# Remote Method Invocation (RMI) en Java

**Daniel Díaz Sánchez**
**Florina Almenárez**
**Andrés Marín**
Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid
dds@it.uc3m.es

# Contexto

- **Objetivos**
  - Conocer las abstracciones del API de socket que facilitan la comunicación cómo WebServices y RMI.

# Índice

- Introducción a RMI
- Conceptos básicos
  - Serialización
- Un vistazo rápido a RMI
- RMI



# Introducción a RMI: ¿por qué?

Básicamente

- Las comunicaciones en la red son algo más que intercambiar documentos entre máquinas, resolver consultas o leer el correo.
- En ocasiones, es interesante que una máquina remota realice un trabajo para nosotros. RMI puede ayudar a eso.



# Introducción a RMI : ¿por qué?

- Las clases de implementación de sockets TCP y UDP en Java nos permiten realizar cualquier programa cliente-servidor y distribuirlo en la red
- Sin embargo presentan una serie de inconvenientes:
  - El protocolo debe implementarse en clases diferentes (para clientes y servidores)
  - Es necesario trabajar a nivel de bytes transferidos (y como mucho utilizar clases derivadas de Reader para trabajar con caracteres)
  - Dificultad de mantenimiento y actualización de versiones
  - No está integrado dentro de la metodología de OO de Java
- Java nos ofrece RMI para paliar estas deficiencias



# Introducción a RMI: ¿qué es?

- Mecanismo de invocación de métodos de objetos remotos.
- No es original de Java otros antes lo han hehco: XDR+RPC, Corba
  - RMI es la manera en que Java
- Permite tratar objetos remotos como si fueran locales:
  - Siguiendo una metodología (definición de interfaces, extensión de Remote, UnicastRemoteObject, etc.)
  - utilizando facilidades adicionales (compilador rmic, rmiregistry, rmid, dgc)
- Permite incorporar la metodología de objetos en clases cliente-servidor
- No es necesario trabajar a nivel de bytes transferidos, podemos trabajar directamente con objetos.



# Introducción a RMI: ¿cómo?

- La invocación de un método en un objeto remoto se parece bastante a la invocación de métodos locales.
- Sin embargo los mecanismos internos son más complejos aplanamiento (marshalling) de argumentos y resultados...

...pero Java lo hace por nosotros

  - RMI no requiere modificación del lenguaje, ni del compilador, ni de la máquina virtual.
  - La invocación remota se obtiene por medio de nuevas bibliotecas y un compilador adicional "rmic".



# Conceptos básicos: Serialización

- El concepto más importante es el de Serialización y aplanado (marshalling) de los argumentos
  - En RMI, como en cualquier aplicación Java, llamaremos a objetos, recibiremos objetos.
  - Estos objetos hay que aplanarlos para enviarlos por la red
- Java tiene un marco genérico para convertir objetos y grupos de objetos a una representación externa que puede ser posteriormente leida por cualquier JVM.
- Este marco genérico es la serialización de objetos de java.



# Conceptos básicos: Serialización

### La serialización preserva grafos de objetos

- Consideremos una clase de árboles binarios:

```
class Node implements Serializable {

    Node() {}
    Node(Node left, Node right)
    {
        this.left = left ;
        this.right = right ;
    }

    private Node left, right ;

    }
```

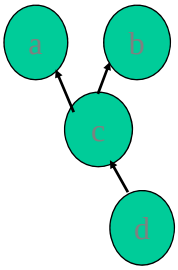
- Creamos una árbol, **d**:

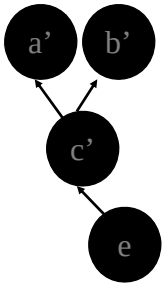
```
Node a = new Node();// Hoja
Node b = new Node();// Hoja
Node c = new Node(a, b) ;
Node d = new Node(c, null) ;
```



# Conceptos básicos: Serialización

Se reproduce el total del árbol original. Se recrean copias de los nodos originales a', b', c' también e. Se preserva el patrón de referencias.




**Escribir comenzando por la raíz:**

**out.writeObject(d) ;**

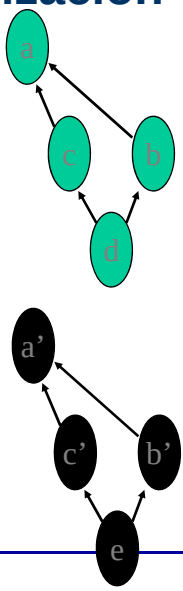
Posteriormente leemos un nodo

**Node e = (Node) in.readObject() ;**



# Conceptos básicos: Serialización

**Se preserva la integridad referencial**




•Este comportamiento no se limita a árboles.

•En este ejemplo tanto b como c referencian un mismo objeto a.

•Tras enviarlo por RMI preserva el patrón de referencias. Cuando se reconstruye el objeto raíz de su forma serializada se obtiene un único a', referenciado dos veces.

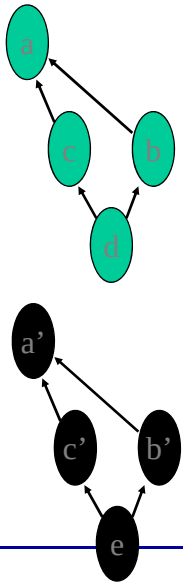
•La integridad de referencias se preserva en todos los objetos escritos a un mismo ObjectOutputStream.






# Conceptos básicos: Serialización

- `Serializable:` una clase debe implementar serializable para ser pasado a writeObject(). En caso contrario se levanta `NotSerializableException`.
- ¿Qué objetos son serializables?
  - Los arrays son serializables si lo son sus elementos.
  - La mayoría de las clases en la biblioteca estánar de Java son serializables.
  - Por ejemplo, clases contenedoras complejas como HashMap pueden ser serializadas (si los objetos que almacenan lo son) y ser utilizados como argumentos y resultados en métodos RMI.



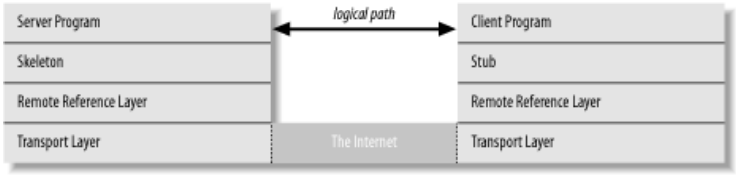






# RMI: implementación

- La mayoría de los paquetes con los que trabajar se encuentran en:
  - java.rmi: define las clases, interfaces y excepciones visibles desde el cliente.
  - java.rmi.server: define las clases, interfaces y excepciones visibles desde el servidor.
  - java.rmi.registry: define las clases, interfaces y excepciones del servicio que permite localizar y nombrar los objetos remotos





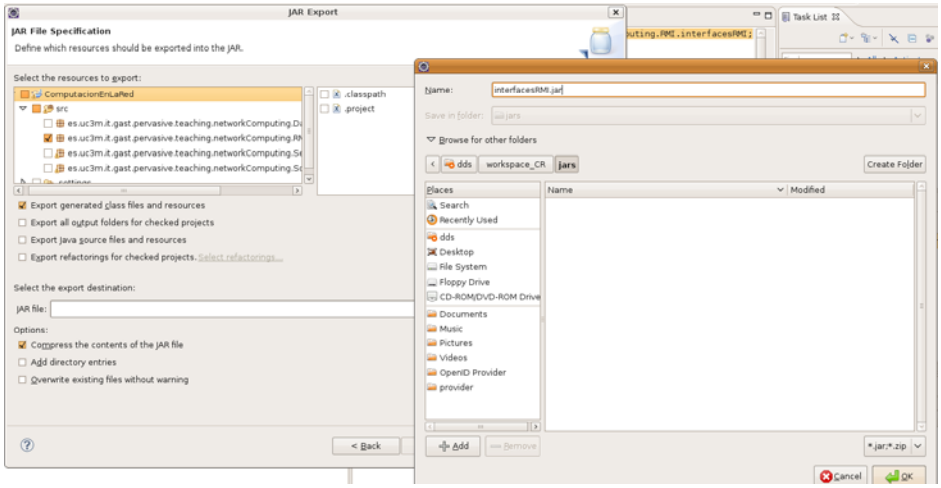

# RMI: implementación

- El interfaz de nuestro objeto remoto
  - Debemos definir un interfaz con los métodos del objeto remoto
  - Para ello, declaramos un interfaz que extienda java.rmi.remote
    - Este interfaz no obliga a implementar método alguno
    - Únicamente sirve para indicar que se trata de un objeto remoto

Compilaremos este objeto y lo meteremos en un jar (interfacesRMI.jar): sobre el paquete, click derecho, import/export…

```
package crRMI.interfacesRMI;
import java.rmi.*;
public interface Calculadora extends Remote {
    public int suma(int a, int b) throws RemoteException;
    public int multiplica(int a, int b) throws RemoteException;
}
```

# RMI: implementación



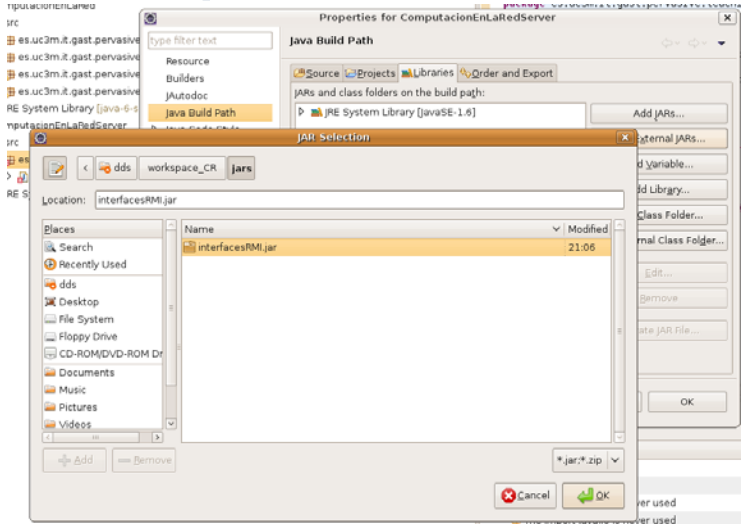



# RMI: implementación Servidor

- El servidor (la máquina que contiene el objeto remoto)
  - Una vez hemos definido el interfaz, vamos a crear una clase que lo implemente
  - La llamaremos CalculadoraImpl
  - Dicha clase deberá extender la clase UnicastRemoteObject
    - Implementa los métodos necesarios para permitir la invocación remota
    - Hace marshaling (aplanado) de los datos (recibidos y enviados)
    - El objeto solo vive durante el tiempo que lo hace el servidor
  - Podríamos hacer un objeto normal (no extiende UnicastRemoteObject) y luego hacer UnicastRemoteObject.exportObject( )
  - Podríamos extender también la clase java.rmi.activation.Activatable
    - Permite recuperar un objeto (se guarda hasta que se vuelva a usar)



# RMI: implementación Servidor






# RMI: implementación Servidor

Implementacion de calculadora

```
package crRMI.implementacionesRMI;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

import es.uc3m.it.gast.pervasive.teaching.networkComputing.RMI.interfacesRMI.calculadora;

public class CalculadoraImp extends UnicastRemoteObject implements calculadora {

    public CalculadoraImp() throws RemoteException {
           super();
    }

    @Override
    public int multiplica(int a, int b) throws RemoteException {
           return a*b;
    }

    @Override
    public int suma(int a, int b) throws RemoteException {
           return a+b;
    }
```

# RMI: implementación Servidor

- El servidor (la máquina que contiene el objeto remoto)
  - Ahora necesitamos una clase que instancie la implementación de la clase CalculadoraImpl
  - Al instanciar CalculadoraImpl, dado que ésta extiende UnicastRemoteObject, el objeto se exporta por RMI
  - Para poderlo encontrar, necesitamos hacer uso de un sistema de nombres, como un DNS.
    - Para eso usaremos el RMI Registry que permite dar un nombre al objeto y que los clientes lo encuentren
    - Usaremos la clas Naming (RMI registry) para incorporar nuestro objeto al Registro



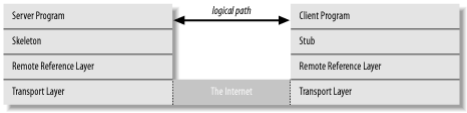

# RMI: implementación Servidor

Servidor, registra el objeto en el Registry

```
package es.uc3m.it.gast.pervasive.teaching.networkComputing.RMI.servidorCalculadora;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;

import crRMI.implementacionesRMI.CalculadoraImp;
public class ServidoCalculadora {

          public static void main(String[] args) {
           try {
                CalculadoraImp c = new CalculadoraImp( );
                Naming.rebind("calculadora", c);
                System.out.println("El servidor calculadora está preparado.");
              }
               catch (RemoteException rex) {
                System.out.println("Excepcion en CalculadoraImp.main: " + rex);
              }
              catch (MalformedURLException ex) {
                System.out.println("MalformedURLException " + ex);
              }
    }

}
```


# RMI: implementación Cliente

- El cliente (el que hace uso del objeto)
  - El cliente no necesita la implementación del objeto (CalculadoraImp) ya que está en el servidor
  - Pero necesita un mecanismo que le indique como interactuar con él. Este mecanismo son los stubs.




  - En las versiones de Java > 1.5, siempre que la implementación extienda UnicastRemoteObject no es necesario generarlos
  - En otras versiones o cuando el objeto no extiende UnicastRemoteObject hay que generarlos con RMIC



# RMI: implementación Cliente

- rmic es un generador de stubs, y es el único compilador específico de RMI.
- La entrada de rmic es una clase que implementa remote, compilada normalmente (con javac).
- rmic genera una clase que implementa las mismas interfaces remotas que la clase original.
- Los métodos de la nueva clase contienen código para enviar los argumentos y recibir los resultados de un objeto remoto cuya dirección está almacenada en la instancia del stub.



# RMI: implementación Cliente

- ¿Qué tenemos hasta el momento?
  - Proyecto1: Interfaz que describe nuestro objeto
    - crRMI.interfacesRMI.calculadora
    - Compilado y almacenado en interfacesRMI.jar
  - Proyecto2:
    - Clase que implementa el objeto
      - crRMI.implementacionesRMI.calculadoraImp
    - Clase que instancia el objeto remoto y lo introduce en el Registry
      - crRMI.servidorCalculadora. ServidorCalculadora



# RMI: implementación Cliente

- Generamos los stubs sobre la clase que implementa el objeto remoto
  - Posicionados en el directorio donde está compilado el servidor

```
dds@saml:~/workspace_CR/ComputacionEnLaRedServer/bin$ ls
   crRMI/implementacionesRMI

crRMI/implementacionesRMI:

CalculadoraImp.class
```

  - Ejecutamos RMIC

```
rmic   -classpath   .:/home/dds/workspace_CR/jars/interfacesRMI.jar
   crRMI.implementacionesRMI.CalculadoraImp
```



# RMI: implementación Cliente


- Creamos un jar con el stub para el cliente
  - El cliente va a necesitar InterfacesRMI.jar para conocer cómo es el objeto remoto
  - El stub para saber cómo comunicarse con él

```
dds@saml:~/workspace_CR/ComputacionEnLaRedServer/bin$
   jar                cf                CalculadoraImpStub.jar
crRMI/implementacionesRMI/CalculadoraImp_Stub.class
```

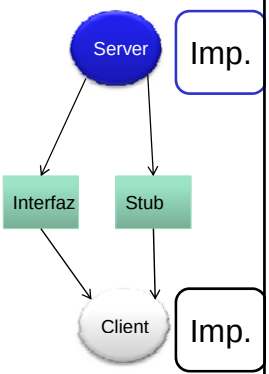
- Lo copiamos a una carpeta junto con InterfacesRMI.jar



# RMI: implementación Cliente

Hora podemos desarrollar el cliente de forma independiente al servidor (siempre que no cambie el interfaz)

- Interfaz que describe nuestro objeto
  - es.uc3m.it.gast.pervasive.teaching.networkComputing.RMI.interfacesRMI.calculadora
  - Compilado y almacenado en interfacesRMI.jar
- Clase que implementa el objeto
  - es.uc3m.it.gast.pervasive.teaching.networkComputing.RMI.implementacionesRMI.calculadoraImp
- Clase que instancia el objeto remoto y lo introduce en el Registry
  - es.uc3m.it.gast.pervasive.teaching.networkComputing.RMI.servidorCalculadora. ServidorCalculadora
- Stub compilado con RMIC en CalculadoraImp_stub.jar



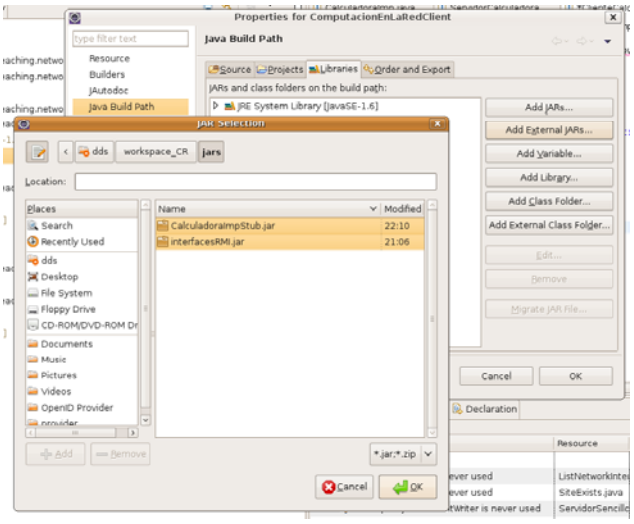



# RMI: implementación Cliente

- El cliente deber obtener una referencia al objeto remoto mediante una búsqueda en el registro.
- A partir de ese momento lo usará como un objeto local
- Hay que incluir el jar con el stub y el jar con el interfaz en el path



# RMI: implementación Cliente






# RMI: implementación Cliente

Cliente RMI

```
package crRMI.clienteRMI;

import java.rmi.Naming;

import es.uc3m.it.gast.pervasive.teaching.networkComputing.RMI.interfacesRMI.calculadora;

public class ClienteCalculadora {

    public static void main(String args[]) {

          try {
                    Object o = Naming.lookup("rmi://localhost:1099/calculadora");
                    calculadora calc = (calculadora) o;
                    System.out.println("10 + 20 = " + calc.suma(10, 20));
                    System.out.println("10 * 20 = " + calc.multiplica(10, 20));

          } catch (Exception ex) {
                    ex.printStackTrace();
          }
    }
}
```



# RMI: prueba

- Para ejecutarlo, abrimos dos terminales uno en la ruta del cliente y otro en la ruta del servidor. De modo que veamos:

```
dds@saml:~/workspace_CR/ComputacionEnLaRedClient/bin$ tree
.
`-- crRMI
    `-- clienteRMI
        `-- ClienteCalculadora.class

dds@saml:~/workspace_CR/ComputacionEnLaRedServer/bin$ tree
.
|-- crRMI
|   |-- implementacionesRMI
|   |   `-- CalculadoraImp.class
|   `-- servidorCalculadora
|       `-- ServidorCalculadora.class
```



# RMI: prueba

- Creamos dos scripts para que sea más sencillo

**server**

```
java -cp .:/home/dds/workspace_CR/jars/interfacesRMI.jar
-Djava.security.policy=/home/dds/jars/policy.all
-Djava.rmi.server.codebase=
   file:///home/dds/workspace_CR/jars/interfacesRMI.jar
crRMI.servidorCalculadora.ServidorCalculadora
```

**client**

```
java -cp .:/home/dds/workspace_CR/jars/interfacesRMI.jar
-Djava.security.policy=/home/dds/jars/policy.all
-Djava.rmi.server.codebase=
   file:///home/dds/workspace_CR/jars/interfacesRMI.jar
crRMI.clienteRMI.ClienteCalculadora
```



# RMI: prueba

- Obtenemos

```
dds@saml:~/workspace_CR/ComputacionEnLaRedServer/bin$ ./server
El servidor calculadora está preparado.

dds@saml:~/workspace_CR/ComputacionEnLaRedClient/bin$ ./client
10 + 20 = 30
10 * 20 = 200
```



# El paquete java.rmi

- Este paquete contiene las clases vistas por el cliente
  - El interfaz Remote
  - La clase Naming
  - La clase RMISecurityManager



# java.rmi: interfaz Remote

java.rmi.Remote

Interfaz

```
public interface myInterfazRemoto extends java.rmi.Remote
```

Descripción

Este interfaz no declara métodos únicamente se emplea para "marcar" un objeto. Es decir, informar a Java que ese objeto es remoto

Ejemplo

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



# java.rmi: clase Naming

java.rmi.Naming

## Métodos

```
public static String[] list(String url) throws RemoteException,
    MalformedURLException
public static Remote lookup(String url) throws RemoteException,
    NotBoundException, AccessException, MalformedURLException
public static void bind(String url, Remote object) throws RemoteException,
    AlreadyBoundException, MalformedURLException, AccessException
public static void unbind(String url) throws RemoteException, NotBoundException,
    AlreadyBoundException, MalformedURLException, AccessException
public static void rebind(String url, Remote object) throws RemoteException,
    AccessException, MalformedURLException
```

## Descripción

**list** devuelve una lista con las URLs que actualmente están asociadas a un objeto.
**lookup** devuelve el objeto remoto asociado con la URL.
**bind** asocia un objeto remoto a una URL. **unbind** lo elimina.
**rebind** se comporta igual que bind salvo que fuerza la asociación incluso si la URL está en uso.



# java.rmi: clase Naming

## Ejemplo

```
package crRMI.test;
import java.rmi.*;

public class DescubreObjetos {
  public static void main(String[] args) {
    // puerto RMI por defecto
    int port = 1099;
    // si no especifica servidor tomamos "localhost"
    String host = "localhost";
    if (args.length == 0) {
         host = args[0];
    }
    if (args.length > 1) {
      try {
        port = Integer.parseInt(args[1]);
        if (port <1 || port > 65535) port = 1099;
      }
      catch (NumberFormatException ex) {}
    }
```



# java.rmi: clase Naming

Ejemplo

```
 //creamos la URL
    String url = "rmi://" + host + ":" + port + "/";
    try {
      String[] objetosRemotos = Naming.list(url);
      for (int i = 0; i < objetosRemotos.length; i++) {
        System.out.println(objetosRemotos[i]);
      }
    }
    catch (RemoteException ex) {
      System.err.println(ex);
    }
    catch (java.net.MalformedURLException ex) {
      System.err.println(ex);
    }
  }
}
dds@saml:~/workspace_CR/ComputacionEnLaRedClient/bin$ java
    crRMI.test.DescubreObjetos localhost 1099
//localhost:1099/calculadora
```



# java.rmi.registry

- Este paquete permite a un cliente encontrar un objeto en un servidor remoto
  - El interfaz Registry: lo implementa la clase Naming. Dispone de los métodos list, lookup, bind, unbind, rebind…
  - La clase LocateRegistry: permite encontrar el registro antes de interactuar con él
- Para interactuar con el registro se utiliza la clase Naming



# java.rmi.registry: clase LocateRegistry

java.rmi.registry.LocateRegistry

Métodos

```
public static Registry getRegistry( ) throws RemoteException
public static Registry getRegistry(int port) throws RemoteException
public static Registry getRegistry(String host) throws RemoteException
public static Registry getRegistry(String host, int port) throws
   RemoteException
```

Descripción

```
getRegistry( ) devuelve el registro en localhost, puerto estandard 1099
getRegistry(int port) devuelve el registro en localhost:port
getRegistry(String host) devuelve el registro en un host remoto
   indicando el host (se supone puerto estandar 1099)
public static Registry getRegistry(String host, int port) devuelve el
   registro en la máquina host y puerto port
```



# java.rmi.server

- Proporciona las herramientas para la parte de servidor. Entre ellas:
  - La clase RemoteObject
  - La clase RemoteServer
  - La clase UnicastRemoteObject



# java.rmi.server: la clase RemoteObject

java.rmi.server.RemoteObject

Declaración y métodos

```
public abstract class RemoteObject extends Object implements Remote,
   Serializable
toString( ) , hashCode( ), clone( ), equals(), getRef()
```

Descripción

Todo objeto remoto es una subclase de RemoteObject. La clase está pensada como clase de la que heredar. Si se creara un objeto remoto directamente, habría que implementar todos los métodos toString(), hashCode()…



# java.rmi.server: la clase RemoteServer

java.rmi.server.RemoteServer

Declaración y métodos

```
public abstract class RemoteServer extends RemoteObject
Cons:
protected RemoteServer( )
protected RemoteServer(RemoteRef r)
Métodos
public static String getClientHost( ) throws ServerNotActiveException
public static void setLog(OutputStream out)
```

Descripción

Es una superclase abstracta para objetos remotos como UnicastRemoteObject. No se instancia ni extiende, típicamente se recurre a UnicastRemoteObject.



## java.rmi.server: la clase UnicastRemoteObject

java.rmi.server. UnicastRemoteObject

Declaración y métodos

```
public abstract class RemoteServer extends RemoteObject
Cons:
protected UnicastRemoteObject( ) throws RemoteException
protected UnicastRemoteObject(int port) throws RemoteException
Métodos
public static Remote exportObject(Remote r, int port)
public static RemoteStub exportObject(Remote r) throws RemoteException
```

Descripción

Es una subclase de RemoteObject. Para crear un objeto remoto, se extiende la clase UnicastRemoteObject.



## Codebase y seguridad

- Codebase: permite crear objetos dinamicamente en el cliente si no se conoce el interfaz al compilarlo
  - Se puede proporcionar una URL de donde bajar el jar con el interfaz
- Seguridad: en la mayoría de las ocasiones, encontraremos problemas de seguridad a menos que establezcamos una política



# java.rmi.server.codebase

- Necesitamos una manera de anotar serializaciones con los URLs apropiados.
- La opción más directa es utilizando la propiedad java.rmi.server.codebase en la JVM del objeto serializado.
- El valor de la propiedad es una URL con el codebase apropiado.
- Las clases de serialización de RMI leen la propiedad y embeben la URL en la representación serializada de los objetos.
  - Si esta JVM obtuvo el código a su vez de un servidor web, embeberían la URL de la que lo obtuvieron..



# java.rmi.server.codebase

- En el ejemplo:

```
java -Djava.rmi.server.codebase=
file:///home/dds/workspace_CR/jars/interfacesRMI.jar
```

- El valor de java.rmi.server.codebase será:

```
file:///home/dds/workspace_CR/jars/interfacesRMI.jar
```

  - Esta URL se embebe en los streams de serialización creados por el programa HelloServer.
- Si se crea algún objeto por deserialización en otra máquina y no se encuentra una copia local de su código, se solicita automáticamente al servidor lm001 en el directorio especificado.



# Security Managers

- Último paso necesario, la seguridad.
- Antes de que una aplicación pueda cargar código dinámicamente, es necesario poner un gestor de seguridad apropiado.
- Esto requiere la definición de una política de seguridad (security policy).



# Security Managers

- En una aplicación RMI, si no hay un gestor de seguridad, los stubs y las clases se cargan únicamente del CLASSPATH local.
- Para permitir carga dinámica es necesario ejecutar el método del sistema:

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

  al comienzo de la ejecución del programa.
- Es necesario en cualquier aplicación que requiera descargar algún código (incluyendo los stubs RMI).
- También es necesario definir una propiedad: java.security.policy
  - En los casos más simples está propiedad se necesita para los clientes, igual que java.rmi.server.codebase se necesita para los servidores.



# Security Managers

- La política de seguridad más simple que podemos definir es un fichero de texto plano que especifique:

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

- Esta política permite al código descargado de la red hacer virtualmente lo mismo que puede hacer el usuario actual:
  - Leer, escribir y borrar filcheros; abrir leer y escribir sockets; ejecutar comandos en la máquina local, etc.
  - Es una política muy peligrosa pues solo aconsejada para primeros desarrollos en RMI.
  - Nunca debemos dar tales permisos a código que no sea de nuestra confianza.



# Security Managers

- Si el fichero con la política se denomina **policy.all**, el cliente se ejecutaría así:

```
java  –Djava.security.policy=policy.all  Client
```

- También podemos poner la propiedad en el programa con **System.setProperty()**.
- Seguiremos viendo más a fondo las políticas de seguridad y los gestores de seguridad.