



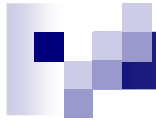
# Remote Method Invocation (RMI)

Basado en: Fundamentals of  
RMI. Short  
Course. JGuru.



# Paradigmas de desarrollo

- Desarrollo basado en protocolos
  - Se diseña el protocolo
  - Se elaboran los módulos para manejo del protocolo
  - Se hace la aplicación en función del envío y procesamiento (*parsing*) de estos mensajes
- Desarrollo basado en Funcionalidad
  - Se diseña la aplicación
  - Se define el servicio en función de firmas de métodos
  - Se implementa
    - El cliente en forma casi tradicional
    - En el servidor sólo se colocan las funciones remotas



# Programacion tradicional

```
Prog x {  
    void f(){  
        Y = suma(a,b);  
    }  
  
    int suma(int h, int g){  
        Return h + g;  
    }  
}
```



# Programacion RMI

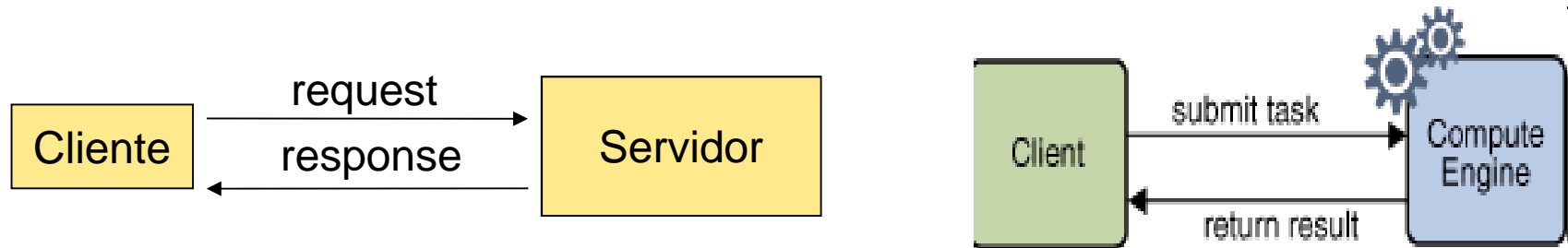
## Cliente

```
Prog x {  
    ...  
    ...  
    Y = suma(a,b);  
    ...  
}
```

## Servidor

```
int suma(int f, int g) {  
    return f + g;  
}  
  
int resta(int f, int g) {  
    return f - g;  
}
```

# ¿Cómo Comunicamos Objetos Remotos en un Sistema Distribuido?

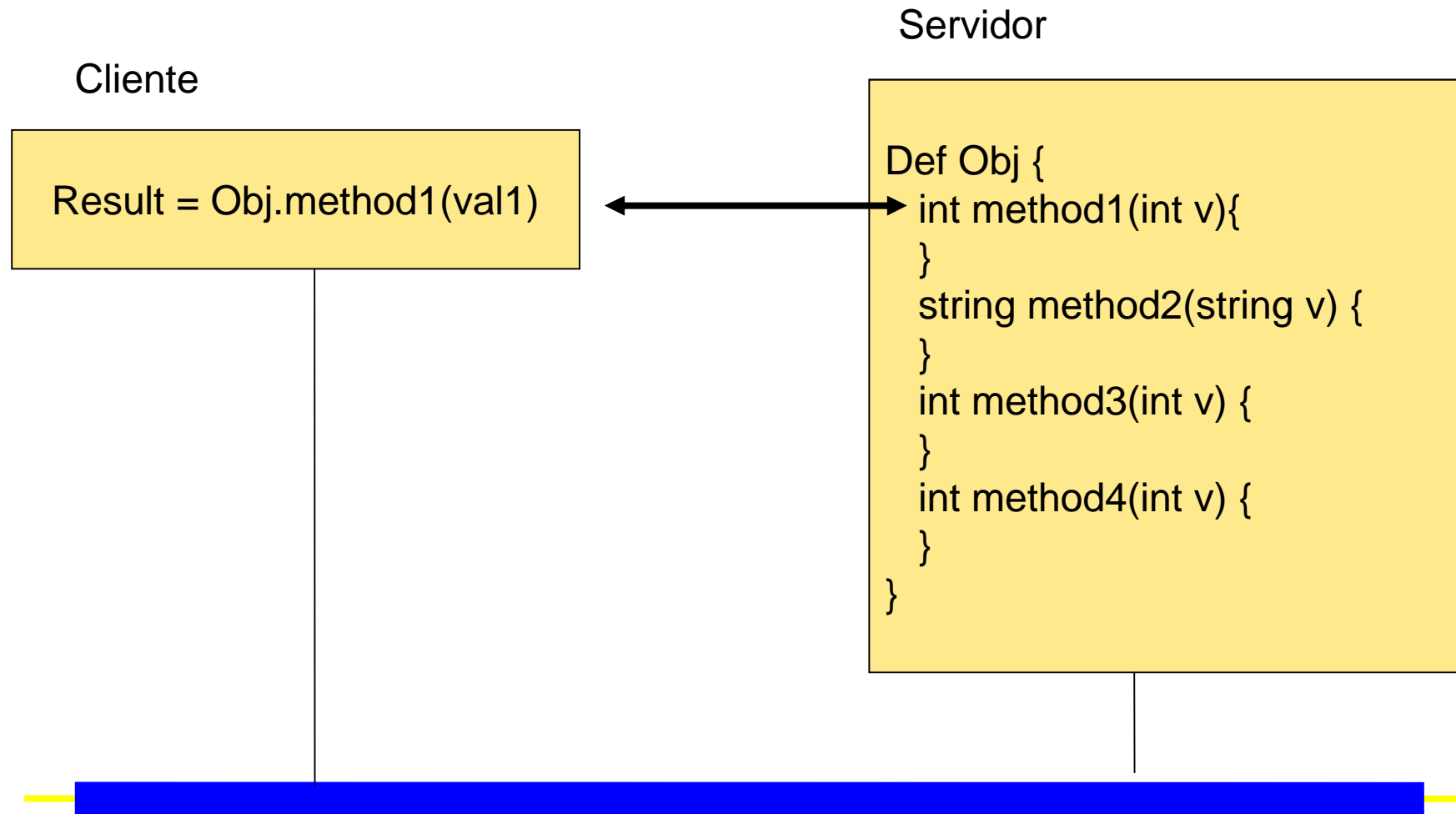


- ¿ Cómo permitir comunicación entre objetos situados en diferentes máquinas ?
- ¿ Cómo llamar a métodos de programas que corren en máquinas diferentes ?

## RMI ( Remote Method Invocation )

- La invocación del servicio remoto es a partir de un objeto (creado en el cliente) que llama a un método (implementado en el servidor)
- *La idea es tener un contexto de ejecución parecido al local bajo una arquitectura cliente/servidor*

# Qué deseamos hacer con objetos remotos





# Metas a Alcanzar con RMI

- La meta principal de los diseñadores de RMI fue permitir a los programadores el desarrollo de programas distribuidos en JAVA, usando la misma sintáxis y semántica de los programas no distribuidos.
- La arquitectura de RMI define la forma como deben interactuar los objetos, cómo y cuándo ocurren las excepciones, cómo se administra la memoria y cómo pasan y retornan los parámetros a/desde los métodos remotos.

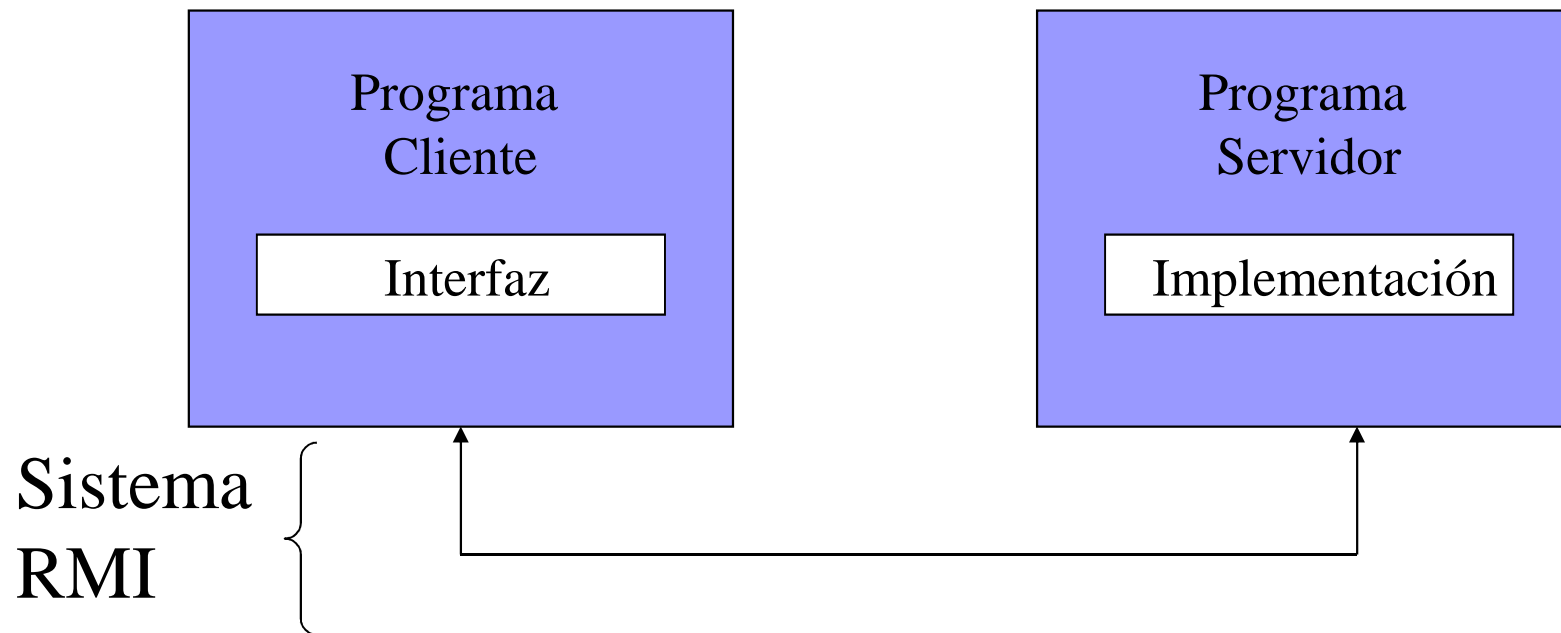


# La Arquitectura de JAVA RMI

- Interfaces:
  - La definición de un comportamiento y su implementación son conceptos separados. En RMI, el código que define el comportamiento y el código que lo implementa están y se ejecutan en diferentes JVMs
  - La definición de un servicio remoto se hace usando una interfaz de JAVA . La implementación se codifica en una clase.



# La Arquitectura de JAVA RMI: La Interfaz

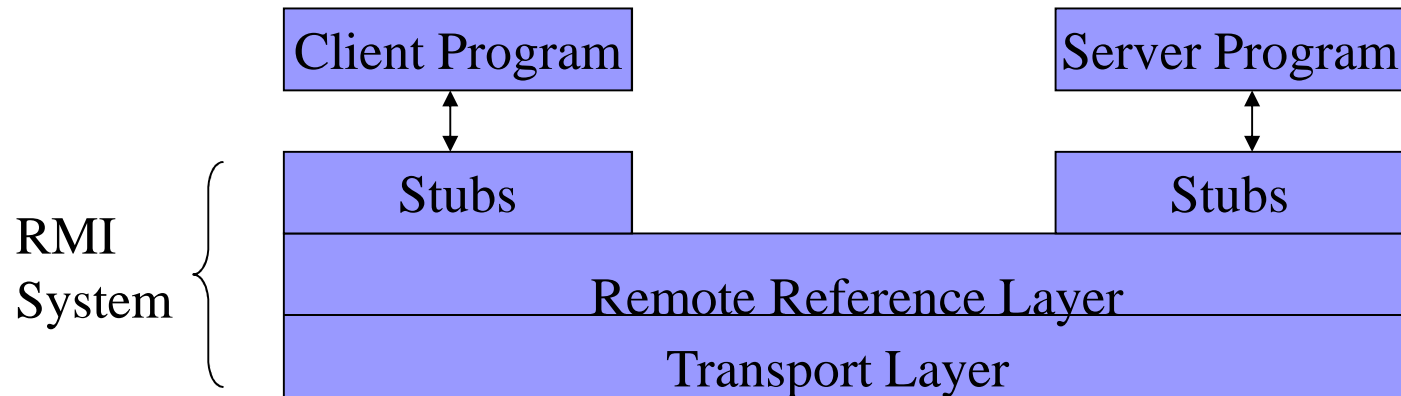




# La Arquitectura de JAVA RMI: Los Stubs

- El cliente necesita hacer transparente la llamada a métodos remotos => *stubs*
- El *stub* paquetiza los parámetros (*marshalling*) y los codifica en un formato estándar independiente de los procesadores involucrados en la aplicación cliente/servidor
- El *stub* da la impresión de localidad al cliente, asegurando transparencia
- El servidor:
  - desempaqueta los parámetros (*unmarshalling*),
  - llama al método invocado,
  - paquetiza los valores de retorno o excepción (*marshalling*)
  - envía información al cliente.

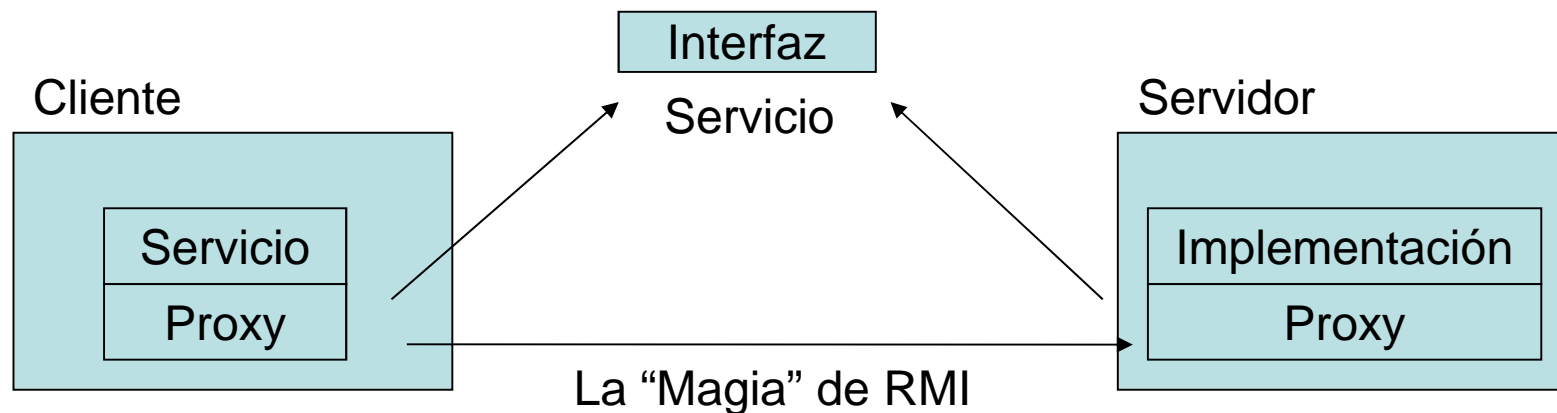
# Niveles Arquitectónicos de RMI



- Nivel del Stub y el Skeleton: están justo por debajo de la vista del desarrollador.
- Intercepta las llamadas que hace el cliente y las redirecciona a un servicio RMI remoto.

# Niveles Arquitectónicos de RMI

- RMI soporta 2 clases que implementan la misma interfaz. La primera implementa el servicio y se ejecuta en el servidor. La segunda actúa como un proxy para el servicio remoto y se ejecuta en el cliente.



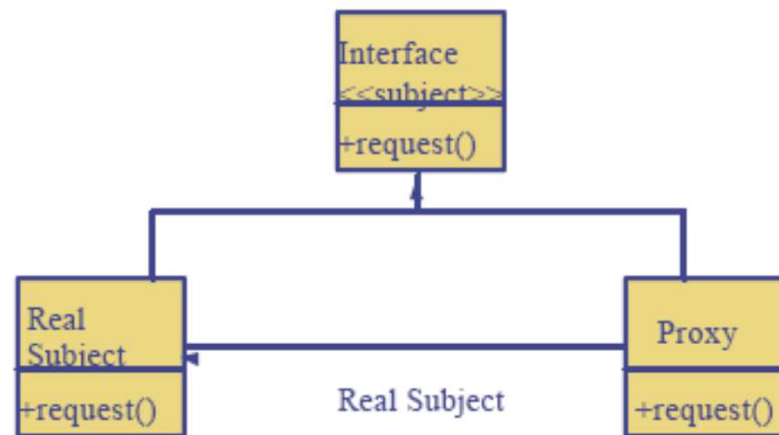


# Niveles Arquitectónicos de RMI

- Nivel de referencia remota (Remote Reference Layer). Esta al tanto de cómo interpretar y administrar las referencias que realizan los clientes a objetos remotos.
- El Nivel de Transporte: se basa en conexiones TCP/IP entre máquinas en una red. Ofrece conectividad básica, así como algunas estrategias para atravesar cortafuegos.

# Nivel de los Stubs

- En este nivel RMI usa el patrón de diseño de Proxy (*Proxy design pattern*). En este patrón un objeto en un contexto es representado por otro (el proxy) en un contexto diferente. El proxy sabe cómo dirigir (*to forward*) las invocaciones a métodos entre los distintos objetos participantes .





# Nivel de Referencia Remota

- Los niveles de referencia remota definen y soportan la semántica de la invocación de una conexión RMI. Este nivel ofrece un objeto RemoteRef que representa el enlace al objeto que alberga la implementación del servicio remoto.
- Los stubs usan el método invoke() para dirigir (*to forward*) la llamada. El objeto RemoteRef entiende la semántica de invocación para servicios remotos.



# Nivel de los Stubs

- Un Stub es una clase auxiliar generada por RMI, que “entiende” cómo comunicarse con otro Stub a través del enlace RMI.
- Un Stub mantiene una conversación con el otro Stub; lee los parámetros de la llamada, efectúa la invocación al servicio remoto (to the remote service implementation object), acepta el valor de retorno y lo retorna al stub.



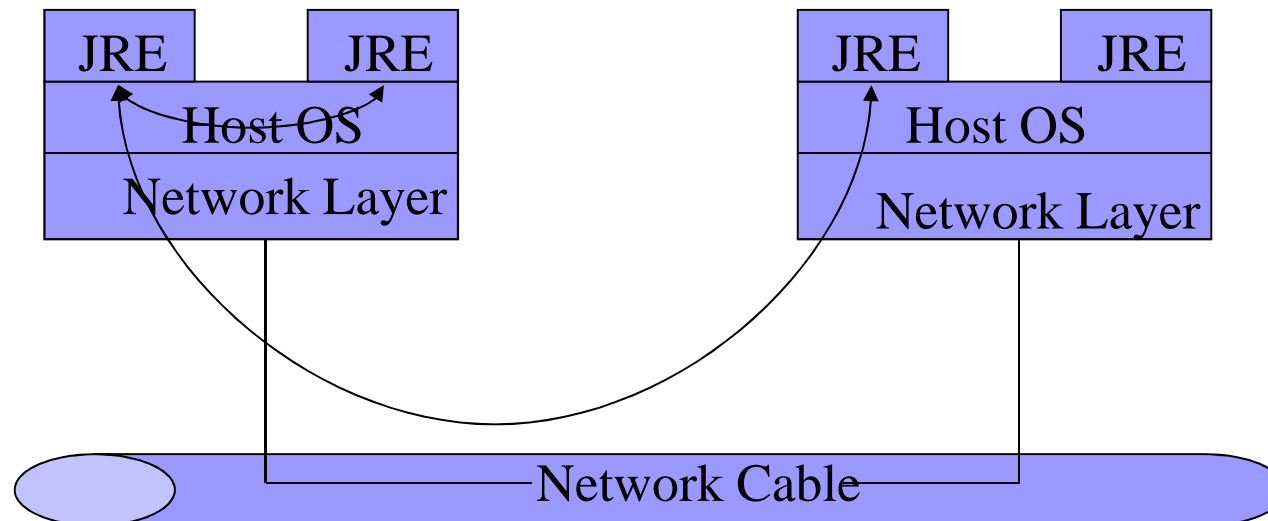


# Nivel de Transporte

- El nivel de transporte realiza la conexión entre JVMs. Todas las conexiones son stream-based y usan TCP
- Aún si dos JVMs se están ejecutando en la misma computadora, ellas se conectan a través del protocolo TCP/IP.

# Nivel de Transporte

- Por encima de TCP/IP, RMI usa un wire level protocol llamado Java Remote Method Protocol (JRMP).
- JRMP es un protocolo propietario, stream-based que está especificado para dos versiones. La primera se liberó con JDK 1.1 y requiere del uso de clases Skeleton en el servidor. La segunda versión se liberó con Java 2 SDK. Se ha optimizado para mejorar el desempeño por lo que no requiere de las clases skeletons.





# Nombrar objetos remotos

- Cómo un cliente encuentra un servicio remoto?
- R: Usando un servicio de nombres o directorios.
- RMI puede usar diferentes tipos de servicios de directorio, incluyendo JNDI (the Java Naming and Directory Interface).
- RMI incluye un servicio muy simple llamado: RMI Registry, rmiregistry. Este corre en cada máquina que posee objetos de servicio remoto y acepta preguntas sobre el servicio. Se ejecuta en el puerto 1099.
- En un host, un servidor crea un servicio remoto , construyendo primero un objeto local que implementa el servicio.
- Luego, se exporta el objeto a RMI. Cuando el objeto es exportado, RMI crea un servicio que espera que los clientes se conecten para solicitar el servicio (listening service).
- Después de exportar el objeto , el servidor lo registra en el RMI registry con un nombre público.



# Nombrar los objetos remotos

- Del lado del cliente, el RMI Registry se accede a través de la clase estática Naming. Ésta provee el método lookup() que acepta un URL donde se especifica el nombre del host del servidor y el nombre del servicio. El método retorna una referencia remota al objeto que ofrece el servicio. El URL tiene la forma:

`rmi://<host_name> [:<name_service_port>] /<service_name>`

- El name\_service\_port sólo debe especificarse si el servicio deseado corre en un puerto diferente al 1099.



# Un Sistema RMI

Un sistema RMI se compone de varias partes:

- Definiciones de Interfaces para servicios remotos.
- Implementación de los servicios remotos.
- Archivos Stub.
- Un servidor donde se encuentran los servicios remotos.
- Un servicio de nombres para que los clientes puedan encontrar los servicios remotos.
- Un proveedor de clases (A class file provider, an HTTP or FTP server)
- Un programa cliente que requiere de servicios remotos.

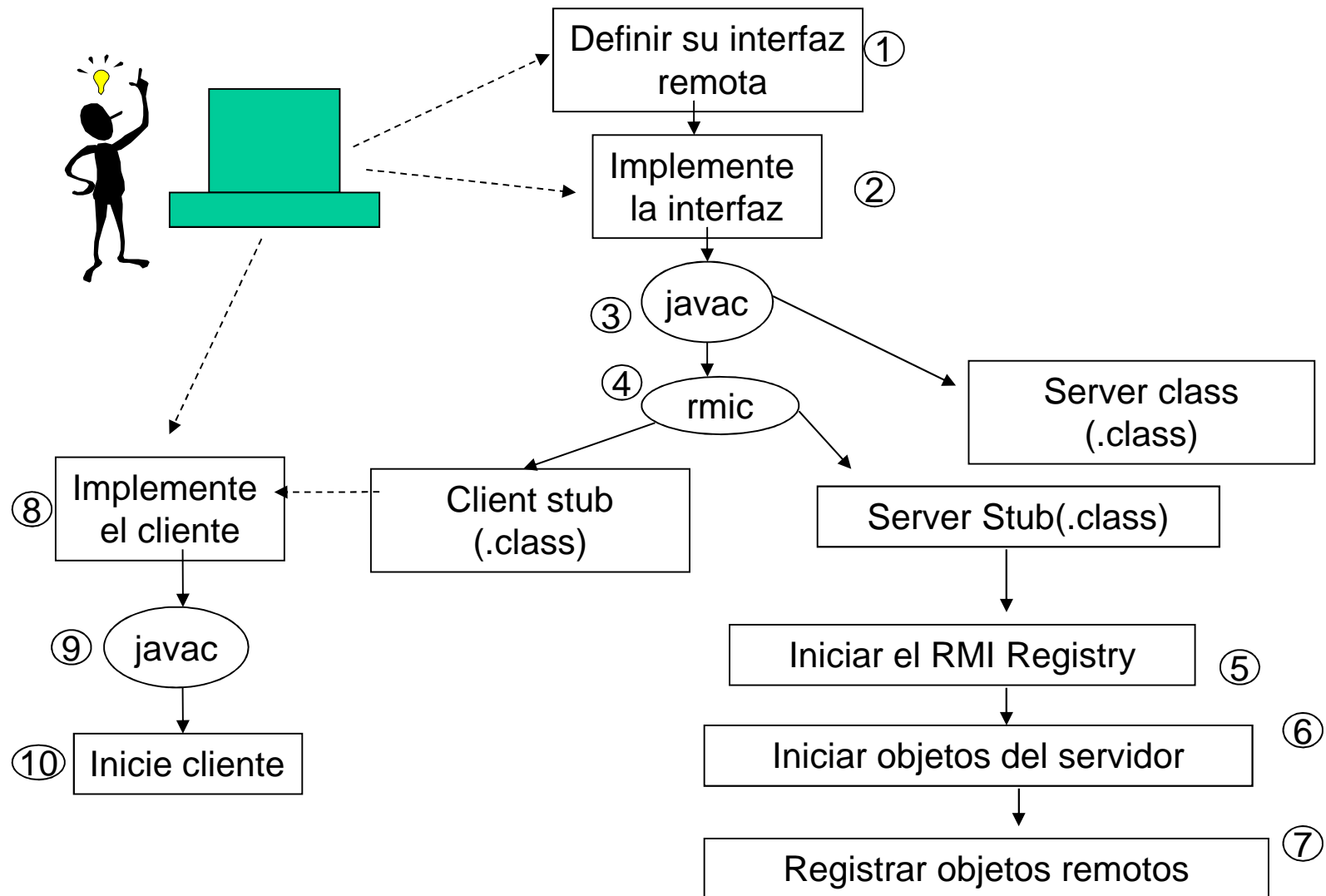


# Cómo usar RMI

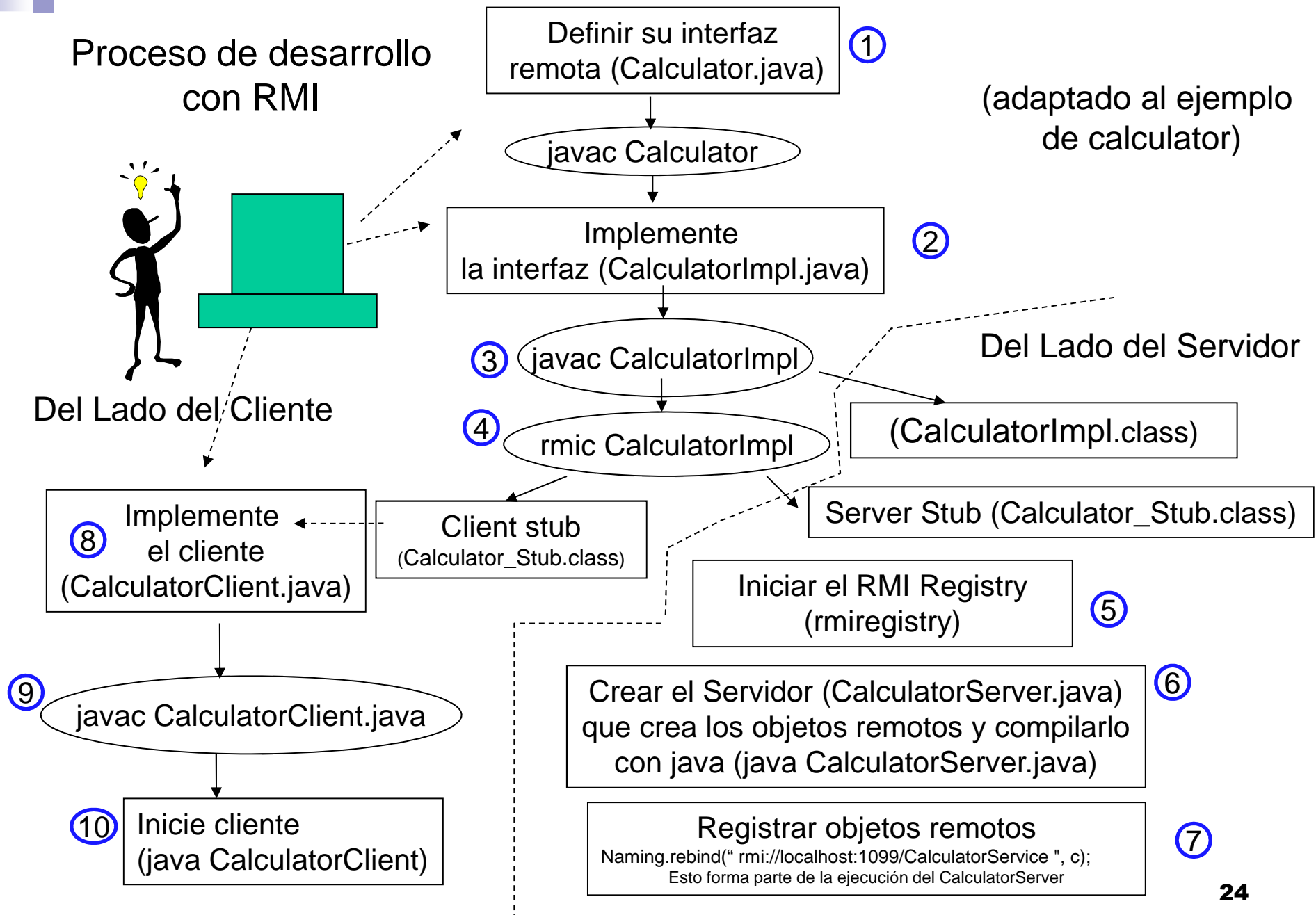
Pasos para construir una aplicación que usa RMI:

1. Escriba y compile el código JAVA para las interfaces.
2. Escriba y compile código JAVA para las clases del tipo implementación.
3. Genere los archivos Stub. (esto ya no hace falta)
4. Escriba código JAVA para un servicio remoto.
5. Desarrolle código Java para el cliente RMI
6. Instale y corra su sistema RMI

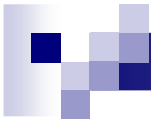
# Proceso de desarrollo con RMI



## Proceso de desarrollo con RMI







# 1 Interfaces

El primer paso es escribir y compilar el código JAVA para las interfaces.

```
public interface Calculator
    extends java.rmi.Remote {
    public long add(long a, long b)
        throws java.rmi.RemoteException;
    public long sub(long a, long b)
        throws java.rmi.RemoteException;
    public long mul(long a, long b)
        throws java.rmi.RemoteException;
    public long div(long a, long b)
        throws java.rmi.RemoteException;
}
```

>javac Calculator.java



## 2 Implementación

```
public class CalculatorImpl
    extends
        java.rmi.server.UnicastRemoteObject
    implements Calculator {
    // Implementations must have an explicit
    // constructor in order to declare the
    // RemoteException exception
```

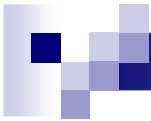
```
    public CalculatorImpl( )
        throws java.rmi.RemoteException {
        super();
    }
```

```
    public long add(long a, long b)
        throws java.rmi.RemoteException {
        return a + b;
    }
```

```
    public long sub(long a, long b)
        throws java.rmi.RemoteException {
        return a - b;
    }
```

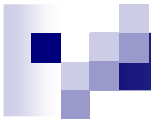
```
    public long mul(long a, long b)
        throws java.rmi.RemoteException {
        return a * b;
    }
```

```
    public long div(long a, long b)
        throws java.rmi.RemoteException {
        return a / b;
    }
}
```



## 2 Implementación

- Copie el código anterior en su directorio y compile.
- La clase donde está la implementación usa el objeto `UnicastRemoteObject` para enlazarse al sistema RMI. Una clase que no extienda de este objeto puede usar su método `exportObject()` para enlazarse a RMI.
- Cuando una clase extiende del `UnicastRemoteObject`, debe proveer un constructor que declare que ésta puede enviar una excepción `RemoteException`.



## 5 Stubs y skeletons

- Use el compilador de RMI, `rmic`, para generar el stub y el skeleton.  
    `>rmic CalculatorImpl`
- Después de ejecutar `rmic`, Ud. Debería encontrar el archivo `Calculator_Stub.class` y si está ejecutando Java 2 SDK, debería obtener también el archivo `Calculator_Skel.class` (ya no es así).

## 6 Implementación

- La clase CalculatorServer es un servidor muy simple que ofrece el núcleo esencial para hosting.

```
import java.rmi.Naming;
public class CalculatorServer {
    public CalculatorServer() {
        try {
            Calculator c = new CalculatorImpl();
            Naming.rebind(" rmi://localhost:1099/CalculatorService ", c);
        }
        catch (Exception e) {
            System.out.println ("Trouble: " + e);
        }
    }

    public static void main(String args[]) {
        new CalculatorServer();
    }
}
```



## 8 El Cliente

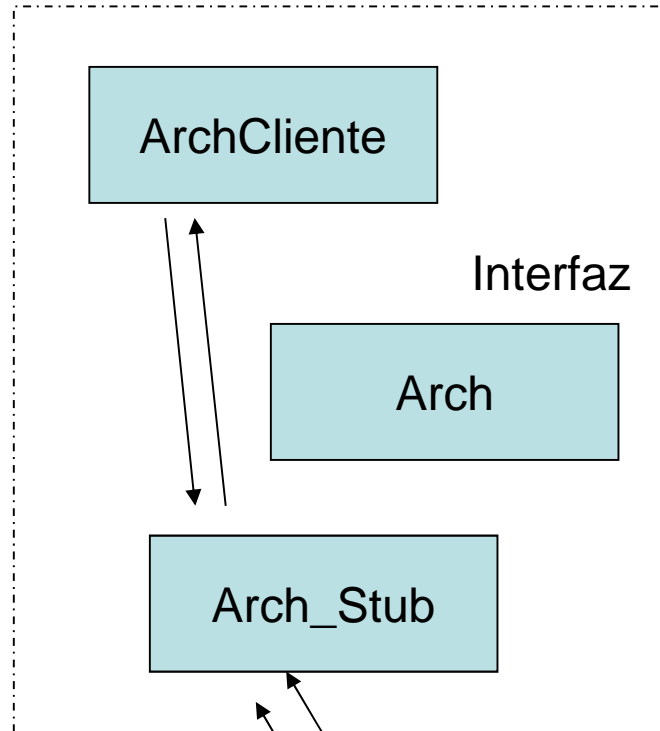
```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Calculator c = (Calculator)
            Naming.lookup( "rmi://remotehost/CalculatorService");
            System.out.println ( c.sub(4, 3) );
            System.out.println ( c.add(4, 5) );
            System.out.println ( c.mul(3, 6) );
            System.out.println ( c.div(9, 3) );
        }
    }
}
```

```
        catch (MalformedURLException murle ) {
            System.out.println ();
            System.out.println (
                "MalformedURLException");
            System.out.println ( murle ); }
        catch (RemoteException re) {
            System.out.println ();
            System.out.println ( "RemoteException");
            System.out.println (re); }
        catch (NotBoundException nbe) {
            System.out.println ();
            System.out.println ("NotBoundException");
            System.out.println (nbe);}
        catch (java.lang.ArithmeticException ae) {
            System.out.println ();
            System.out.println ("java.lang.Arithmetic
                Exception");
            System.out.println (ae);}
    }
}
```

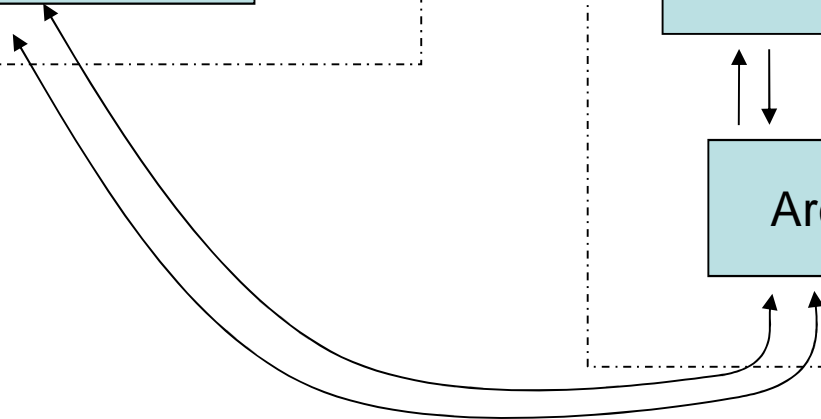
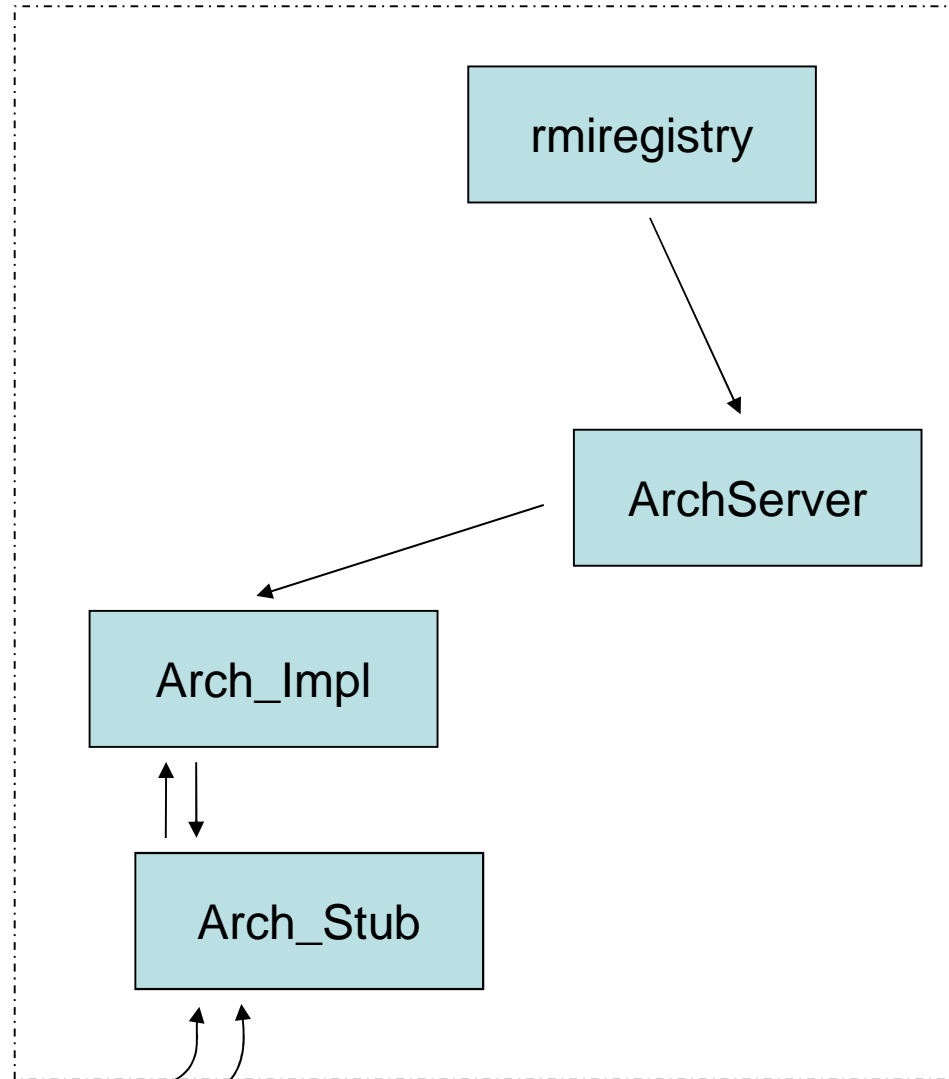


# RMI

Cliente



Servidor



# Ejecución del Sistema RMI

- Se necesita 3 consolas : una para el servidor, una para el cliente y una para el RMIRegistry.
- Comience con el Registry. Debe estar en el directorio que contiene las clases que Ud. ha escrito , introduzca :
  - > rmiregistry (5)
- En la segunda consola inicie el servidor:
  - >java CalculatorServer (6)
- En la última consola ejecute el cliente.
  - >java CalculatorClient (10)
- Si todo está bien la salida del programa será:

1  
9  
18  
3

Cliente

Servidor





# Ejecución del Sistema RMI (Tips)

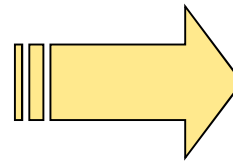
- Se puede sustituir la ejecución del servicio `rmiregistry` por separado, si en el programa servidor se incluye la instrucción:

```
java.rmi.registry.LocateRegistry.createRegistry(1099);
```

- A partir de la versión de java 1.6 no se requiere el uso de `rmic` para generar los stubs.

# Carga dinámica de clases

El cliente requiere cargar las clases y excepciones porque el servidor puede cambiarlas, según su evolución en el tiempo



Carga dinámica de clases con un ClassLoader

Sin embargo ...

- esto implica problemas de seguridad

En consecuencia ...

- se necesita al *security manager* para impedir la llegada de virus a través de los *stubs*



# Comparación Operativa

## RPC - RMI

- rpcgen
- Archivo.x
- Archivo\_srp.c
- Archivo\_svc.c
- Portmapper
- XDR
- rmic
- Interfaz Archivo.java
- ArchivImpl.java
- ArchivoServer.java
- rmiregistry
- Serialización de Objetos




# Pase de Parámetros

- Paso de parámetros en un JVM:
  - La semantica normal para el pase de parámetros en Java es el paso por valor.
  - Los valores obtenidos de la ejecución de los métodos también son copiados.
  - Los objetos son pasados por referencia .
- Pase de Parámetros en RMI
  - Datos Primitivos: RMI pasa los datos primitivos por valor.
  - Objetos:RMI envía a los objetos mismos, no su referencia.
  - RMI emplea serialización para transformar los objetos a un formato que pueda ser enviado por cables de red.
- RMI introduce un tercer tipo de parámetro a considerar: Objetos remotos.
  - Un programa cliente puede obtener la referencia a un objeto remoto mediante el RMI Registry o mediante el resultado de la invocación a un método.



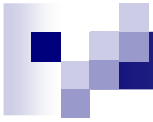
# Comparación de Programas Java Distribuidos y No Distribuidos.

	Objeto Local	Objeto Remoto
Definición	Una clase de Java define a un objeto local.	Un objeto remoto se define por una interfaz que debe extender de la interfaz remota.
Implementación	La clase Java implementa al objeto local	El comportamiento del objeto remoto lo ejecuta una clase que implementa la interfaz remota.
Creación	Con el operador new.	Se crea en el host con el operador new. Un cliente no puede crear directamente un objeto remoto.
Acceso	Una variable que referencia al objeto	Una variable que referencia a un proxy stub de la interfaz remota
Referencias	Una referencia al objeto apunta directamente a un objeto en el heap	Una referencia remota es un apuntador a un objeto proxy (un stub) en el heap local.



# Comparación de Programas Java Distribuidos y No Distribuidos.

	Objeto Local	Objeto Remoto
Referencias Activas	Un objeto se considera "vivo" si existe al menos una referencia a él	Se considera que un objeto remoto tiene una referencia remota activa si ha sido accedido dentro de un cierto período de tiempo ( <i>lease period</i> ). Si las referencias se han eliminado explícitamente o si el <i>leased period</i> ha expirado, el objeto remoto está a la disposición del <i>garbage collector</i> .
Garbage Collection	Un objeto local es candidato a GC cuando todas sus referencias han sido eliminadas	No hay referencias remotas y las referencias locales han sido eliminadas explícitamente.
Excepciones	El compilador de JAVA obliga a los programas a manejar las excepciones	RMI obliga a los programas a manejar cualquier excepción remota posible.
Finalización	Se debe implementar el método <i>finalize</i>	Se debe implementar la interfaz <i>unreferenced</i>



FIN