundException: rmi.Corredor`
 - Esto ocurre porque el servidor no encuentra los codebase para las clases registradas en RMI
 - Hay que definir la ruta a los codebases para RMI de una de estas maneras:
 - `-Djava.rmi.server.codebase="urlClases"`
 - `System.setProperty("java.rmi.server.codebase", "file:/Users/rodri/Documents/workspace/assoo/bin/");`
 - Ojo: debe ser una url, apuntar a las clases compiladas (`/bin/`), y terminar en `/`
 - Ojo: necesitamos el codebase para hacer el stub, así que la llamada debe estar ANTES de ese punto

FAQ

- En el servidor o el cliente, tengo el error:
 - java.security.AccessControlException: `access denied (java.net.SocketPermission 127.0.0.1:1099 connect, resolve)`
 - Hemos activado el gestor de seguridad, pero tiene una seguridad muy alta/indadecuada
 - Crear un fichero `file.policy` con la política de la seguridad y enlazarlo con `-Djava.security.policy` como se vio en los ejemplos más arriba
 - La política de seguridad deberá implementarse en cliente y servidor
 - Recordad que la política de seguridad en principio sólo es necesaria si usamos como parámetro/valor de retorno en los métodos remotos algún objeto que no esté en la API

FAQ

- ¿Cómo funciona la clase que hace de interfaz?
 - Con RMI conseguimos llamar a un objeto desde otro ordenador, sin necesidad del código del objeto, pero sí de su interfaz. Por tanto, la clase interfaz (bien el código .java, bien compilada importando la carpeta con los .class o en un .jar), debe ser importada por el servidor y el cliente (ambos deben tener copia de esos ficheros).
 - Ojo, en caso de usar el código sin compilar, no funcionará si tenemos interfaces que no son idénticas en servidor y cliente, e incluso puede no funcionar aún siendo las mismas (si las versiones de la JVM son distintas, por ejemplo).

FAQ

- ¿Es necesario implementar siempre el servicio de seguridad?
 - No. La política de seguridad es sólo necesaria si los métodos de nuestra interfaz tienen como argumentos (o devuelven) clases que no estén en el API.
 - Si este es el caso (lo cual no es muy frecuente), tenemos que poner el código para el SecurityManager y establecer en un fichero la política de privacidad (ver el .pdf de la presentación para más información)

FAQ

- ¿Es necesario establecer el codebase?
 - Sí, el servidor debe decirnos dónde está el código base del objeto(s) remoto(s), mediante la propiedad `java.rmi.server.codebase`, que apunta a una url con `file://`, en cuyo caso los objetos remotos deberán estar en la misma máquina que el servicio, o con una url con `http://`, en cuyo caso podrán estar en máquinas distintas al servidor.
 - El codebase se puede establecer como argumento a la JVM (`-Djava.rmi.server.codebase=url`) o desde el propio programa
`(System.setProperty("java.rmi.server.codebase", "url"))`

FAQ

- En el laboratorio, en Fedora, tengo este error:
 - `java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:`
 - `java.lang.ClassCastException: java.io.ObjectStreamClass cannot be cast to java.lang.String`
- Es un error bastante puñetero, pues no nos da información relevante de lo que está pasando.
- El problema es que tenemos varias órdenes instaladas para ejecutar un servicio de registro RMI. En particular, en los Fedora del laboratorio tenéis dos órdenes para rmiregistry:
 - `[p1777026@labhp10 ~]$ whereis rmiregistry`
 - `rmiregistry: /usr/bin/rmiregistry /opt/jdk1.6.0/bin/rmiregistry`
- Y se ejecuta por defecto el de `/usr/bin`. Tendremos que correr el de la jdk para solventar este problema

FAQ

- En el laboratorio, en Fedora, el cliente falla con:
 - `java.rmi.ConnectException: Connection refused to host: 127.0.0.1;...`
 - Esto ocurre por una mala configuración en Fedora del `/etc/hosts`
 - 127.0.0.1 localhost labhp10
 - Es decir, labhp10 señala a 127.0.0.1, así que si el cliente accede via RMI a labhp10, está accediendo realmente a 127.0.0.1, que en el cliente se referirá a sí mismo y no al servidor.
 - El modo de solucionarlo es editar `/etc/hosts`:
 - 127.0.0.1 localhost
 - 172.20.2.21 labhp10
 - (para saber vuestra ip en fedora: `/sbin/ifconfig -a`)
 - Otra opción (quizás más inteligente) es publicar el servicio con la ip en vez de con el nombre

FAQ

- ¿Se pueden incluir en Java algunas de las tareas de terminal o de las opciones de la VM?
 - Sí, por ejemplo:
 - Puedes, en general, lanzar cualquier orden del sistema con `Runtime().getRuntime.exec()`
 - Puedes crear el registro RMI con
`Runtime.getRuntime().exec("rmiregistry")` o con `LocateRegistry.createRegistry()`
 - Puedes conocer tu IP con
`InetAddress.getLocalHost().getHostAddress()`
 - Puedes pasar propiedades como por ejemplo `java.rmi.server.codebase` con `System.setProperty("propiedad", "valor")`