PROGRAMACION DISTRIBUIDA

Aspectos avanzados de RMI

Héctor Pérez

Objetivos

- Análisis de un conjunto de estrategias de diseño utilizando RMI que nos pueden servir de guía para el diseño de sistemas distribuidos
- Casos analizados:
  - Transferencia de objetos como parámetros o valores de retorno
  - Carga dinámica de clases
  - Uso del *registry* con varios proyectos independientes
  - Inversión de la dependencia cliente/servidor
  - Activación dinámica de servidores

Transferencia de objetos (1/2)

- Los objetos locales son serializados y transmitidos por la red al destino
  - se transmiten los datos, los metadatos y los métodos del objeto
  - en el destino, la información recibida es utilizada para crear un ***clon*** del objeto original

```
class Position {
    public int x, y;
    public void print() {…}
}
```

```
class Position {
    public int x, y;
}
```

```
public interface AutonomousCar extends Remote {
    public void setPosition(int x, int y) throws RemoteException;
    public void setPosition(Position p) throws RemoteException;
    public Position getPosition() throws RemoteException;
}
```


Transferencia de objetos (2/2)

- Los objetos remotos se transmiten utilizando el *stub*
  - en el destino, el objeto remoto es representado por un *proxy* que nos permite modificar el estado del objeto original

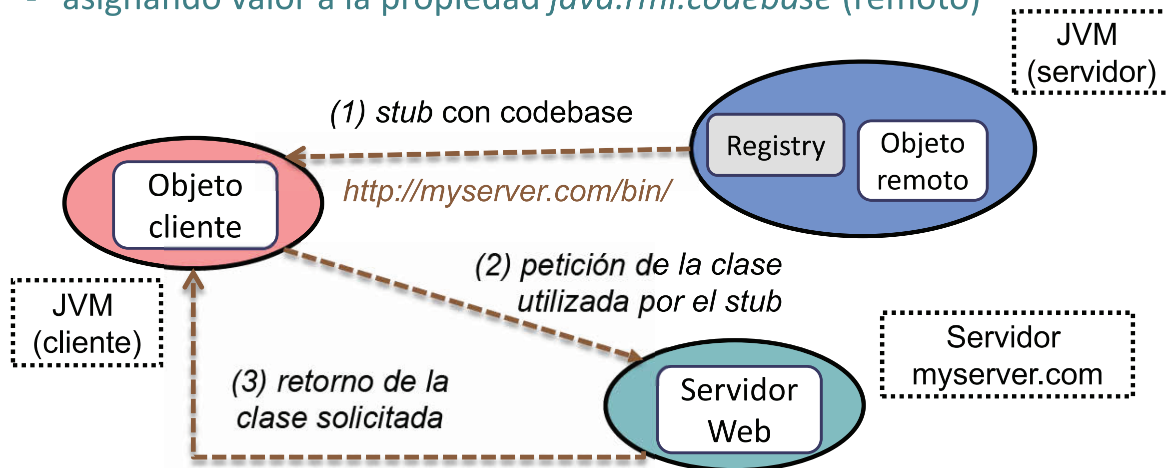
```
public interface CarDatabase extends Remote {
    public AutonomousCar getCar (String numberPlate) throws RemoteException;
}

public class CarCompany implements CarDatabase {
    public AutonomousCar getCar (String numberPlate) throws RemoteException {
        …
        return theCar;
}
```

- También pueden obtenerse también **de forma dinámica**

Carga dinámica de clases (1/6)

- Java RMI requiere conocer el lugar donde están declaradas la clases que ha de construir en las invocaciones remotas
  - utilizando la variable *CLASSPATH* (local)
  - asignando valor a la propiedad *java.rmi.codebase* (remoto)



Carga dinámica de clases (2/6)

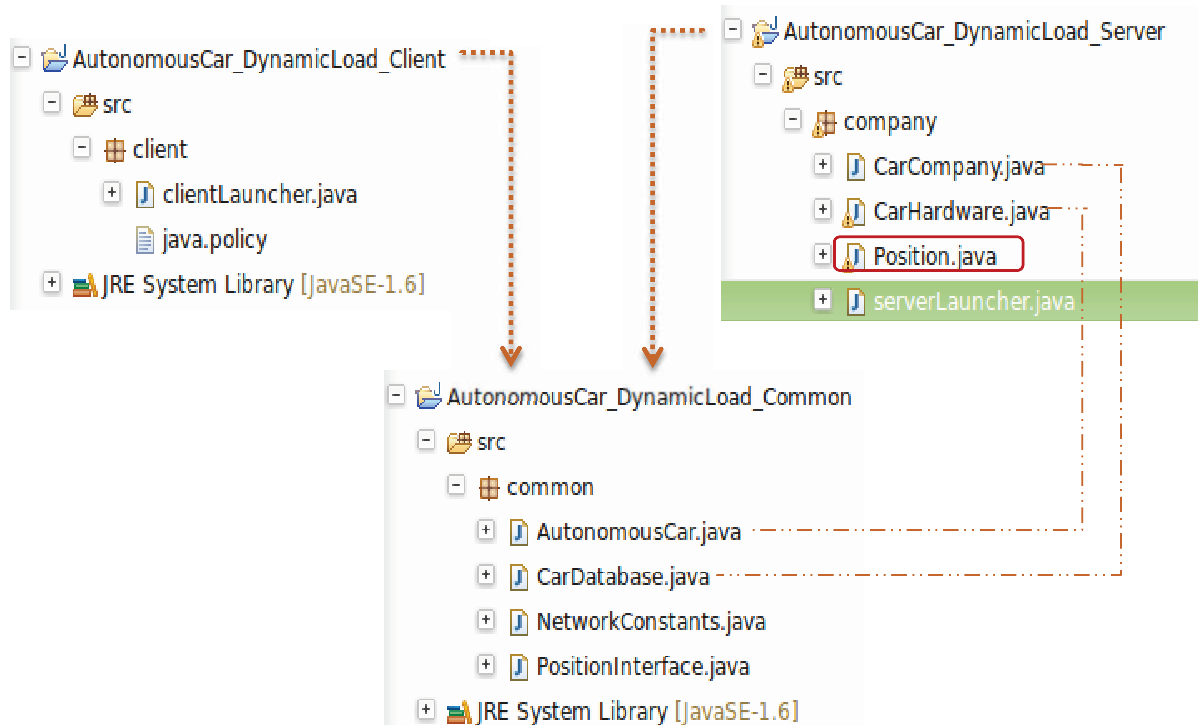
- Si no se encuentra la clase adecuada, el unmarshalling fallará y se **lanzará la excepción** *UnmarshalException* junto con *ClassNotFoundException*
- Ejemplo de carga dinámica *del bytecode de la clase Position*

```
class Position implements PositionInterface {
    private int x, y;
    public void print() {…}
}
```

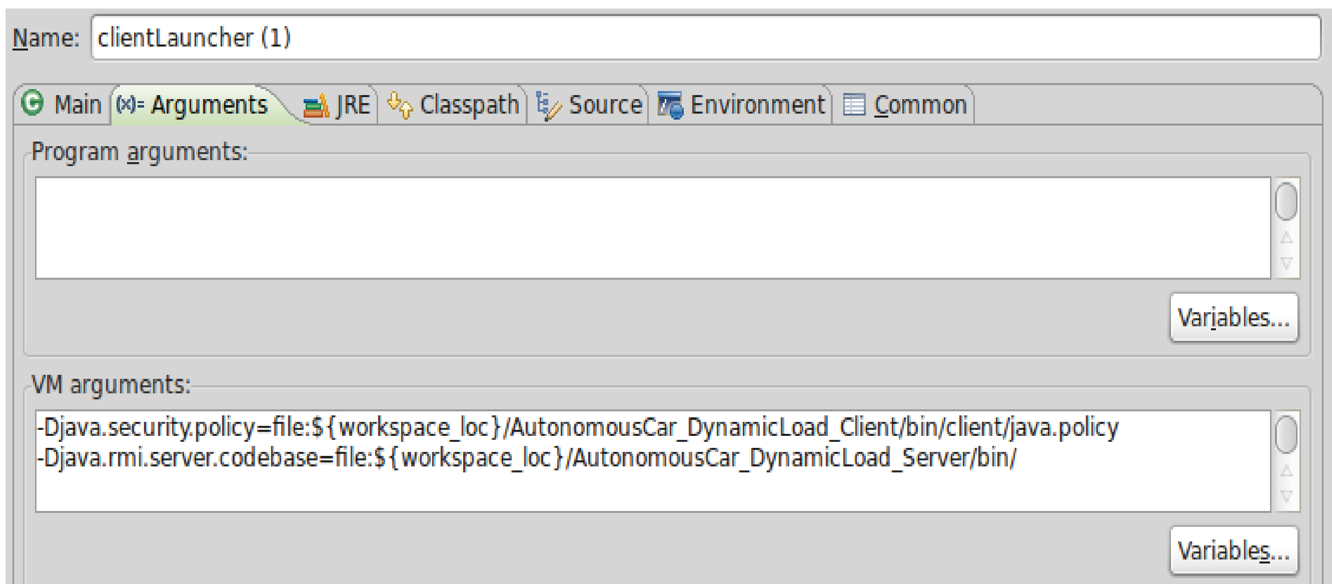
```
public interface PositionInterface {
    public void print();
}
```

```
public interface AutonomousCar extends Remote {
    public void setPosition(int x, int y) throws RemoteException;
    public PositionInterface getPosition() throws RemoteException;
}
```

Carga dinámica de clases (3/6)



Carga dinámica de clases (4/6)



Carga dinámica de clases (5/6)

```
public class Position implements Serializable, PositionInterface {
      private int x, y;
      …
      @Override
      public void print () { System.out.println("Posición actual: "+x+" "+y);}
}
```

```
public class clientLauncher {
  public static void main(String[] args) {
    …
    try{distribuidor=(CarDatabase) registry.lookup(NetworkConstants.ServiceName);
    } catch(Exception e){…}
    try {MyCar = distribuidor.getCar(myNumberPlate);  // Manejo del coche
           MyCar.setPosition(2,2);
           System.out.print ("Posición final: ");  MyCar.getPosition().print();
     } catch (RemoteException e) {…}
 }
```

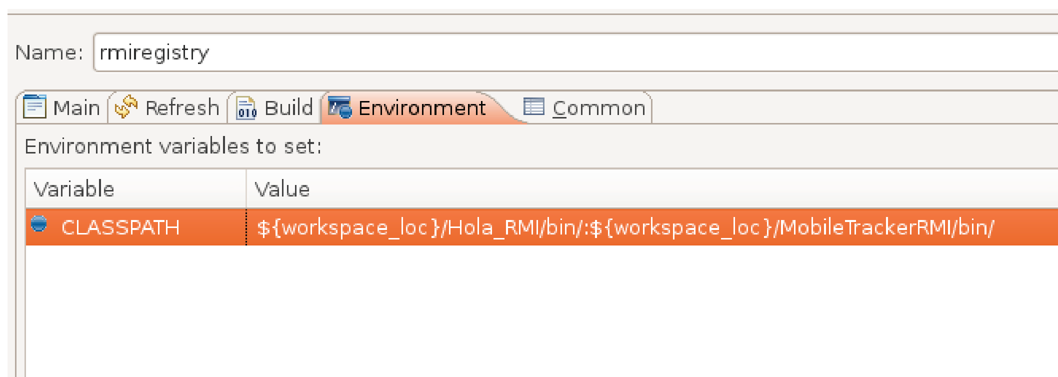

Carga dinámica de clases (6/6)

- La carga dinámica permite que no sea imprescindible conocer la implementación de la clase con antelación
  - con conocer la **interfaz** es suficiente
- Antes de ejecutar cargar clases de forma dinámica, debe activarse el **gestor de seguridad** con la política adecuada

```
public class clientLauncher {
   public static void main(String[] args) {
       …
       // Configuramos la JVM para poder obtener clases de forma dinámica
       // java.rmi.server.codebase y java.security.policy
       if (System.getSecurityManager()==null) {
           System.setSecurityManager(new RMISecurityManager());
       }
```

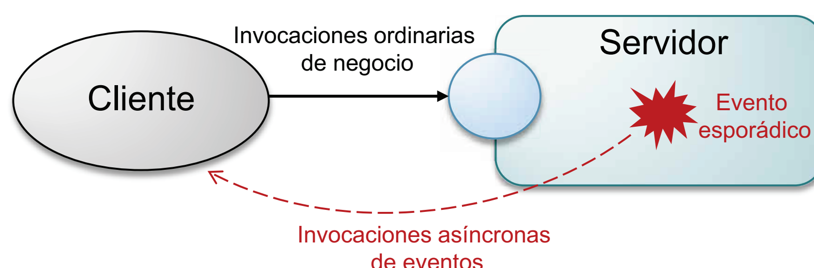
Uso del *registry* con varios proyectos independientes

- El *registry* tiene acceso a la clase del objeto a registrar si se ejecuta en la misma JVM que el servidor. Esto se consigue ejecutando el *registry* con
  `LocateRegistry.createRegistry (port)`
- Si se comparte un *registry* entre servidores provenientes de varios proyectos, es más sencillo utilizar la aplicación ***rmiregistry***
  - *debe arrancarse con el CLASSPATH incluyendo todas las clases remotas*



Inversión de la dependencia cliente/servidor (1/3)

- De forma general, el cliente conoce al servidor y puede invocar sus métodos, pero el servidor no conoce quiénes son sus clientes
  - en el modelo cliente-servidor puro, el servidor es **pasivo**
- Hay casos en los el cliente necesita conocer **cambios de estado** en el servidor
  - el servidor debe informar al cliente

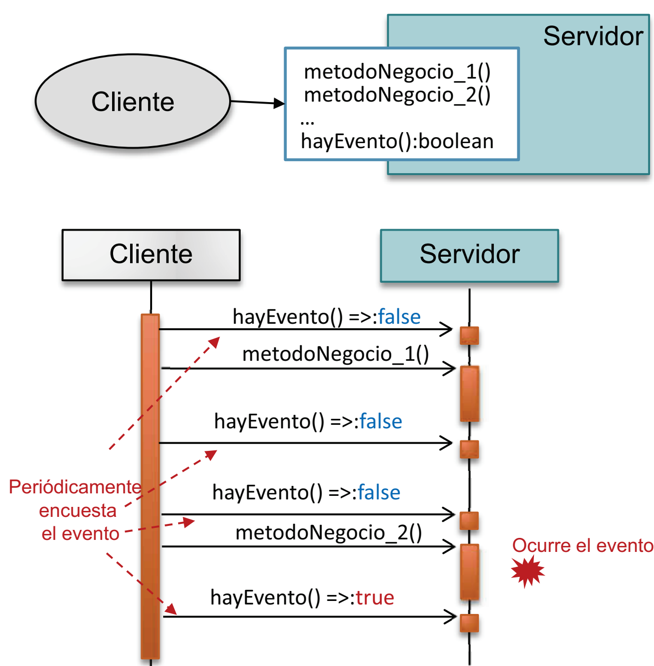


Inversión de la dependencia cliente/servidor (2/3)

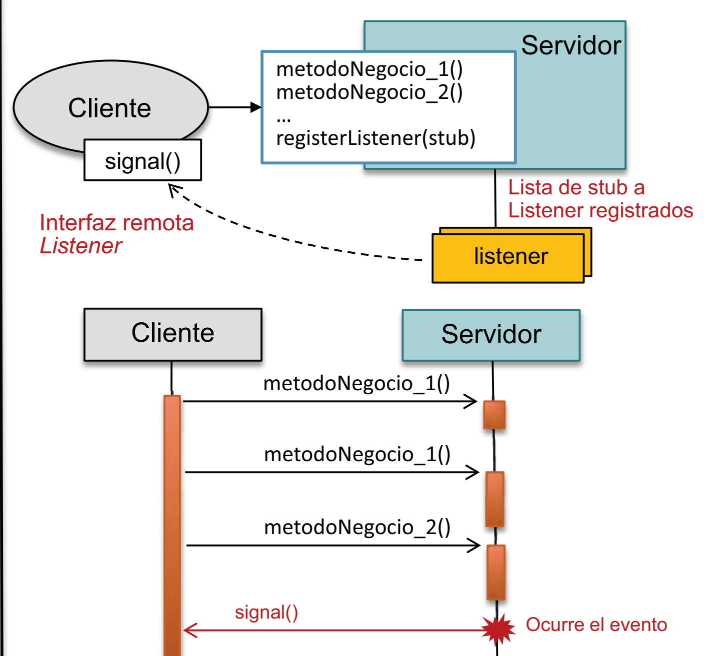
- Esto se puede resolver con dos estrategias:
  - **Polling**: El cliente periódicamente pregunta si ha existido el evento esporádico
    - Solución sencilla: no se modifica el servidor
    - No invierte la dependencia cliente/servidor
    - Sobrecarga el servidor
  - **Callback**: El cliente aloja en el servidor un ***objeto de notificación***, que el servidor utiliza para comunicar eventos al cliente
    - Solución más compleja: requiere modificar cliente y servidor
    - Se invierte la dependencia cliente/servidor, ya que el cliente debe implementar un objeto remoto

Inversión de la dependencia cliente/servidor (3/3)

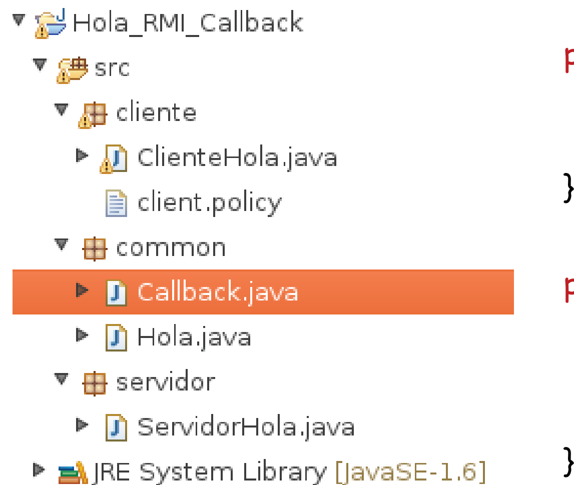
Estrategia de polling



Estrategia de callback



Ejemplo de callback: Hola (1/3)




```
public interface Callback extends Remote {
    void notificacion (String mensaje)
                         throws RemoteException;
}

public interface Hola extends Remote {
    void di_hola() throws RemoteException;
    void registraCallback (Callback cliente)
                         throws RemoteException;
}
```


Ejemplo de callback: Hola (2/3)

```
public class ClienteHola extends UnicastRemoteObject implements Callback {

    public ClienteHola() throws RemoteException {
          super (7000);
    }

    @Override
    public void notificacion(String mensaje) throws RemoteException {
        String returnMessage = "Callback recibido: " + mensaje;
        System.out.println (returnMessage);
    }

    public static void main(String[] args) {
          ...
          elServidor.registraCallback( new ClienteHola());
          elServidor.di_hola();
          ...
    }
```

Ejemplo de callback: Hola (3/3)

```
public class ServidorHola implements Hola {
    private Vector<Callback> ListaClientes;
    public ServidorHola() {
        ListaClientes = new Vector<Callback>();
    }
    public void di_hola() throws RemoteException {
        String elMensaje="Hola nuevo Mundo";
        System.out.println (elMensaje);
        ListaClientes.firstElement().notificacion("Saludo realizado correctamente");
    }
    @Override
    public void registraCallback(Callback cliente) throws RemoteException {
            ListaClientes.addElement (cliente);
    }
```


Activación dinámica de servidores (1/2)

- Un servidor instalado en un procesador consume recursos: memoria, puertos de comunicación, etc.
  - frecuentemente una aplicación mantiene múltiples servidores, de los cuales sólo algunos de ellos están siendo invocados concurrentemente
  - por lo tanto, es importante disponer de servidores que puedan **activarse o desactivarse** según estén siendo invocados
- Un servidor tiene **estado** que le permite operar consistentemente
- En estos casos hay que preguntarse por el ciclo de vida de un servidor:
  - ¿Cuando un servidor debe activarse?
  - ¿Cuando debe desactivarse?
  - ¿Cuando debe almacenarse el estado en una base de datos persistente?

Activación dinámica de servidores (2/2)

- RMI proporciona recursos para declarar y registrar servidores que no se instancian hasta que un cliente remoto lo invoca
  - Para el cliente, el proceso de activación del servidor durante la invocación es **transparente**:
    - localiza su referencia en el *registry* e invoca el servidor. Si es el primero en invocarlo, el servidor se activa
  - El servidor también puede desactivarse
    - y guardar su estado si es **persistente**, y recuperarlo posteriormente en la siguiente activación
- La base del proceso es un servidor RMI proporcionado por Java que se denomina **rmid** (RMI Daemon).

rmid (The Java RMI Activation System Daemon)

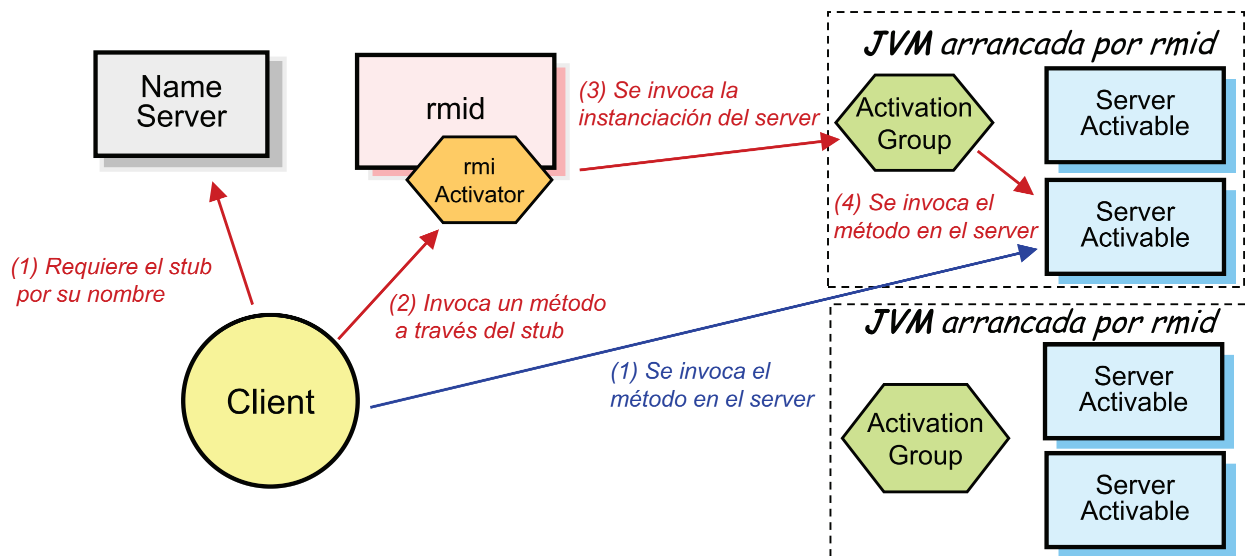
- La ejecución de la herramienta **rmid** inicia la actividad del servicio *ActivationSystem* en modo *demonio*
  - debe estar en ejecución antes de que sea invocado (y activado) cualquier objeto *activable*
- La herramienta se lanza con el comando (al menos hay que especificar una política de seguridad Java):

```
rmid -J-Djava.security.policy=rmid.policy
```

- Existen muchas opciones de lanzamiento que condicionan el funcionamiento del servicio *ActivationSystem*. Por defecto:
  - el servicio atiende en el puerto **1098**. La opción –port permite configurarlo
  - en el registry se asocia la clave "**java.rmi.activation.ActivationSystem**" con el servicio ActivationSystem

Elementos del framework Activation

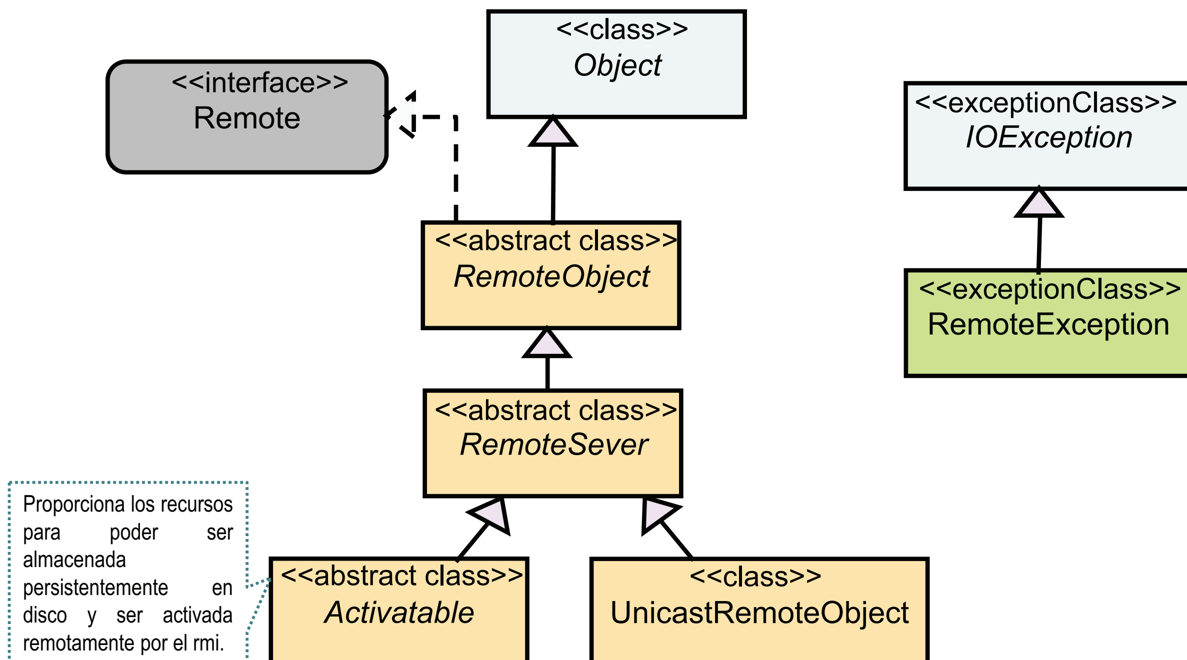
NameServer	rmid	ActivationGroup	ServerActivable
• Servicio RMI en el que se registra el servidor cuando se crea	• Servicio RMI que recibe las invocaciones si el servidor no está activo	• Servicio RMI que instancia el servicio en una determinada JVM	• Servidor RMI que se instancia cuando se invoca



Instanciación del servidor

- El *rmid* no instancia directamente el servidor en su propia JVM, sino que crea una JVM en la que instancia el servidor
    - todos los servidores que se crean con referencia a un mismo *ActivationGroup* se ejecutan en una misma JVM
- En cada JVM se instancia un *ActivationGroup*
    - es un servidor RMI encargado de instanciar e inicializar cada uno de los servidores de su grupo de activación
- Cuando el *ActivationGroup* instancia el servidor en su JVM, modifica la referencia del servidor (*stub*) para que haga referencia al servidor creado
    - por tanto, el stub deja de apuntar al servicio *Activator* del rmid

Implementación de un servidor activable (1/3)



Implementación de un servidor activable (2/3)

- La clase ***Activatable*** define un objeto remoto que se activa cuando es invocado algún método remoto, y que se desactiva cuando se le requiere

static Remote **exportObject** (Remote obj, ActivationID id, int port)	*Export the activatable remote object to the RMI runtime to make the object available to receive incoming calls.*
static Remote **register** (ActivationDesc desc)	*Register an object descriptor for an activatable remote object so that is can be activated on demand.*
static boolean **unexportObject** (Remote obj, boolean force)	*Remove the remote object, obj, from the RMI runtime.*
boolean **inactive** (ActivationID id)	*Informs the system that the object with the corresponding id is inactive.*
static void **unregister** (ActivationID id)	*Revokes previous registration for the activation descriptor associated with id*

Implementación de un servidor activable (3/3)

- El servidor es activable si se crea un objeto *Activatable,* que es una especialización de RemoteServer. Hay dos formas de realizar la implementación:
  - El servidor extiende la clase *java.rmi.activation.Activatable*
  - Exportándolo a través del método estático Activatable.ExportObject()
- Un stub a un objeto *Activatable* cambia dinámicamente en función del estado del servidor:
  - Inicialmente contiene la referencia para acceder al *ActivationSystem* del demonio rmid
  - Cuando el servidor está creado, se modifica el stub para referenciar directamente al server instanciado

Registro de un servidor activable

1. Escribir el código del servidor para que sea activable

- Escribir el código de la interfaz remota
- Escribir el código del servidor que la implementa

2. Registrar el servidor en el rmid y rmiregistry

- Ejecutar el rmid
- Ejecutar el rmiregistry
- Ejecutar programa de setup

Instalar gestor de seguridad
Transferir la información de la clase al rmid
Registrar la referencia remota en el registry

3. Ejecutar el cliente que invoca el servidor activable

- Requiere la referencia al rmiregistry
- Invoca un método del servidor

Desactivación de un servidor

- Para desactivar un servidor existen tres método estáticos de la clase *Activatable*:
  - **public static boolean unexportObject(Remote obj, boolean force)**
    - Cancela la exportación del servidor: comunica al rmid que el servidor ya no acepta mas llamadas remotas
  - **public static boolean inactive(ActivationID id)**
    - ***Desactivación temporal****: c*onfigura el registro del servidor en el rmid para que se active de nuevo si es invocado
  - **public static void unregister(ActivationID id)**
    - ***Desactivación definitiva:*** *e*limina el registro del servidor en el rmid, para que ya no pueda reactivarse de nuevo el servidor