4.3: Computación distribuida: Java RMI

A. Goñi, J. Ibáñez, J. Iturrioz, J.A. Vadillo

informatika fakultatea
facultad de informática

OCW 2013

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

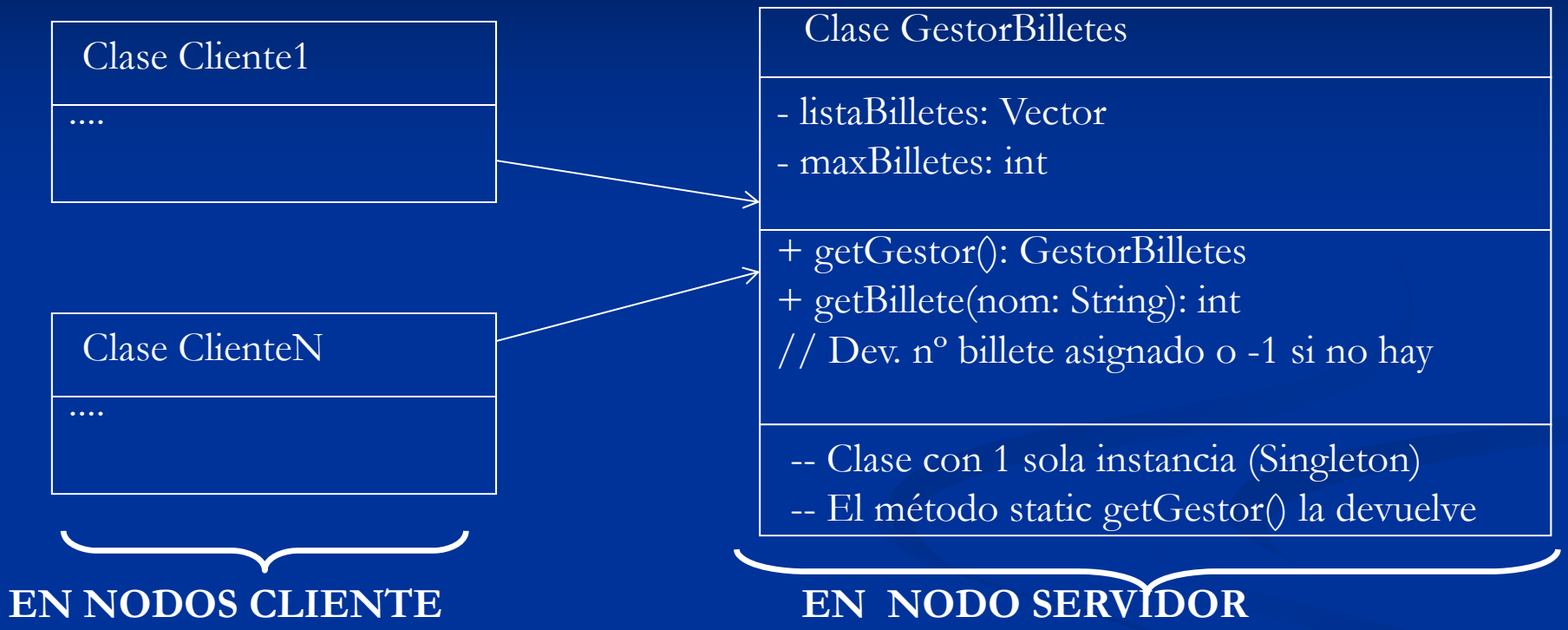
Indice

- RMI: Introducción
- Construcción de aplicaciones RMI
  - Definir Interfaz remota
  - Implementar interfaz remota: Clase remota
  - Registrar objeto clase remota
  - Localizar y ejecutar objeto remoto
- Arquitectura RMI
- Ejemplo
- Conexión entre nivel de presentación, lógica del negocio y nivel de datos
- Evolución del sistema

Introducción a RMI

- RMI (Remote Method Invocation)
  - Es un API,
  - Conjunto de interfaces, clases y métodos que permiten desarrollar en Java aplicaciones distribuidas de manera sencilla
- Equivalente a RPC (Remote Procedure Call)
- Existen otros estándares como CORBA
  - Permite desarrollar aplicaciones distribuidas usando distintos Lenguajes de Programación

Introducción a RMI



En una arquitectura distribuida no se puede hacer lo siguiente:

```
GestorBilletes g = GestorBilletes.getGestor();
return g.getBillete("Kepa Sola");
```

Ya que GestorBilletes es un objeto de la máquina virtual remota

Construcción aplicaciones RMI

Para construir una aplicación Cliente/Servidor donde un cliente acceda a un servicio remoto (clase remota) usando RMI hay que:

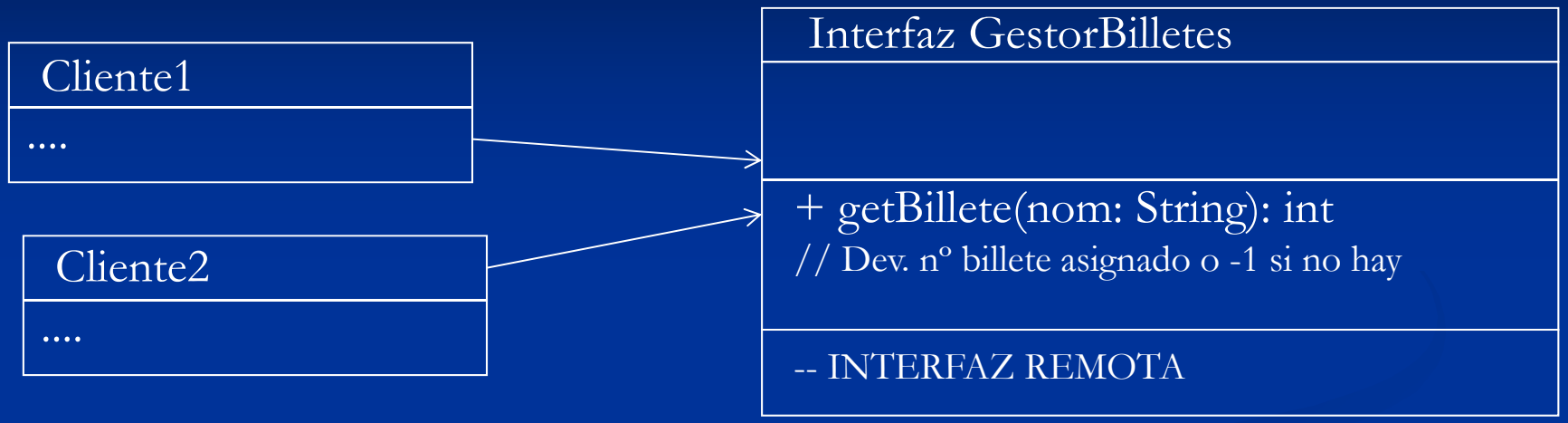
1.- Definir la interfaz remota

2.- Implementar la interfaz remota (clase remota)

3.- Registrar un objeto de la clase remota

4.- Localizar y ejecutar el objeto remoto

1. Definir la interfaz remota

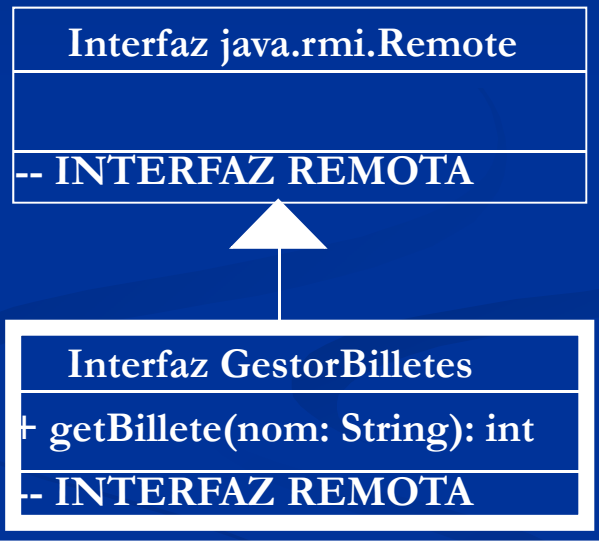


RMI permite invocar a un objeto remoto
Para ello hay que definir una interfaz remota
Así, un cliente puede hacer lo siguiente:

```
GestorBilletes g;
// Código para obtener la dirección del
// objeto remoto y dejarlo en g
return g.getBillete("Kepa Sola");
```

1. Definir la interfaz remota

```
// GestorBilletes.java
import java.rmi.*;
public interface GestorBilletes
        extends Remote
{
 public int getBillete(String nom)
      throws RemoteException;
}
```

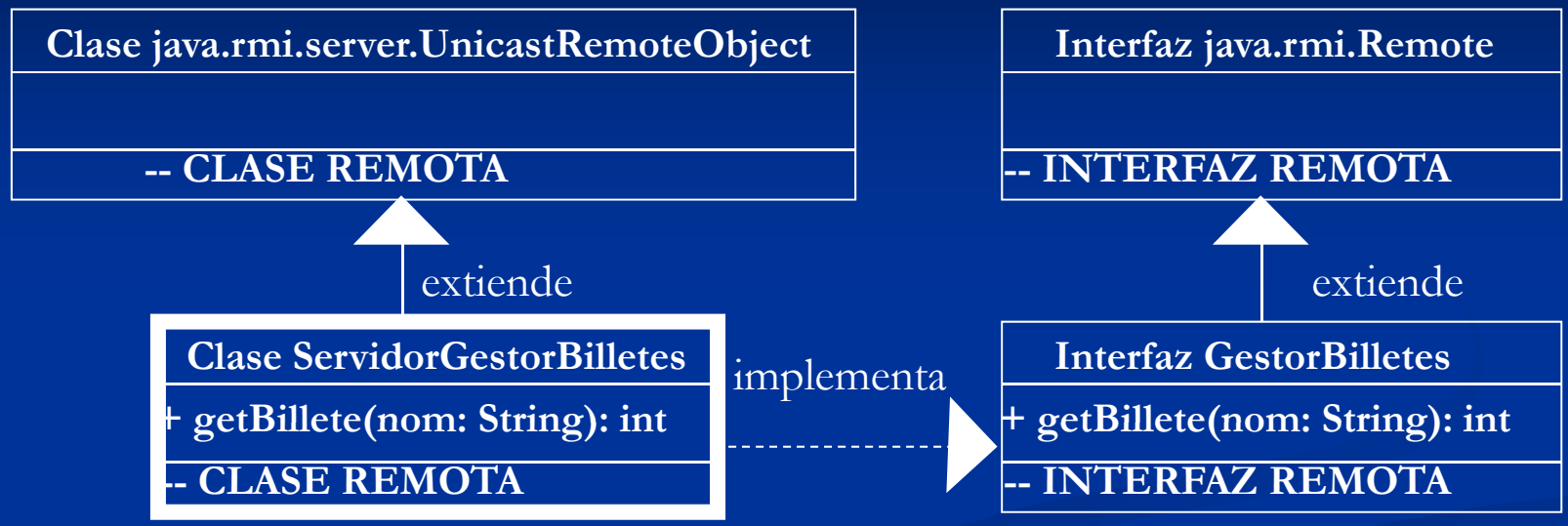


La interfaz extiende **`java.rmi.Remote`**

Todos los métodos deben lanzar **`java.rmi.RemoteException`**

2. Implementar la Interfaz Remota

La clase Remota



El servidor implementa la interfaz remota

Extiende `java.rmi.server.UnicastRemoteObject`

2. Implementar la Interfaz Remota

La clase Remota

```
// ServidorGestorBilletes.java
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
public class ServidorGestorBilletes
                              extends UnicastRemoteObject
                              implements GestorBilletes{
  private Vector listaBilletes = new Vector();
  private static int maxBills = 50;
  public ServidorGestorBilletes() throws RemoteException{}

  public int getBillete(String nom) throws RemoteException{
        //lógica de negocio
        num=9999;  // Devuelve siempre el billete 9999...
        return num;
  }
```

3. Registrar un objeto de la clase remota

- Se crea un registro en el servidor (en el main() de la misma clase que el objeto u otra distinta)

```
java.rmi.registry.LocateRegistry.createRegistry(numPuerto);
```

- Se crea un objeto de la clase remota

```
ServidorGestorBilletes objetoServidor =
                        new ServidorGestorBilletes();
```

- Se crea un nombre para ese servicio

```
String servicio = "rmi://localhost:1099//gestorBilletes"; // si 1099 es numPuerto
```

- Se registra ese servicio con ese nombre

```
Naming.rebind(servicio,objetoServidor)
```

3. Registrar un objeto de la clase remota

Ejemplo

```
class ServidorRemoto
 public static void main(String[] args) {
 //Falta el java.policy
 try { java.rmi.registry.LocateRegistry.createRegistry(1099);
} catch (Exception e)
{System.out.println("Rmiregistry ya lanzado"+e.toString());}
   try {
       ServidorGestorBilletes objetoServidor =
             new ServidorGestorBilletes();

       String servicio = "//localhost/gestorBilletes";
       //          "//localhost:NumPuerto/NombreServicio"
       // Registrar el servicio remoto
       Naming.rebind(servicio,objetoServidor);
   } catch (Exception e)
       {System.out.println("Error al lanzar el servidor");}
}
```

1

2

3

4

3. Registrar un objeto de la clase remota

El registrador: RMIRegistry

- java.rmi.registry.LocateRegistry.createRegistry(p)

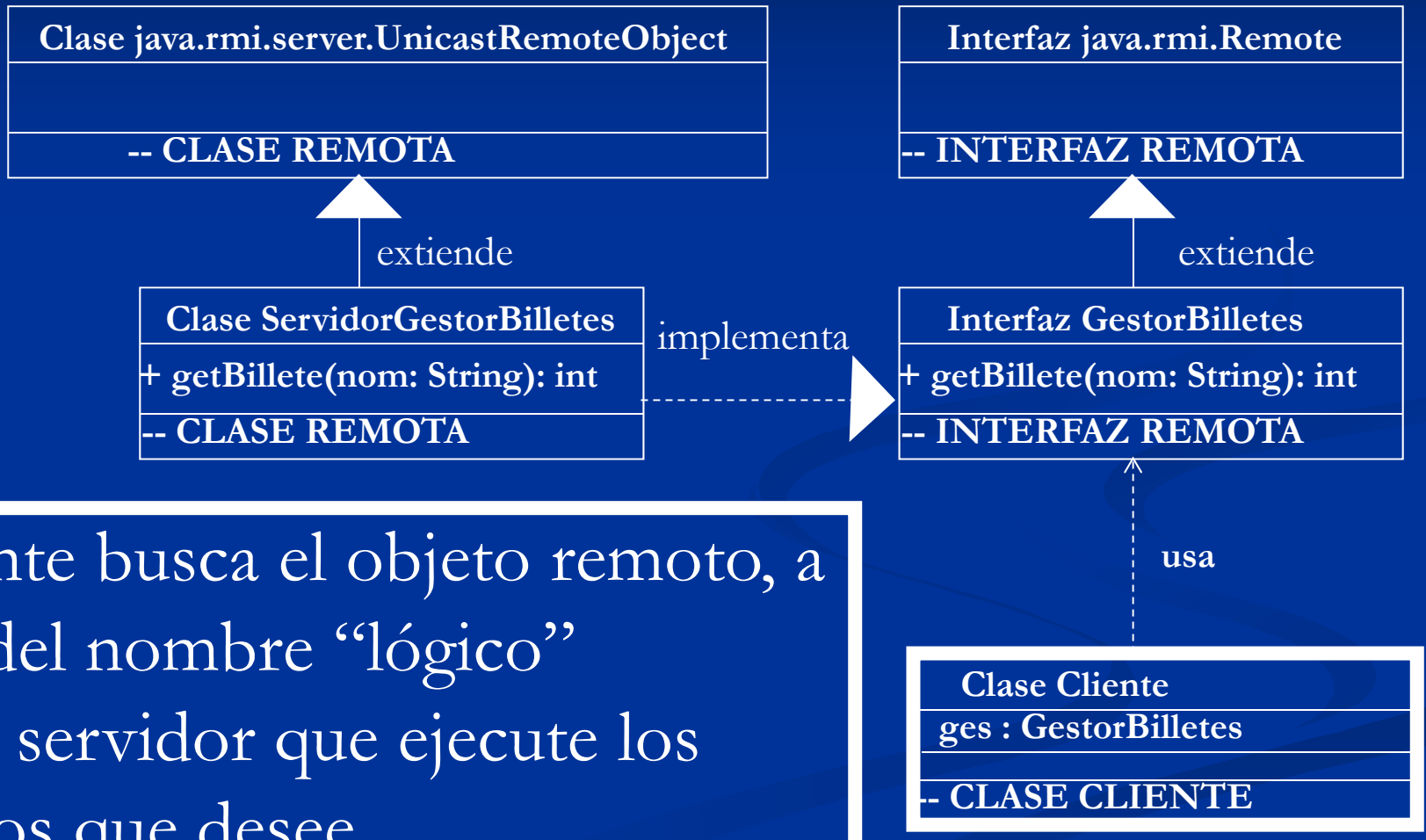
Crea el proceso rmiregistry en el puerto p.
El rmiregistry lanzado no acaba aunque acabe el servidor RMI
Lanza una excepción si el puerto está ocupado

```
try { java.rmi.registry.LocateRegistry.createRegistry(1099);
} catch (Exception e)
{System.out.println("Rmiregistry ya lanzado"+e.toString());}
```

Código que lanza el rmiregistry y controla la excepción que se puede levantar al reejecutar varias veces el servidor RMI

Parar el rmiregistry:UnicastRemoteObject.*unexportObject*(registry, **true**);

4. Localizar y ejecutar el objeto remoto



El cliente busca el objeto remoto, a partir del nombre "lógico"
Pide al servidor que ejecute los métodos que desee

4. Localizar y ejecutar el objeto remoto

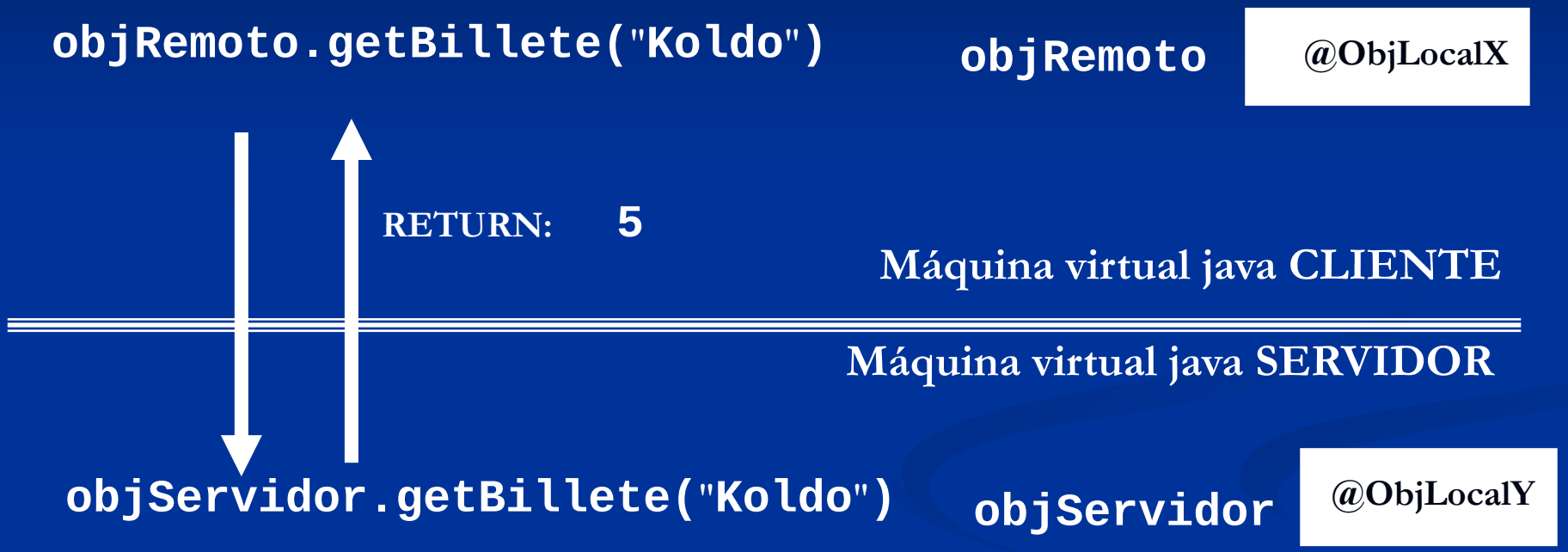
```
import java.rmi.*;
public class Cliente {
 public static void main(String[] args) {
   // Falta gestion java.policy
     GestorBilletes objRemoto;
   String nomServ = "rmi://localhost/gestorBilletes";
   //        "rmi://DireccionIP:NumPuerto/NombreServicio"
 try {
      objRemoto = (GestorBilletes)Naming.lookup(nomServ);
      int b = objRemoto.getBillete(args[0]);
      if (b==-1) System.out.println("No hay billetes");
      else System.out.println("Obtenido : "+b);
 } catch (Exception e) { System.out.println("Error... ");}
}}
```

1

2

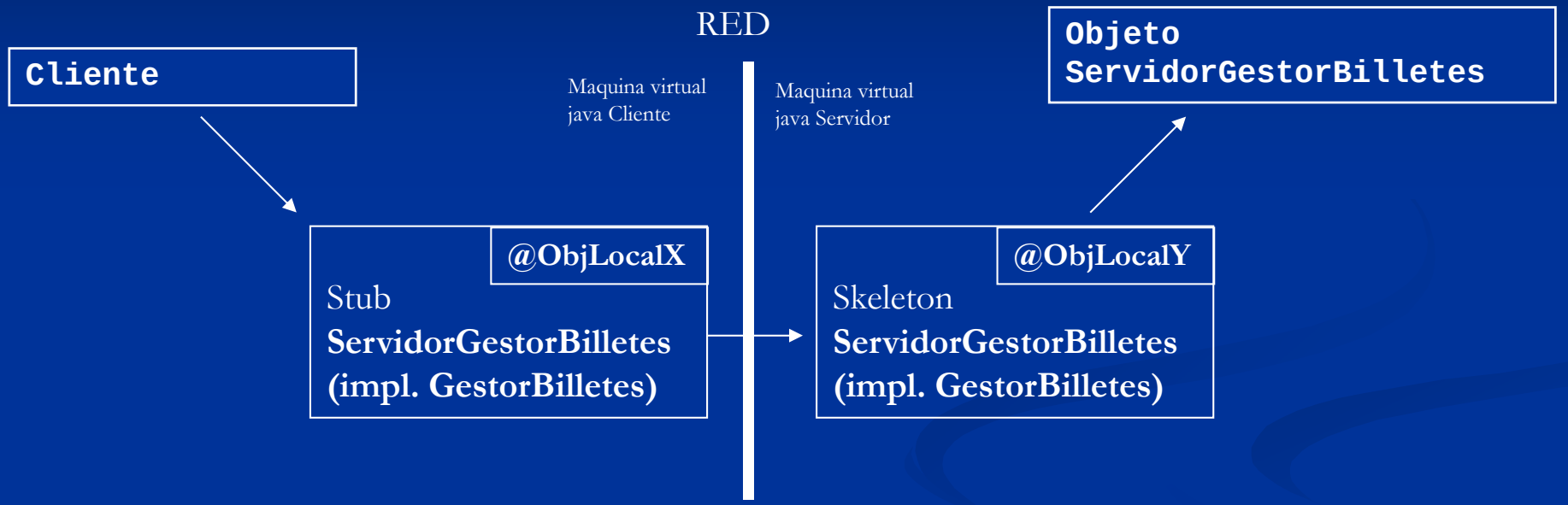
3

Arquitectura RMI



Se desea conseguir que lo que se le pida al objeto **`@ObjLocalX`** sea ejecutado por el objeto **`@ObjLocalY`** en otra máquina virtual Java

Arquitectura RMI



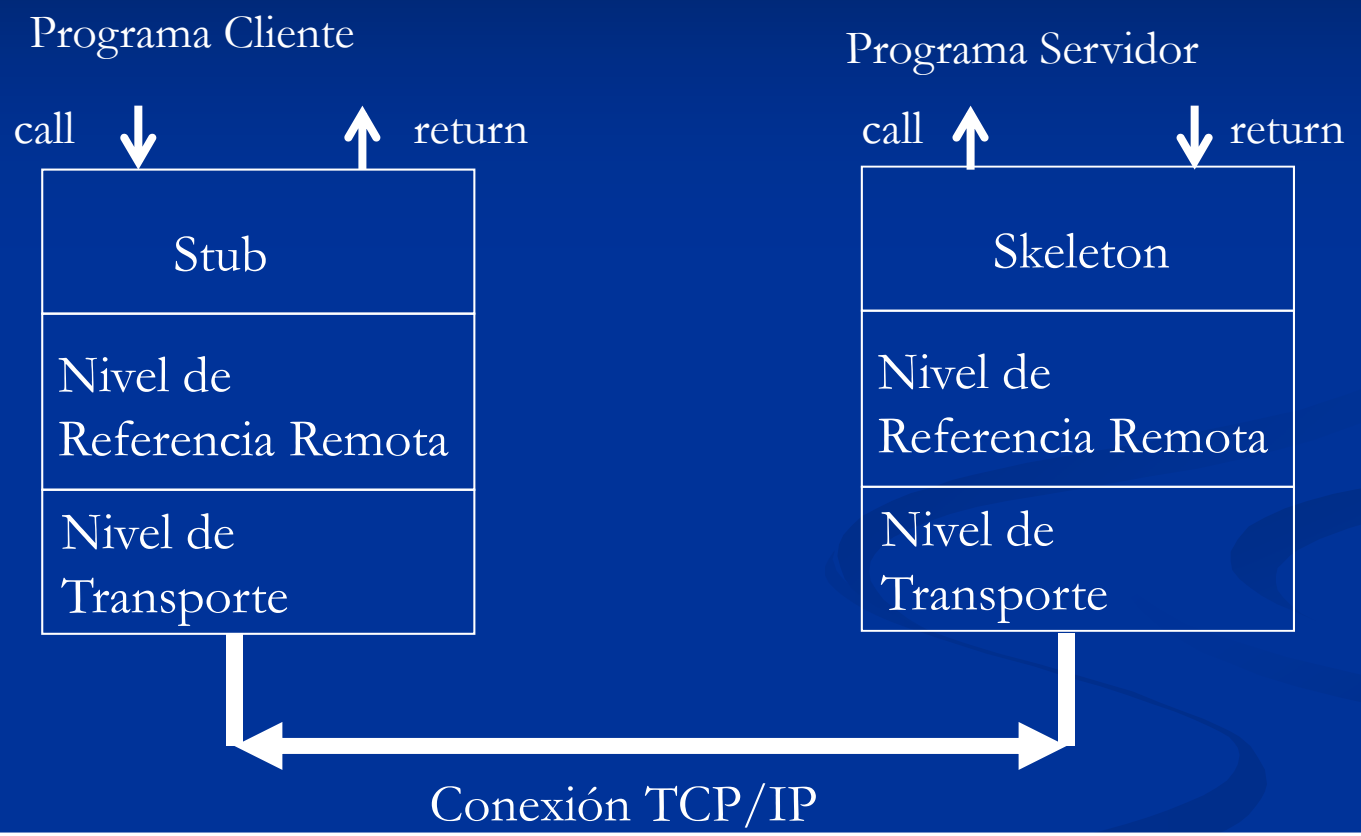
Stub ServidorGestorBilletes: Representante del objeto ServidorGestorBilletes en el cliente
Skeleton ServidorGestorBilletes: Representante del objeto ServidorGestorBilletes en el servidor

NOTA: Los objetos Stub, Skeleton y el objeto remoto comparten la misma interfaz

POR LO TANTO: hay que asegurarse de que las clases STUB y la INTERFAZ REMOTA están accesibles en el cliente (copiándolas en el mismo, por ejemplo)

Arquitectura RMI:

Objetos *Stub* y *Skeleton*



Los objetos Stub y Skeleton se encargan de realizar la conexión y del paso de parámetros y resultados

Arquitectura RMI:

Objetos *Stub* y *Skeleton*

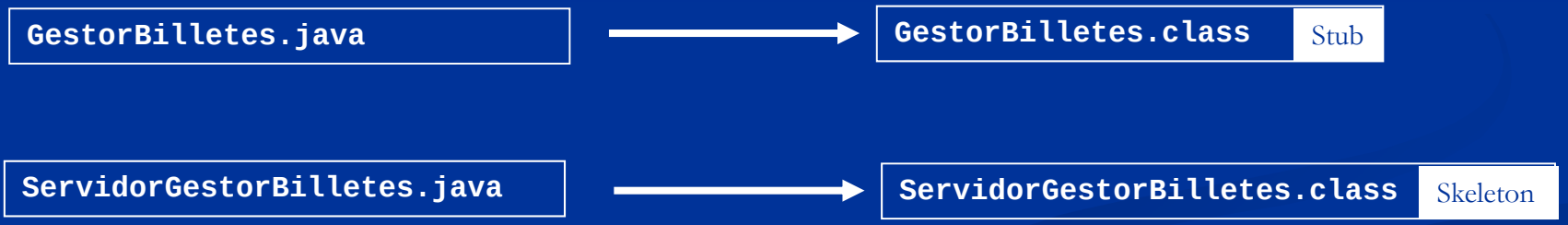
- En principio, los stubs y los skeleton se pasan los objetos enviados como parámetros y los resultados REALIZANDO UNA COPIA DE SUS VALORES (y de los objetos incluidos en ellos, recursivamente).
  - No se pasan referencias a un objeto remoto.
- Para ello, se usan los mecanismos de serialización de Java

POR LO TANTO: las clases de los objetos que se pasen por parámetro en métodos remotos deben implementar la interfaz **`Serializable`**

Arquitectura RMI:

Generación de stubs y skeletons

El STUB y SKELETON están asociados a la interfaz remota y a la clase remota



Cuando se compilan se añaden el *Stub* y el *Skeleton* a las clases correspondientes(jdk6.0)

Nota: se generaban explícitamente como clases independientes usando la herramienta RMIC

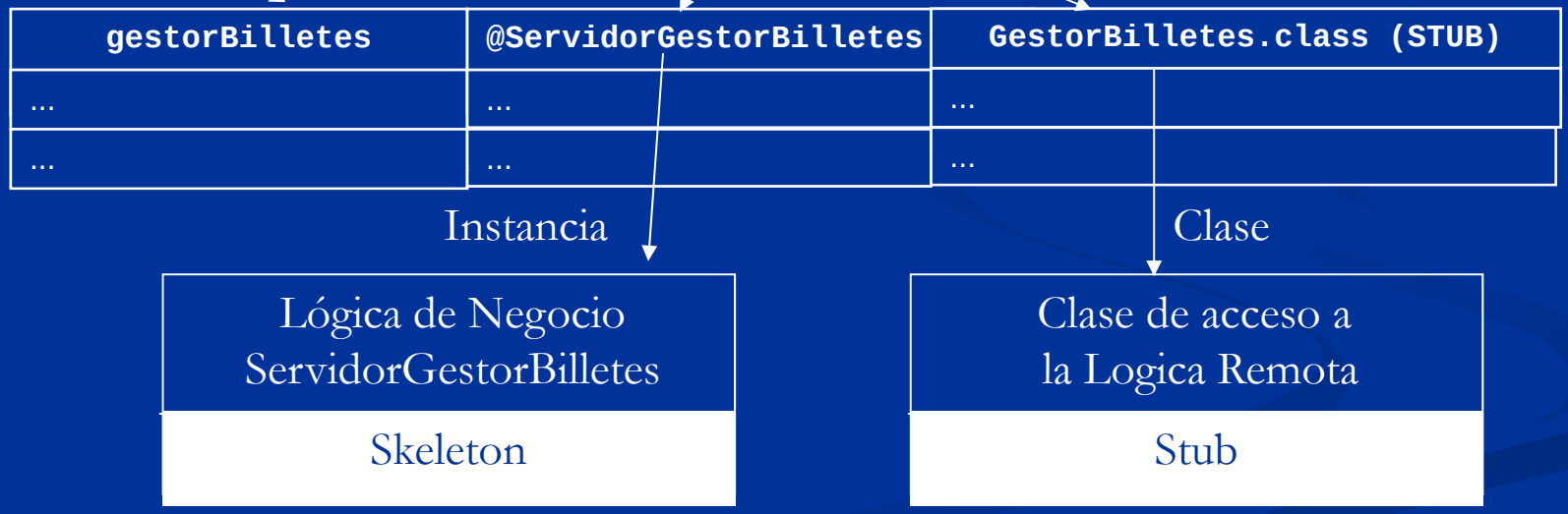
EL STUB: GestorBilletes.class tiene que quedar disponible para la máquina cliente.

Arquitectura RMI

Relación entre Stub y Skeleton

rmiregistry – rebind (Servidor)

```
java.rmi.registry.LocateRegistry.createRegistry(p)
GestorBilletes objetoServidor =new ServidorGestorBilletes();
String nombreServicio = "//localhost/gestorBilletes";
                // "//localhost:NumeroPuerto/NombreServicio";
// Registrar servicio remoto
Naming.rebind(nombreServicio,objetoServidor);
```



Arquitectura RMI

Relación entre Stub y Skeleton

rmiregistry – lookup (Cliente)

```
GestorBilletes objetoRemoto =
 (GestorBilletes)Naming.lookup("rmi://IP_Servidor/gestorBilletes");
```

en puerto 1099, por defecto

El método lookup, devuelve del registro una instancia de acceso a la lógica remota(Stub).
Esta instancia será la responsable de comunicarse (a través del Skeleton) con el objeto servidor.

puerto 1099

Maquina virtual Java ubicada en `IP_Servidor`

RMIREGISTRY

gestorBilletes	@ServidorGestorBilletes	GestorBilletes
...	...	...
...	...	...

```
Naming.rebind("//localhost/GestorBilletes",objetoServidor);
```

en puerto 1099, por defecto

Arquitectura RMI

Gestor de seguridad

- Un programa Java debe especificar un gestor de seguridad que determine su política de seguridad.
- Algunas operaciones requieren que exista dicho gestor. En concreto, las de RMI.
  - RMI sólo cargará una clase Serializable desde otra máquina si hay un gestor de seguridad que lo permita
  - Se puede establecer un gestor de seguridad por defecto para RMI de la siguiente manera:

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

Arquitectura RMI
Gestor de seguridad

- El gestor de seguridad por defecto de RMI utiliza una política muy restrictiva.
  - Sólo se pueden ejecutar STUBs del CLASSPATH local
- Se puede cambiar, indicando otro fichero de política de seguridad:

System.*setProperty*("java.security.policy",
"c:\\Mipath\\java.policy");

Contenido del fichero **`java.policy`**:

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

Arquitectura RMI

Gestor de seguridad

- También se puede indicar para cada ejecución cuál es la política de seguridad (esto es, usar la opción
  `-Djava.security.policy = `*`fichero_PolíticaSeguridad`*`)`

- O bien, el fichero **`java.policy`**, hay dejarlo en un directorio concreto, conocido por la máquina virtual. De esa manera, todas las aplicaciones lanzadas con esa máquina virtual usarán dicha política de seguridad
  - Si la máquina virtual que se ejecuta es esta:
    `DIRECTORIO_JDK_O_JRE\bin\java.exe`
  - El fichero **`java.policy`** hay que dejarlo en
    `DIRECTORIO_JDK_O_JRE\lib\security`

Ejemplo: Servidor Remoto accede a BD

```
import java.rmi.*;
import java.sql.*;

import java.rmi.server.UnicastRemoteObject;
import java.util.*;
public class ServidorGestorBilletesBD
    extends UnicastRemoteObject implements GestorBilletes {
 private static  Connection conexion;
 private static  Statement sentencia;
 public ServidorGestorBilletesBD() throws RemoteException{
  try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   conexion=DriverManager.getConnection("jdbc:odbc:Bill");
   sentencia=conexion.createStatement();
  conexion.setAutoCommit(false);// Habrá que hacer COMMITs
  } catch(Exception e)
   { System.out.println(”Error: "+e.toString());}
  }
public int getBillete(String nom) throws RemoteException {..
```

Ejemplo: Servidor Remoto accede a BD

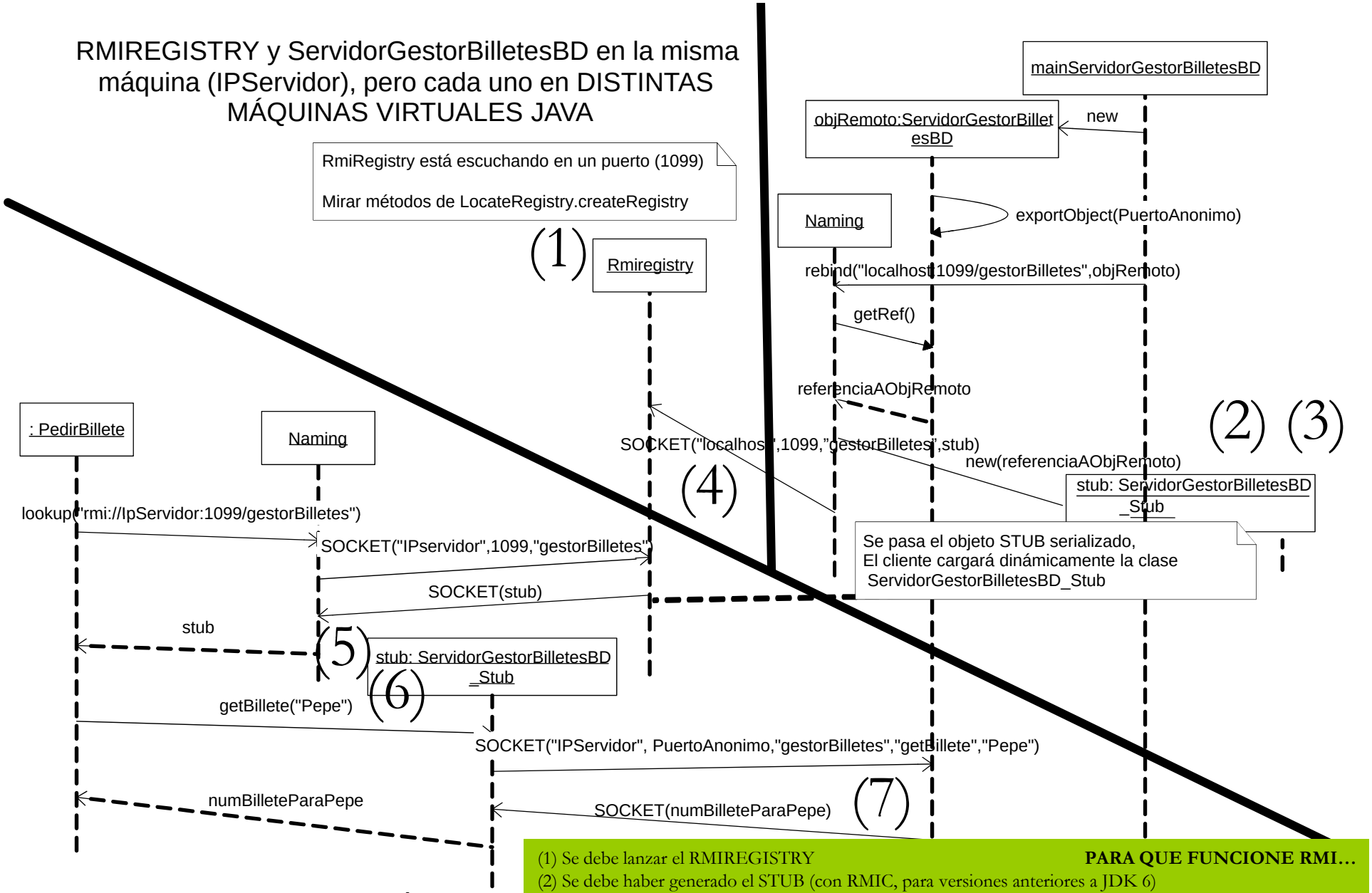
```
// Método main de ServidorGestorBilletesBD.java
public static void main(String[] args) {
System.setProperty("java.security.policy“,"F:\\iso\\java.policy");
System.setSecurityManager(new RMISecurityManager());
try { java.rmi.registry.LocateRegistry.createRegistry(1099);
} catch (Exception e)
{System.out.println(“Rmiregistry ya lanzado“+e.toString());}
try {
 ServidorGestorBilletesBD objetoServidor =
            new ServidorGestorBilletesBD();

 String maquina = "//localhost/";
          // o bien el nombre IP: //sisd00.si.ehu.es/
 String servicio = "gestorBilletes";
 String servicioRemoto = maquina+servicio;

 // Registrar el servicio remoto
 Naming.rebind(servicioRemoto,objetoServidor);
}catch (Exception e)
   {System.out.println("Error: "+e.toString());}
}}
```

Conexión entre nivel de presentación, lógica del negocio y datos

- Hasta ahora hemos considerado que en el nivel de presentación, el objeto con la lógica del negocio se encuentra en un atributo (de tipo interface Java)
- Utilizando RMI se puede seguir con esa misma idea, pero en este caso la interfaz es remota
- En vez de asignarle al objeto de presentación, el objeto con la lógica del negocio se puede hacer que sea el objeto de presentación quien lo busque (usando lookup).
- Cambiar la lógica del negocio consiste en sustituir un objeto por otro en el servidor.



PresentacionRemoto en otra máquina

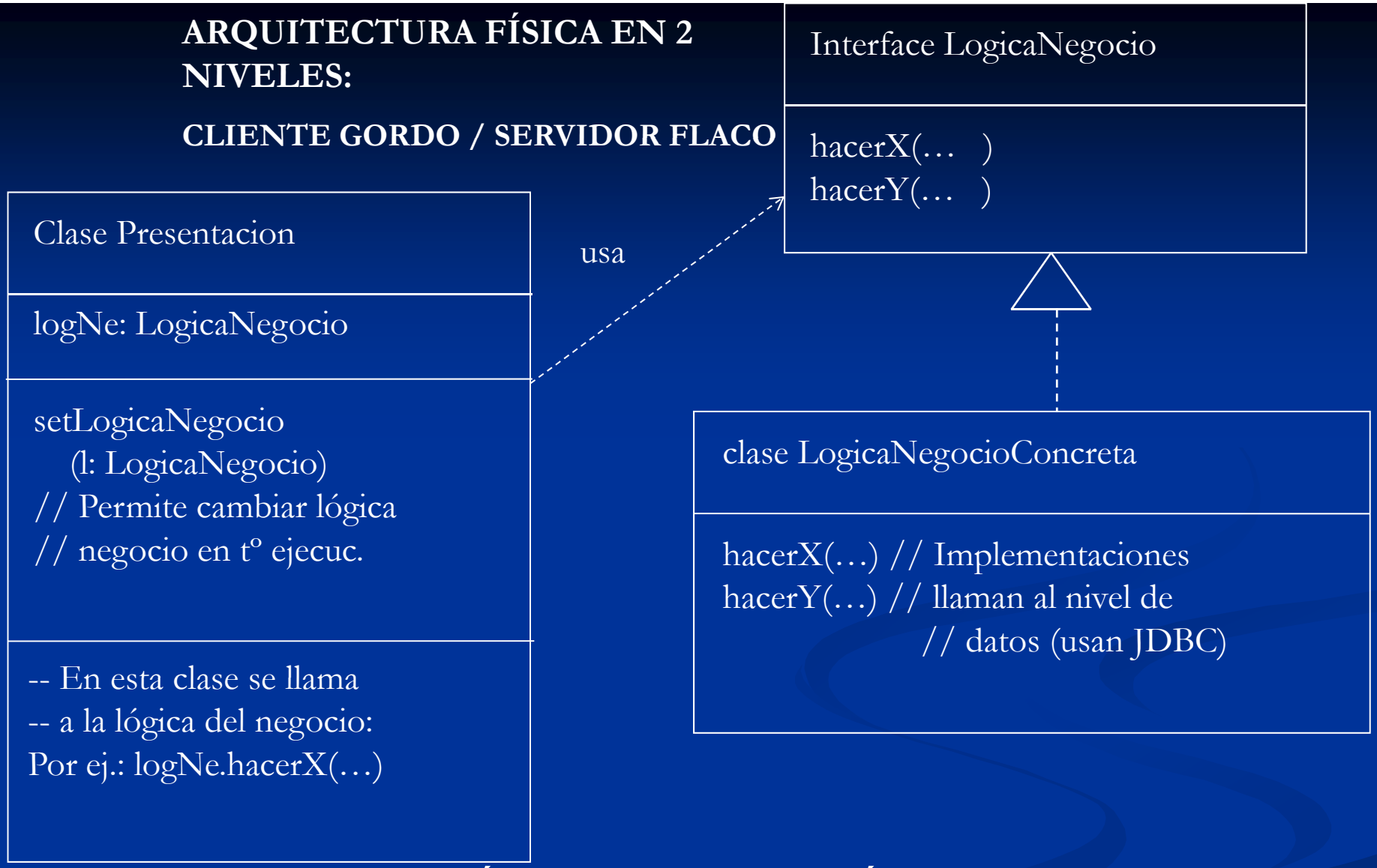
A. Goñi. Dpto. LSI, UPV/EHU

PARA QUE FUNCIONE RMI…

(1) Se debe lanzar el RMIREGISTRY
(2) Se debe haber generado el STUB (con RMIC, para versiones anteriores a JDK 6)
(3) La clase STUB debe estar accesible en el CLASSPATH del SERVIDOR
(4) Se debe permitir el USO DE SOCKETS (POLÍTICA DE SEGURIDAD)
(5) La clase STUB debe estar accesible en el CLASSPATH del CLIENTE, o bien, se debe lanzar indicando de dónde descargarlo (CODEBASE)
(6) Se debe permitir la DESCARGA DINÁMICA DE CLASES (POLÍTICA DE SEGURIDAD)
(7) Los parámetros y resultados de los métodos remotos deben ser de clases SERIALIZABLES

ARQUITECTURA FÍSICA EN 2 NIVELES:

CLIENTE GORDO / SERVIDOR FLACO



CREAR LA INTERFAZ GRÁFICA Y ASIGNAR LÓGICA DEL NEGOCIO:

```
Presentacion p = new Presentacion();
p.setLogicaNegocio(new LogicaNegocioConcreta());
p.setVisible(true);
```

ARQUITECTURA FÍSICA EN 3 NIVELES usando RMI