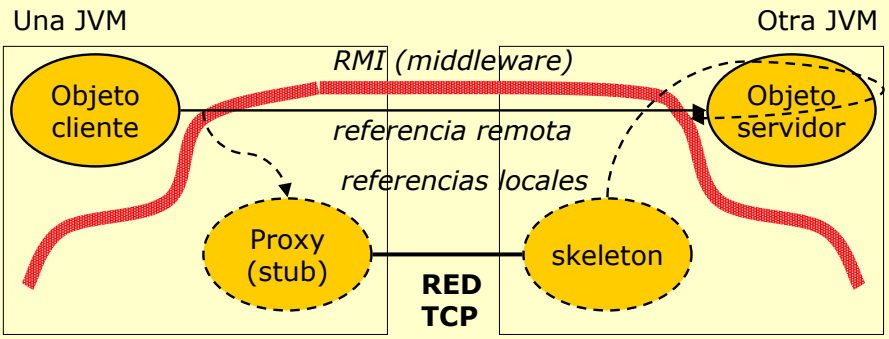
Una introducción a Java RMI

César Llamas Bello

Estas transparencias no hubieran sido posible sin el tutorial de José M. Vidal

Swearingen Engineering Center, University of South Carolina, Columbia

Introducción



- Java RMI (*Remote Method Invocation*) es una extensión de Java que permite la programación de objetos distribuidos.

Introducción

- RMI proporciona invocación de métodos dinámica
  - Un cliente puede invocar un servidor sin conocer qué métodos contiene el servidor
- La referencia remota permite determinar la naturaleza del objeto
  - Si es local o remoto
  - Si es remota, si se arranca automáticamente o necesita ser inicializado

Introducción

- Las referencias a objetos remotos son independientes de plataforma
  - Pero hay que usar Java en ambos extremos
- Los objetos remotos se recolectan en el GC por conteo de referencias
- El programa "`rmiregistry`" proporciona acceso a los objetos remotos.

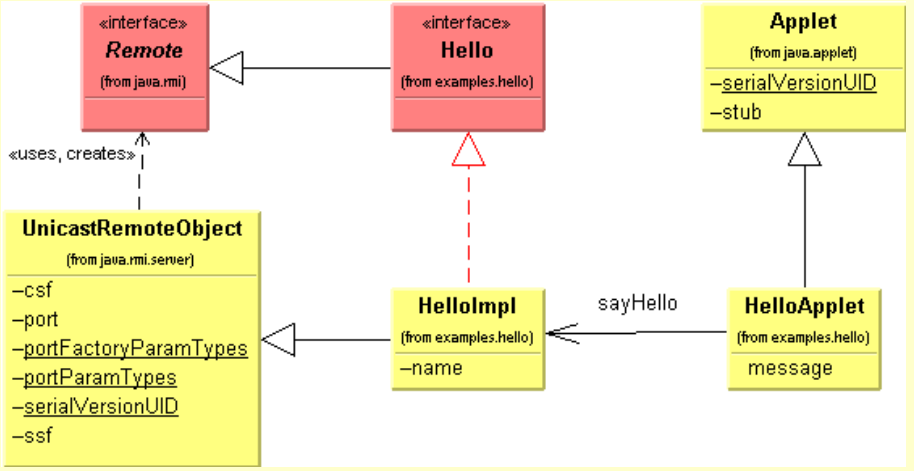
Una aplicación que dice "Hello" desde lejos

- La aplicación que dice "`Hello`" se compone de los siguientes elementos:
  - `HelloApplet`: un applet Java que invoca un método remoto que implementa la interfaz...
  - `Hello`: que se declara como remota, y según la cual...
  - `HelloImpl`: crea un objeto sobre el que se puede invocar el método "`sayHello( )`"
- El applet puede lanzarse desde una página web o el "`appletviewer`".

Una aplicación que dice "Hello" desde lejos

- Lo primero que hay que hacer es definir la interfaz `Hello`.
- A continuación se implementa esta interfaz. En este caso mediante `HelloImpl`.
- También hay que tener un programa que use el objeto remoto, como `HelloApplet`.
- Se compila la aplicación
- Se despliega (*deployment*) (fase compleja)
- Y se ejecuta.

Diagrama de clases de la aplicación completa



Hello.java

```
package examples.hello;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

- Todos los métodos deben lanzar `RemoteException`
  - puesto que la llamada puede fallar debido a la red
- Debe ser `public`
- El objeto remoto debe ser del tipo de la interfaz,
  - **no** el tipo de la implementación (`HelloImpl`)

Hello.java (comentario)

- Los argumentos y valores de retorno pueden ser de cualquier tipo.
  - Pero si un argumento o valor de retorno es un objeto debe implementar `java.io.Serializable`
- Los objetos locales se pasan por copia (*call-by-value*).
- Los objetos remotos se pasan por referencia (en realidad, una referencia a un stub).
  - Muy interesante porque permite obtener referencias a otros objetos remotos a partir del primero.

HelloImpl.java

```
package examples.hello;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject  implements Hello {
   public HelloImpl() throws RemoteException { super(); }
   public String sayHello() { return "Hello World!"; }
   public static void main(String args[]) {
        // Crea e instala un gestor de seguridad
        if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager()); }
        try {
                HelloImpl obj = new HelloImpl();
                // Enlaza este objeto instancia bajo el nombre "HelloServer"
                Naming.rebind("HelloServer", obj);
                System.out.println("HelloServer bound in registry");
        } catch (Exception e) {
                System.out.println("HelloImpl err: " + e.getMessage());
                        e.printStackTrace();
} } }
```

HelloImpl.java (desglose)

```
package examples.hello;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject
      implements Hello {
  public HelloImpl( ) throws RemoteException { super( ); }
  public String sayHello( ) { return "Hello World!"; }
  public static void main( String args[ ] ) { .........
  }
}
```

import java.rmi.*;

HelloImpl.java (comentario)

- Esta clase sirve para dos cosas:
  - Implementa el objeto remoto.
  - Sirve para lanzar el objeto, es decir es el servidor del objeto (en main).
  - Estas dos funciones pueden separarse.
- La implementación deberá implementar todos los métodos de la interfaz o sería una clase abstracta.
- Podría haberse hecho como una clase activable (*activatable*) de modo que se lanzaría el objeto cuando se requiriera desde la aplicación remota.

HelloImpl.java (comentario)

- Extiende `UnicastRemoteObject` de modo que utiliza el transporte por defecto de RMI
  - El transporte por defecto es TCP/IP.
  - Además permite que el objeto perdure.
- El constructor es el mismo que para una clase no remota.
- El constructor debe lanzar `RemoteException`, al igual que la clase base.

HelloImpl.java (desglose)

```
public static void main(String args[]) {
  // Crea e instala un gestor de seguridad
  if (System.getSecurityManager() == null) {
    System.setSecurityManager(new
          RMISecurityManager());
  }
  try {
    HelloImpl obj = new HelloImpl();
    // Enlaza el objeto bajo el nombre "HelloServer"
    Naming.rebind("HelloServer", obj);
    System.out.println("HelloServer enlazado en el
      registro");
  } catch (Exception e) {
    System.out.println("HelloImpl err: " +
          e.getMessage());
    e.printStackTrace();
  }
```

HelloImpl.java (comentario)

- Los objetos remotos deber ser **exportados**.
  - Hace que el objeto escuche en un puerto anónimo los mensajes del stub (resguardo).
  - La exportación es automática si se extiende: `RemoteObject` o `Activatable`.
  - En caso contrario deberá invocarse: `UnicastRemoteObject.exportObject` o `Activatable.exportObject`.
- Se pueden implementar otros métodos pero solo serán invocables desde la JVM que corre el servicio.

HelloImpl Seguridad y Registro

- Debe crearse e instalarse un gestor de seguridad `SecurityManager`.
  - Si el cliente es una aplicación (y no un applet) podría requerir otro.
  - El gestor de seguridad (`SecurityManager`) garantiza la política de acceso a los recursos.
- Cuando el servidor arranca (`main`) crea una o más instancias del objeto remoto.
  - Puesto que se exporta automáticamente el objeto, queda ya listo para recibir mensajes.

HelloImpl Seguridad y Registro

- Procedemos a registrar el objeto remoto con `Naming.rebind("//host/objectname", obj)`
  - Ojo, ¿porqué `rebind()`? La clase implementa `UnicastRemoteObject`, y ya se exporta y registra nada más crearlo.
- El registro permite al invocador ubicar el objeto por nombre.
  - Si se omite el host, se supone que es localhost.

HelloImpl Seguridad y Registro

- En la exportación **la referencia al objeto se substituye por la referencia al stub**.
  - Si no no habría forma de que se enlazara el stub con el objeto real (`obj`).
- Por razones de seguridad, una aplicación solo puede enlazar y desenlazarse con el registro que corre en la máquina local.
  - Se pueden hacer búsquedas (lookups) sobre cualquier host.

HelloApplet.java

```
package examples.hello;
import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {
   String message = "blank";
   // "obj" es el identificador que usaremos para referirnos al objeto
   // remoto que implementa la interfaz "Hello"
   Hello obj = null;
   public void init() {
      try {
         obj = (Hello)Naming.lookup("//" + getCodeBase().getHost() +
                                    "/HelloServer");
         message = obj.sayHello();
      } catch (Exception e) {
         System.out.println("HelloApplet exception: " + e.getMessage());
         e.printStackTrace();
   }}
   public void paint(Graphics g) {
      g.drawString(message, 25, 50);
}}
```


HelloApplet.java (comentario)

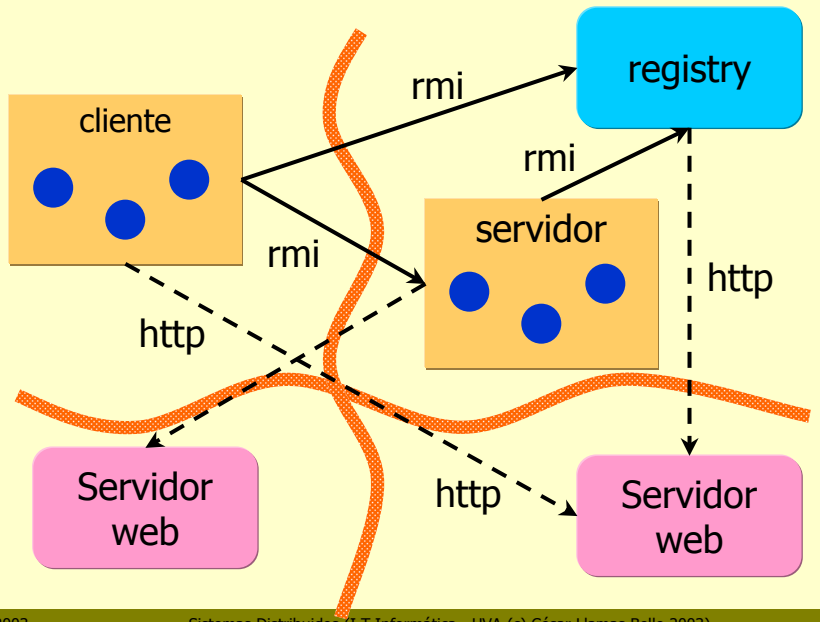
- La función `lookup` hace dos cosas:
  1. Construye una instancia del stub del registro para poder acceder al registro (¡que también es remoto!).
  2. Usa este stub para invocar al método lookup del registro.
- Al invocar al método `sayHello()` ocurre:
  1. Se envía una invocación remota al servidor vía stub.
  2. RMI serializa la cadena "`Hello world`" y la envía.
  3. RMI la des-serializa en el cliente y retorna una cadena con este contenido.

hello.html

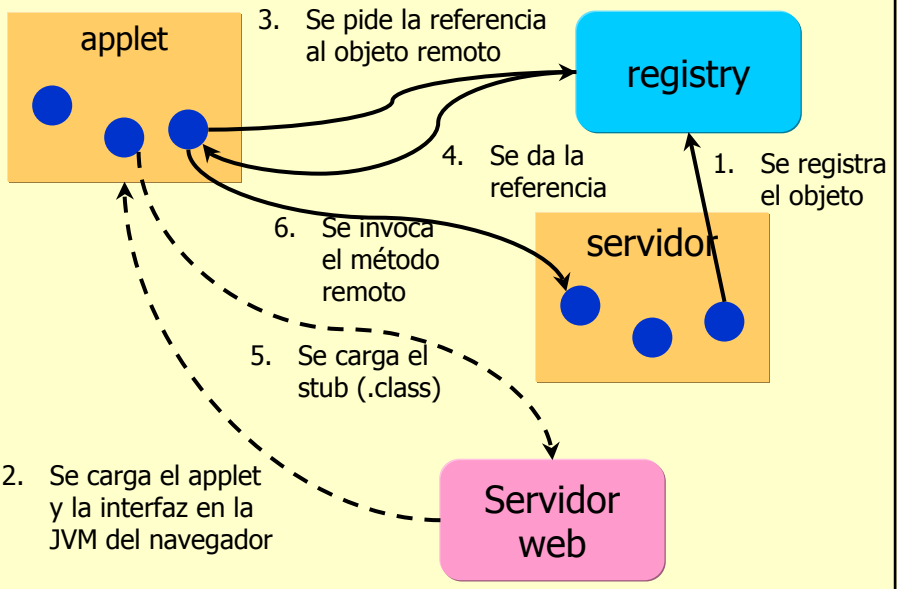
```
<HTML>
<head>
  <title>Hello World</title>
</head>
<body>
  <center> <h1>Hello World</h1> </center>
  The message from the HelloServer is:
  <p> <applet codebase="."
  code="examples.hello.HelloApplet" width=500
  height=120> </applet>
</body>

</HTML>
```

- El codebase es el directorio desde donde el applet descargará los archivos `.class`.

Protocolos involucrados en la aplicación



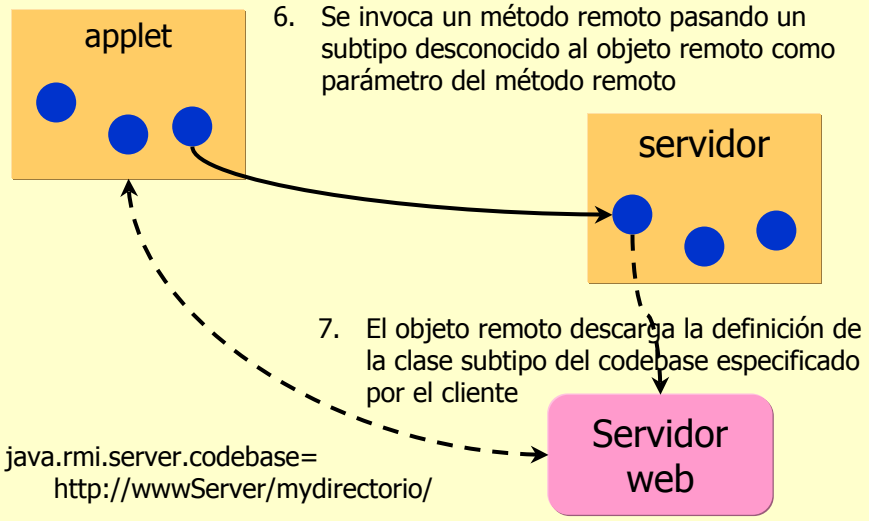
Pasos involucrados en "esta" aplicación



Pasos involucrados en "esta" aplicación

- El objeto servidor y el servidor web corren en la misma máquina
  - Aunque pudiera no ser así
- El valor de codebase (URL) representa el lugar de donde se puede descargar código
  - Apunta al lugar del stub
  - Y de la interfaz remota, en este caso

Pasos involucrados en cualquier aplicación



Pasos involucrados en cualquier aplicación

- Cuando se pasa un objeto remoto como argumento:
  - Si es un tipo primitivo, nada.
  - Si es un objeto cuya clase está en el CLASSPATH del objeto remoto se carga de allí.
  - Si no está en el CLASSPATH, o bien:
    - Es una implementación de la interfaz declarada como parámetro del método, o
    - Es una subclase de la clase declarada como parámetro del método
    - En cualquier caso se carga del codebase.

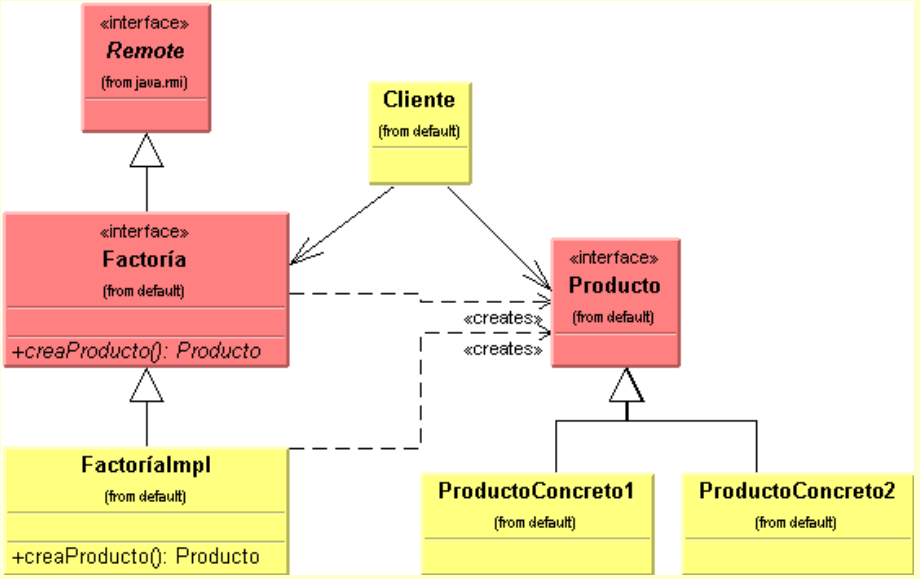
Despliegue

1. Compile los fuentes:
   ```
   javac -d . *.java
   ```
2. Use `rmic` para generar los archivos .class del stub y skeleton, a partir de la implementación y la interfaz:
   ```
   rmic -d . examples.hello.HelloImpl
   ```
3. Mueva `hello.html` a un servidor web (junto a los archivos .class) o use el appletviewer.
4. Arranque el registro rmi en el servidor:
   ```
   rmiregistry &
   ```


Despliegue

1. Arranque el servidor.
   Especifique el codebase y la política de la clase stub para poder cargarla dinámicamente en el registro
   ```
   java -
   Djava.rmi.server.codebase=http://myhost/~my
   usrname/myclasses/ -
   Djava.security.policy=$HOME/mysrc/policy
   examples.hello.HelloImpl
   ```
2. Ejecute el applet (desde un navegador o desde el appletviewer)

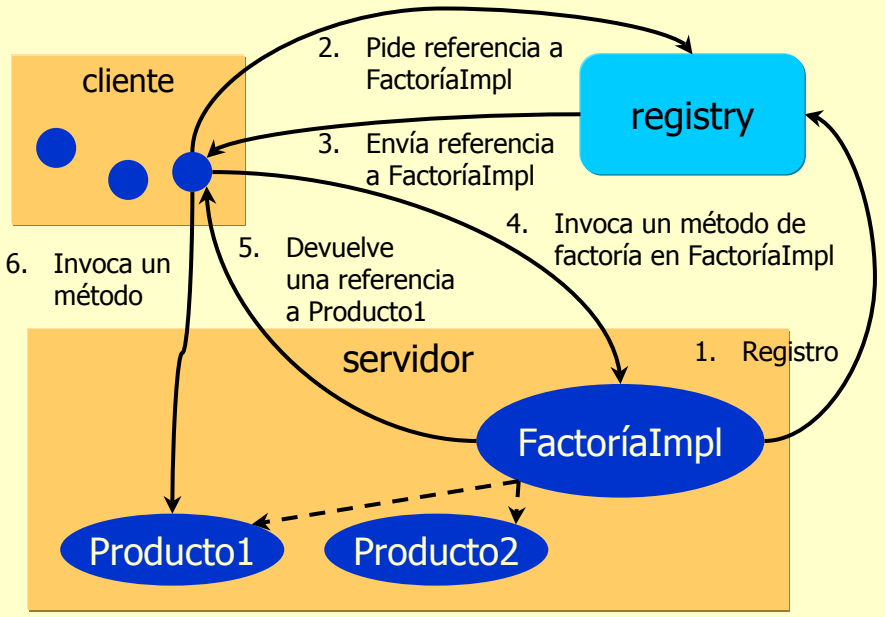
Usando métodos de factoría en RMI



Métodos de factoría

- En el patrón Factoría (o fábrica), una clase, que habitualmente es un Singleton se usa para crear instacias de otra clase(s).
  - Suele usarse para crear subclases de cierta clase, o diferentes implementaciones de cierta interfaz (como la clase URL en Java).
- En RMI una clase factoría podría servir de gancho (hook) para obtener nuevas referencias a objetos remotos nuevos.
  - Evitando el sobrecoste de programación.

Proceso de creación de objetos mediante factoría



Objetos remotos activables

- Antes de Java 2, sólo se podía acceder `UnicastRemoteObject` desde un programa servidor que creara instancias del objeto y corriera todo el rato.
- Con Java 2 tenemos `Activatable` y el daemon `rmid`.
- Una clase "activable" solo debe registrarse frente al `rmid`.
  - Esto nos permite ahorrar memoria y ganar prestaciones al no tener que activar objetos remotos que no se usen en este momento.

ActivatableImplementation.java

```
package examples.activation;
import java.rmi.*;
import java.rmi.activation.*;

public class ActivatableImplementation extends Activatable
                implements examples.activation.MyRemoteInterface {
   // Constructor para activar y exportar; y es
   // invocado por ActivationInstantiator.newInstance
   // durante la activación para construir el objeto.
   //
   public ActivatableImplementation(ActivationID id,
               MarshalledObject data) throws RemoteException {
   // Registra el objeto frente al sistema de activación
   // luego lo exporta sobre un puerto anónimo
   //
        super(id, 0);
   }
   // Implementa el metodo declarado en MyRemoteInterface
   //
   public Object callMeRemotely( ) throws RemoteException {
        return "Success";
} }
```


Detalles de la implementación

- Implementa (por supuesto) la interfaz remota.
- Precisa de un constructor con dos argumentos para ser usado por `activation`.
- El programa de servicio (`Setup.java`) deberá ser ligeramente más complejo, al acarrear tareas de registro frente al daemon.

Setup.java

```
package examples.activation;
import java.util.Properties;
import java.rmi.*;
public class Setup {
  public static void main(String[] args) throws Exception {
    System.setSecurityManager(new RMISecurityManager());
    Properties props = new Properties();
    props.put("java.security.policy",
              "/home/rmi_tutorial/activation/policy");
    ActivationGroupDesc.CommandEnvironment ace = null;
    ActivationGroupDesc exampleGroup = new
                                    ActivationGroupDesc(props,ace);
    ActivationGroupID agi =
                ActivationGroup.getSystem().registerGroup(exampleGroup);
    String location = "file:/home/rmi_tutorial/activation/";
    MarshalledObject data = null;
    ActivationDesc desc = new ActivationDesc (agi,
       "examples.activation.ActivatableImplementation", location, data);
    MyRemoteInterface mri = (MyRemoteInterface)
                                Activatable.register(desc);
    System.out.println("Got the stub for the ActivatableImplementation");
    Naming.rebind("ActivatableImplementation", mri);
    System.out.println("Exported ActivatableImplementation");
    System.exit(0);
} }
```


Setup.java

```
package examples.activation;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class Setup {
   // Esta clase registra información sobre la clase
   // ActivatableImplementation con rmid y rmiregistry
   //
   public static void main(String[] args) throws Exception {
```

- Aquí se ha separado la implementación del **main** que lanza el servidor.
  - Crea toda la información precisa para activar la clase sin crear instancias
- Ojo con los paquetes que hay que importar

Setup.java

```
System.setSecurityManager(new RMISecurityManager());
// A causa del modelo de seguridad de la 1.2, habrá que
// especificar una política de seguridad para la máquina
// virtual del ActivationGroup. El primer argumento del
// método put de Properties, heredado de Hashtable es la
// clave y el segundo es el valor.
//
Properties props = new Properties();
props.put("java.security.policy",
   "/home/rmi_tutorial/activation/policy");
```

- `Properties` permite manejar un espacio parecido al registro de Windows.
- Hay que conseguir el archivo de política y echar un vistazo.
  - También hay que depositarlo en un lugar accesible a la máquina virtual.

Setup.java

```
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc exampleGroup = new
   ActivationGroupDesc(props, ace);

// Una vez creado el ActivationGroupDesc, regístrese frente al
// sistema de activación para obtener su ID
ActivationGroupID agi =
       ActivationGroup.getSystem().registerGroup(exampleGroup);
// El String "location" especifica un URL desde donde vendrá la
// definición de la clase cuando se pida (active) el objeto.
// No olvidemos la barra del final del URL o no se encontrarán
// las clases.
String location = "file:/home/rmi_tutorial/activation/";
```

- El URL de "location" puede utilizarse vía http.

Setup.java

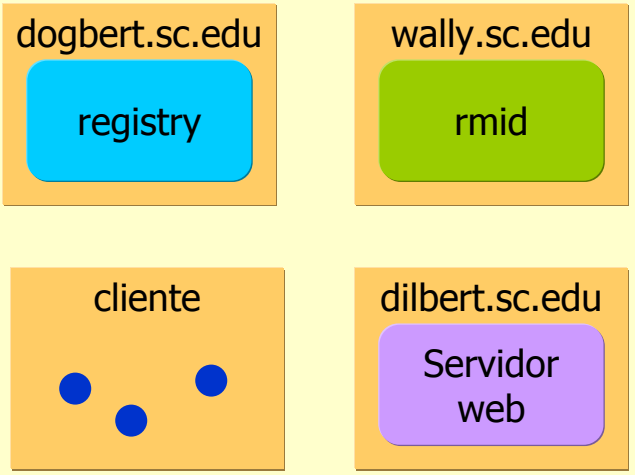
```
// Crea el resto de los parámetros que se pasarán al
// constructor de ActivationDesc
MarshalledObject data = null;
// El argumento de ubicación para el constructor de
// ActivationDesc se utilizará para identificar esta
// clase de forma única, en relación al URL de ubicación
// indicado en "location"
ActivationDesc desc = new ActivationDesc (agi,
  examples.activation.ActivatableImplementation",
  location,data);
```

- `ActivationDesc` dará toda la información que precisa `rmid` para crear una nueva instancia de la clase de implementación
- "Marshall" se utiliza para referirse al empaquetado de datos.
  - El valor de "`data`" debe apuntar a un objeto serializado, si no es un dato elemental.
- Se concatenará "ubicación" con "location" para construir el nombre que irá al `rmiregistry`.

Setup.java

```
    // Registro frente al rmid
    //
    MyRemoteInterface mri = (MyRemoteInterface)
                          Activatable.register(desc);
    System.out.println(
    "Obtuve el stub para ActivatableImplementation");
    // Enlaza el stub al nombre en el registro en el
    // puerto 1099
    //
    Naming.rebind("ActivatableImplementation", mri);
     System.out.println("Exported
    ActivatableImplementation");
    System.exit(0);
  } }
```

Despliegue: máquinas implicadas



Compilación y despliegue del servidor

1. Compilación:
   ```
   javac -d . *.java
   ```
2. Compilación de stubs y skeletons:
   ```
   rmic -d .
   examples.activation.ActivatableImplementation
   ```
3. Arranque del registro. Cerciórese de que CLASSPATH no contiene las clases que se quieran descargar en el cliente
   ```
   rmiregistry &
   ```
4. Lanzamiento del daemon de activación
   ```
   rmid -J -Djava.security.policy=rmid.policy &
   ```

Compilación y despliegue del servidor

5. Ejecución del programa setup.
   El codebase es el URL donde mora el stub, que habitualmente se obtiene vía http.

```
java -
Djava.security.policy=/home/rmi_tutorial/activat
ion/policy -
Djava.rmi.server.codebase=http://dilbert.usc.edu
/rmi_tutorial/activation/
examples.activation.Setup
```

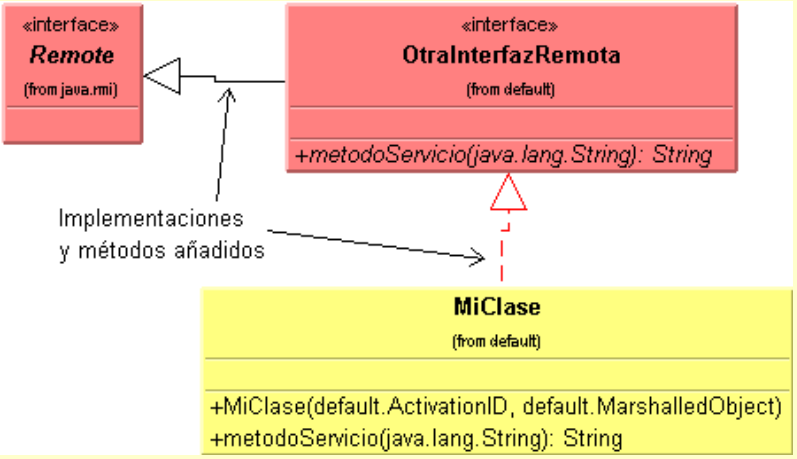
6. Ejecución del cliente

```
java -
Djava.security.policy=/home/rmi_tutorial/activat
ion/policy examples.activation.Client vector
```


Conversión de UnicastRemoteObject a Activatable

- Si hay servidores `UnicastRemoteObject` y queremos que sean `Activatable` habrá que cambiar:
  - Incluir los `import` apropiados.
  - Extender la clase `Activatable`.
  - Cambiar el constructor sin argumentos, por
  - Un constructor con dos argumentos
  - Y crear un programa de inicio como el ya visto.

Clases Activatables para nuestras factorías



- Para hacer que `MiClase` sea `activatable` solo tenemos que implementar una clase que extienda Remote y tener en cuenta un nuevo método.

Factoría de Sockets a gusto del cliente

- RMI usa sockets simples
  - Si queremos encriptar y comprimir datos habrá que implementar una factoría de Sockets RMI de cliente.
- Puede crearse un `RMISocketFactory` para ser usada con cada objeto remoto,
- O asociarla solo a ciertos objetos.

Factoría de sockets de cliente

- Los pasos son:

1. Implementar clases especiales `ServerSocket` y `Socket` (extendiéndolas) que implementen encriptación y/o compresión.
2. Implementar `RMIClientFactory` que devuelva uno de los sockets de cleinte.
   - Basicamente, redefinir la función `createSocket` function.
   - Esta clase deberá ser serializable.
3. Implementar un RMIServerSocketFactory, como antes, y que no tiene por que ser serializable

URLS

- Guía de uso de RMI
  http://java.sun.com/j2se/1.4/docs/guide/rmi/getstart.doc.html
- Carga dinámica de código
  http://java.sun.com/j2se/1.4/docs/guide/rmi/codebase.html
- Creación de un objeto activatable y demás
  http://java.sun.com/j2se/1.4/docs/guide/rmi/activation/activation.1.doc.html (2 y 3)
- Factoría de Sockets RMI del cliente
  http://java.sun.com/j2se/1.4/docs/guide/rmi/socketfactory/index.html/
- Patrones:
  Gamma, E., "Patrones de Diseño", Addison-Wesley, 2003