Asignatura 780014
Programación Avanzada

TEMA 8 –
PROGRAMACIÓN CONCURRENTE DISTRIBUIDA:
RPC Y RMI

Programación distribuida: RPC y RMI

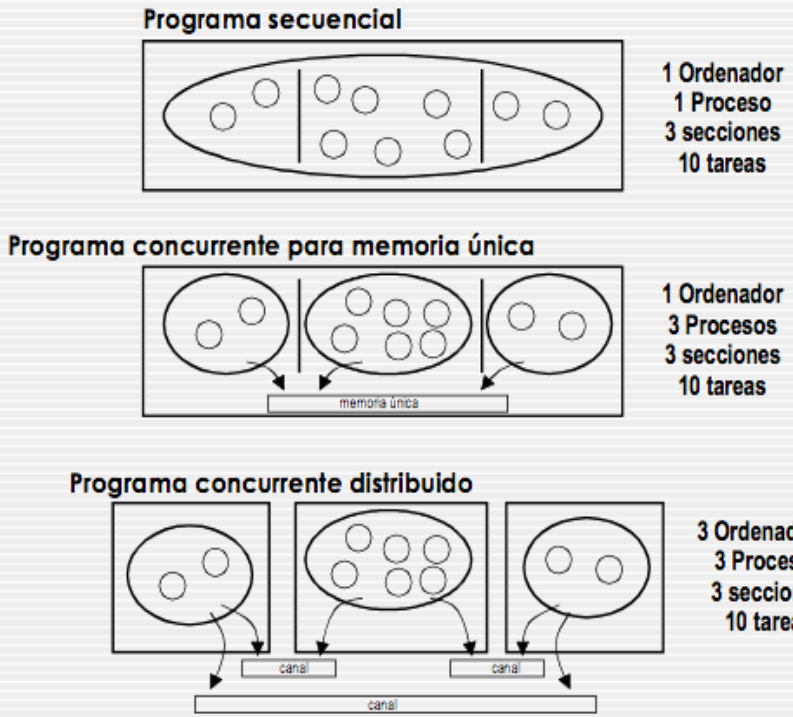
- Objetivo del tema:
  - Conocer el RPC como tipo de programación distribuida, así como su implementación mediante RMI en Java

Índice

1. RPC
   - Tipos de RPC
2. Arquitectura Cliente-Servidor
3. RMI
   - Capas
   - Funcionamiento

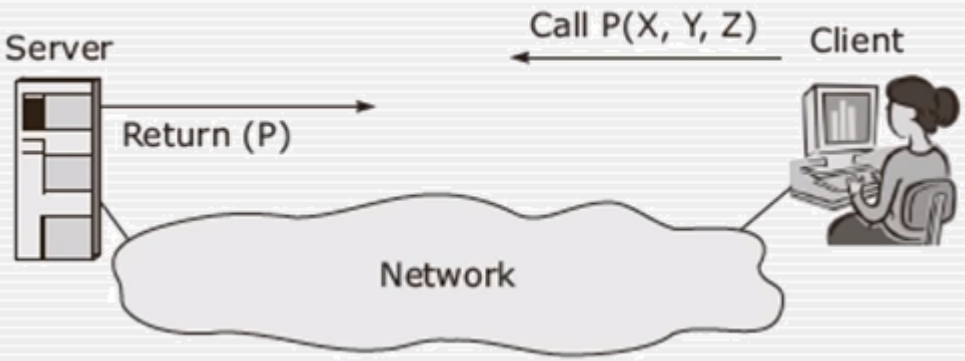
Recordamos

- Computación distribuida
  - Comunicación y sincronización entre procesos
    - **Concurrentes**
    - En **distintas** máquinas



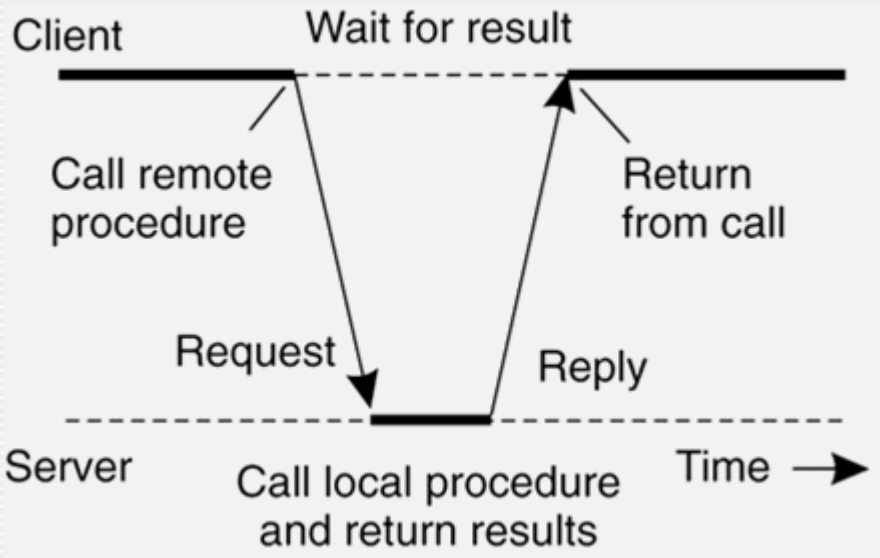
RPC

- RPC (*Remote Procedure Call*):
  - Comunicación a un nivel de **abstracción superior** al del paso de mensajes
    - **No** se manejan **directamente** los canales
    - Haremos llamadas a **procedimientos definidos en <u>otros</u> procesos**



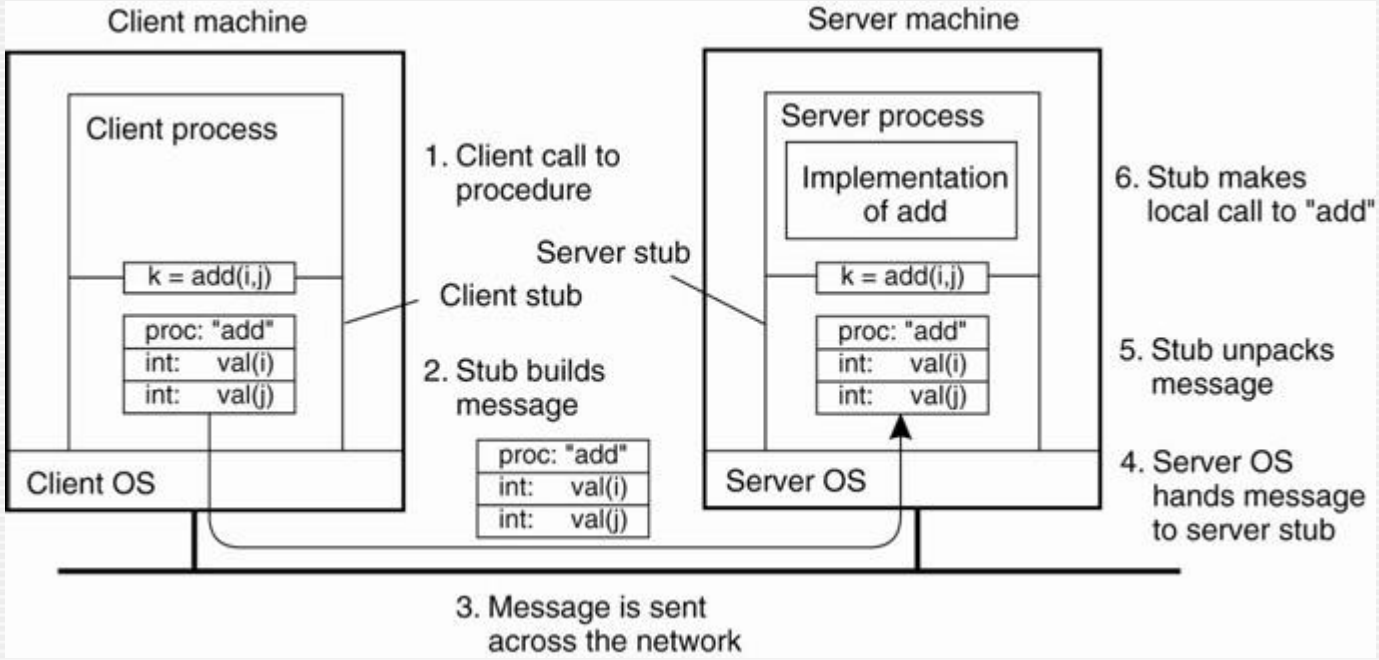
RPC

- La comunicación ahora se consigue mediante la **llamadas a procedimientos remotos** (definidos en otros procesos)
- Los **parámetros** de la llamada serán los datos enviados/recibidos
- La propia llamada implica **sincronización**
  - Los **canales** utilizados siempre serán **síncronos**



RPC

- Funcionamiento general:



RPC

- Comportamiento del **proceso que llama**:

1. **Ejecuta la llamada** RPC escrita por el programador
2. Se produce la **creación** de los canales de forma **automática**
3. Se **envía** la información al receptor
4. Se produce un **bloqueo voluntario** hasta que la información llegue al receptor, que puede no estar aún disponible, éste la procese y responda
5. Se **recibe** la respuesta y se carga en los parámetros de salida, si los hubiera
6. Se pasa a la **siguiente instrucción** a la llamada RPC

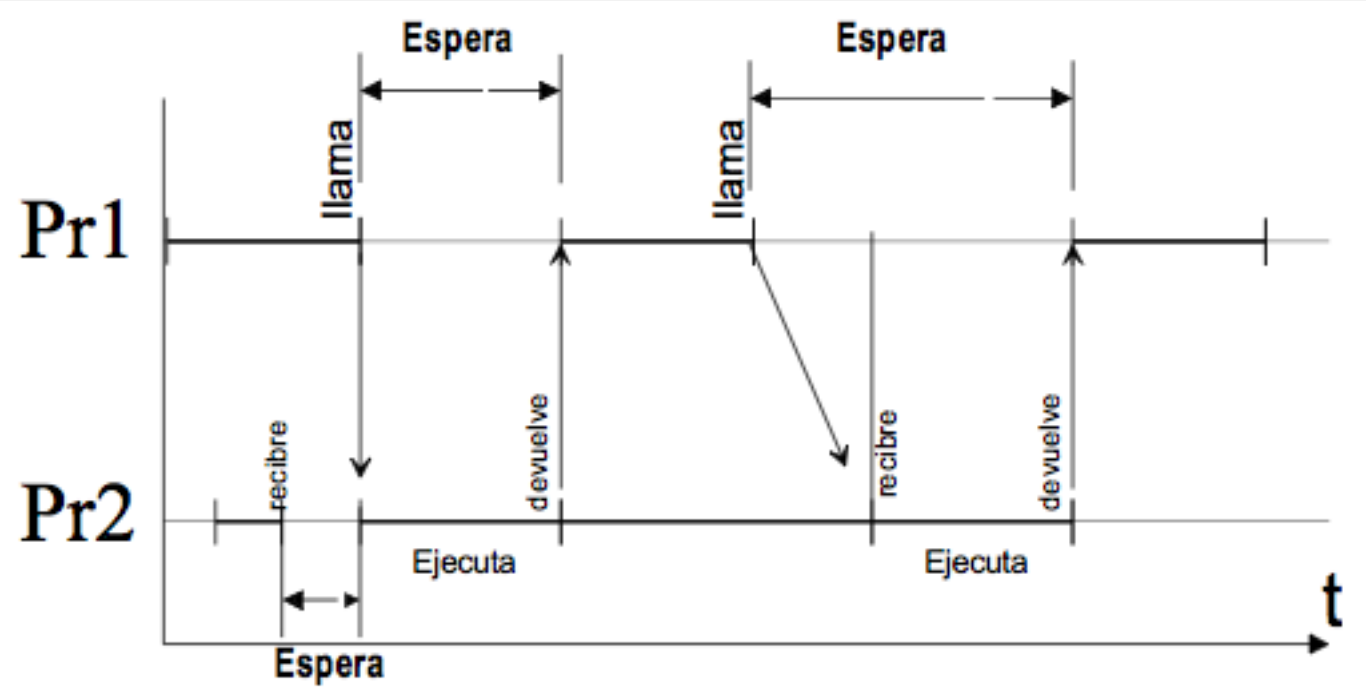
RPC

- Comportamiento del **proceso** que se ofrece **para ser llamado**:

1. **Ejecuta la oferta** RPC escrita por el programador
2. Se produce la **creación** de los canales de forma **automática**
3. Se espera a **recibir** la información del llamador
   - Si no estuviera disponible, se **bloqueará** el proceso
4. Una vez recibida la llamada, se carga en los parámetros y se **ejecuta** el código escrito por el programador
5. Se **envía** la respuesta como resultado de la ejecución
6. Se pasa a la **siguiente instrucción** a la oferta RPC

RPC

- El **proceso que llama** siempre **espera** la ejecución del procedimiento invocado
- Además, se produce **espera** debido al carácter **síncrono** de la comunicación



RPC

- El modelo de RPC original no es práctico porque:
  - El **proceso que se ofrece** queda **bloqueado** a la espera de llamadas
    - Esto elimina toda opción de que el "servidor" realice tareas de gestión o de control
  - El proceso que se ofrece usa una sentencia normal en mitad del código no RPC, no distribuido
    - Debido a que se planteó como una modificación de lenguajes existentes
    - No permite crear infraestructura de gestión de la distribución
- Solución: un nuevo modelo siguiendo la arquitectura Cliente/Servidor

Tipos de RPC

- Hay distintos tipos de RPC (implementaciones):
  - ONC RPC: Llamada a procedimientos remotos de Sun
  - DCE/RPC: Llamada a procedimientos remotos de OSF (*Open Software Foundation*)
  - DCOM: Modelo de Objetos de Componentes Distribuidos de Microsoft
  - **RMI**: Invocación de Métodos Remotos de **Java**

- **Ninguno** de estos protocolos **son compatibles** entre sí

Tipos de RPC

- A pesar de la incompatibilidad, hay puntos de coincidencia:
  - La mayoría de ellos utilizan un **lenguaje de descripción de interfaz** (IDL)
    - **Define** los **métodos exportados** por el servidor
  - Actualmente se utiliza XML como lenguaje para definir el IDL, y HTTP como protocolo de red
    - Se crea el concepto de Servicio Web
      - Por ejemplo SOAP y XML-RPC

Cliente - Servidor

- Se basa en ideas similares a RPC:
  - Existe una **oferta** para ejecutar un cierto código
  - Existe un **proceso que demanda** la ejecución de dicho código
- Sin embargo:
  - El modelo Cliente-Servidor se **independiza de la implementación** concreta que se realice
  - No todos los procesos van a ser iguales:
    - Procesos **cliente**
    - Procesos **servidor**

Cliente - Servidor

Cliente: proceso que solicita la realización de tareas al servidor. Realiza la petición de un servicio

Servidor: proceso cuya misión es ofrecer algo (servicios) a otros procesos. Está escuchando la petición de algún servicio por parte de algún cliente

Servicio: tarea que se ofrece a realizar un servidor cuando le sea solicitada por un cliente mediante una petición

- Sólo los **procesos** de tipo **cliente iniciarán** la comunicación hacia los procesos servidores, que les ofrecen sus servicios

Cliente - Servidor

- Características:
  - Un servidor tendrá en su interior un **bucle** infinito denominado "**de servicio**"
  - Un servidor **ofrecerá uno o más servicios** disponibles **permanentemente** para cualquier cliente (en lugar de ofrecerlo en un momento puntual)
  - El número y tipo de servicios **puede cambiar** con el tiempo
  - Un proceso servidor **no realiza otras tareas** distintas a la de ofrecer servicios (excepto las de inicialización del servidor)

Cliente - Servidor

- **Motivos** por los que un **proceso** se establece como **servidor**:
  - La existencia de un **recurso especial y único** al que sólo puede estar conectado un ordenador y que debe ser usado por otros
    - El proceso servidor **ofrecerá** acceso al recurso mediante la ejecución de **servicios**
  - La posibilidad de **reducir** el número de ciertos **recursos a uno** creando una configuración como la anterior para reducir gastos
    - Ejemplo: impresoras
  - La necesidad de una **potencia de cálculo elevada** en momentos puntuales que es más recomendable **concentrar** en un solo equipo
    - Y que **los demás** le **soliciten** la ejecución del **trabajo pesado** cuando sea preciso
  - La necesidad de un nivel de **seguridad** elevado
    - Si se **concentra** en un equipo ciertos recursos, se puede realizar una inversión en seguridad más **efectiva y barata**

Cliente - Servidor

- Tareas:
  - Cliente:
    - Inicia la comunicación
    - Solicita un servicio al servidor
  - Servidor:
    - Espera peticiones
    - Proporciona el servicio solicitado

Ejemplos:

-Un cliente web solicita una página
-Un proceso P2P solicita un fichero a otro proceso P2P

-El servidor web envía la página solicitada por el cliente
-El proceso P2P envía el fichero solicitado por otro proceso P2P

Cliente - Servidor

- Los **clientes** serán iguales a los **procesos de llamada** RPC
- El **rendimiento** de Cliente-Servidor será **mejor** que en RPC original:
  - Los **clientes no** tienen que **esperar** al servidor
- La **espera** de un servidor que no tiene peticiones se considera que **no** es **activa**
  - La **red** se encarga de **despertar** al proceso servidor cuando recibe alguna petición por los canales asignados
- El modelo **Cliente-Servidor** es totalmente **desacoplado**
  - Lo único que necesita conocer un cliente de un servicio es el nombre, los parámetros y el servidor donde reside

Cliente - Servidor

- Normalmente un **servicio** se ofrece a través de un **canal** asociado a un "**puerto**", identificado mediante un número
- Los puertos son **direcciones lógicas** proporcionadas por el sistema operativo para poder diferenciar varios canales (no confundir con los puertos HW)
- Existen puertos preestablecidos para las aplicaciones más usadas, como:
  - Puertos 20 y 21 – FTP para transferencia de archivos
  - Puerto 22 – SSH para acceder a servidores mediante intérprete de comandos
  - Puerto 25 – SMTP para envío de correo
  - Puerto 53 – DNS para servicio nombre de dominio
  - Puerto 80 – HTTP para Internet
  - Puerto 110 – POP3 para recibir correos
  - Puerto 119 – NNTP para grupos de noticias

RMI

- Definición RMI (*Remote Method Invocation*):
  - Mecanismo Java para invocar métodos remotos
    - Se **conserva la semántica** de los objetos Java
  - Forma parte del entorno estándar de ejecución de Java
    - Mecanismo simple de comunicación en aplicaciones distribuidas basadas **exclusivamente en Java**
  - Objeto remoto: aquel cuyos métodos pueden ser invocados desde otra JVM (habitualmente, en un ordenador diferente)
    - Se describe en una **interfaz remota** que declara los métodos del objeto remoto

RMI

- Dos tipos de programas (aplicación de objetos distribuidos):
  - **Servidor**
    - **Crea objetos remotos**
    - Hace **visibles** las **referencias** a esos objetos remotos
    - **Espera** a que los clientes invoquen métodos en esos objetos
  - **Cliente**
    - **Obtiene** una **referencia remota** a uno de los objetos remotos del servidor
    - **Invoca métodos** sobre esos objetos remotos
- RMI proporciona el **mecanismo** por el cual cliente y servidor se **comunican** y **pasan información** de un lado al otro

RMI

- Las aplicaciones de objetos distribuidos necesitan:
  - Localizar objetos remotos
    - Una aplicación RMI **registra** y **localiza** sus objetos remotos con el RMIRegistry
      - Es un mecanismo de nombramiento que se encuentra en el servidor y mantiene la información sobre los objetos disponibles (accesibles por clientes con una URL)
  - Comunicarse con los objetos remotos
    - Se encarga RMI: para el programador, es **transparente** (invocación estándar a un método)
  - Cargar el código de la clase para los objetos que se pasan como parámetros o los valores de retorno
    - Como se pueden pasar objetos como parámetros a objetos remotos, RMI proporciona los mecanismos necesarios para **cargar el código** de un objeto y transmitir sus datos

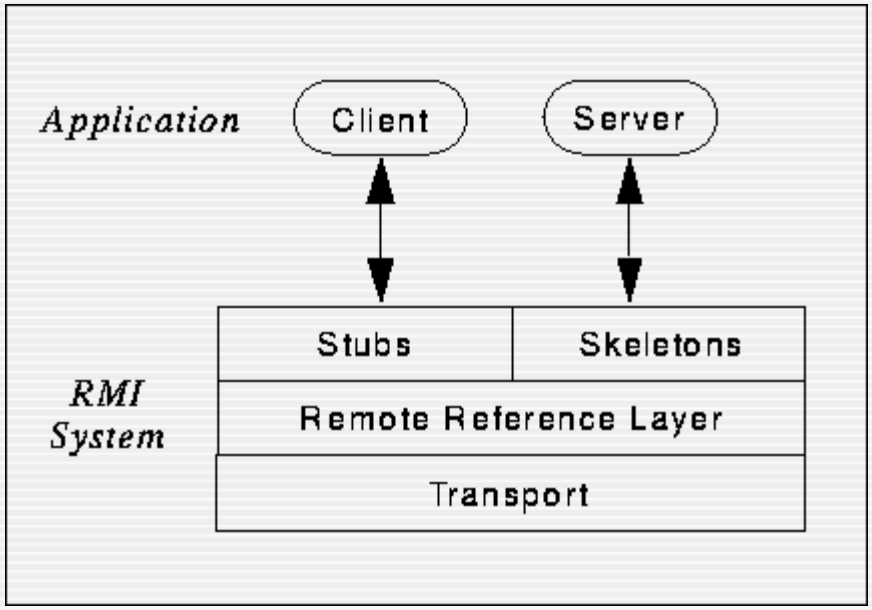
RMI

- Funcionamiento:
  - Un programa **exporta** parte de un objeto (algunos métodos)
    - El objeto y sus métodos publicados estarán **accesibles** a través de la red (mediante un servidor de registro)
      - El programa permanece a la **espera** de peticiones en un puerto **TCP** (gestionado por la librería de Java)
  - Una vez exportado el objeto, un cliente necesita **localizarlo** en una máquina y servidor concreto
    - Una vez localizado (tiene la referencia remota), puede:
      - Conectarse
      - Conseguir acceso al objeto
      - Invocar los métodos proporcionados por dicho objeto

RMI

- Pasos generales de la invocación:
  1. **Encapsulado** de los parámetros (serialización de Java)
  2. **Invocación** del método
     - El invocador se queda **esperando** una respuesta
  3. Al terminar la ejecución, el servidor **serializa** el valor de retorno (si lo hay) y lo **envía** al cliente
  4. El código cliente **recibe** la respuesta y continúa como si la invocación hubiera sido local

RMI – Arquitectura en 4 capas: capa 1

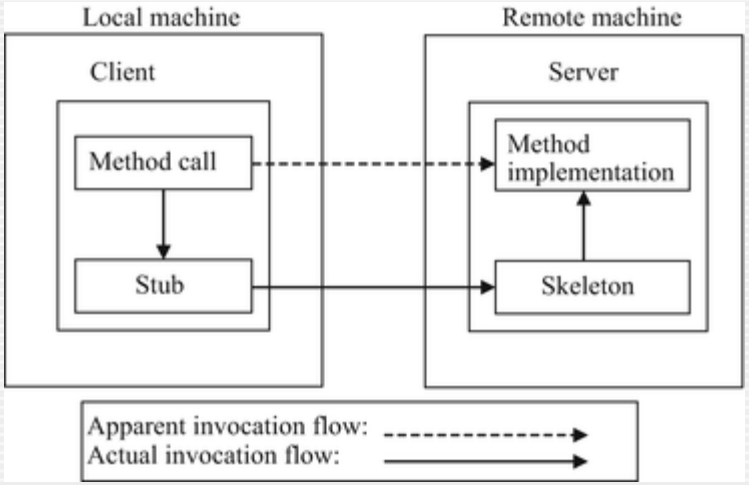


1. **Capa de aplicación:**
   - Implementación de las aplicaciones cliente-servidor
     - **Llamadas alto nivel** para **exportar** objetos remotos y **acceder** a ellos

RMI – Arquitectura en 4 capas: capa 2

2. Capa proxy o capa stub-skeleton

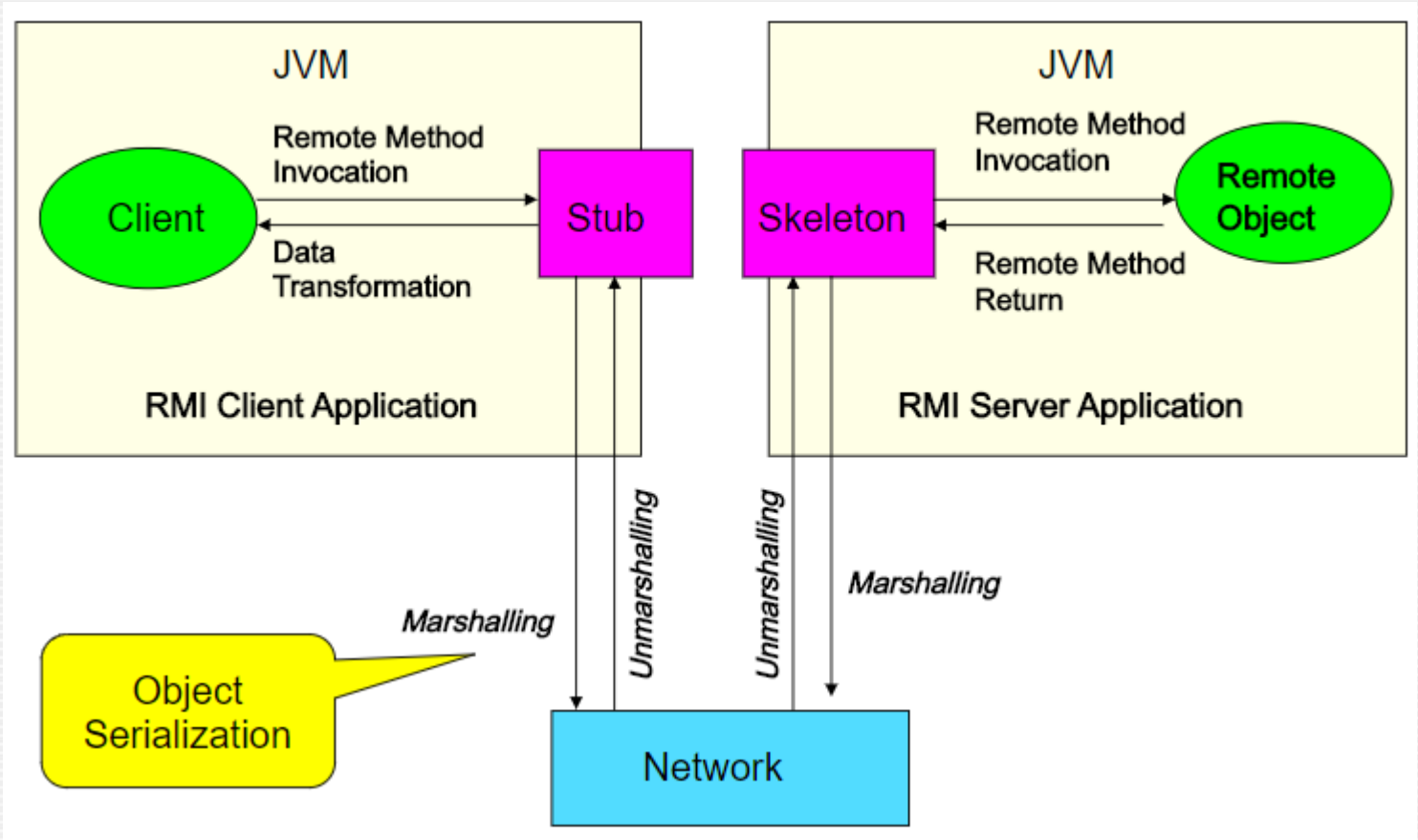
- Esta capa interactúa directamente con la capa de aplicación
- Dota a clientes y servidores de una interfaz que les **permite localizar objetos remotos**
  - La **preparación de parámetros** y **retorno de objetos** tienen lugar en esta capa



RMI – Arquitectura en 4 capas: capa 2

- Utiliza:
  - En el lado del **cliente**: **Stub**
    - Inicia una llamada al objeto remoto (llamando a la capa inferior)
    - Empaqueta (*marshaling*) los parámetros e informa a la capa inferior de que la llamada debería ser invocada
    - (Espera por el resultado de la invocación)
    - Desempaqueta (*unmarshaling*) y devuelve el valor de retorno (o la excepción)
    - Devuelve el valor a quien lo llamó (e informa a la capa inferior de que la llamada se ha completado)
  - En el lado del **servidor**: **Skeleton**
    - Desempaqueta (*unmarshaling*) los parámetros necesarios para la ejecución del método remoto
    - Invoca el método (implementación) del objeto remoto
    - Empaqueta (*marshaling*) el valor de retorno de la llamada (o la excepción, si ocurriera)
    - Avisa a la capa inferior de que lo envíe de vuelta al cliente

RMI – Arquitectura en 4 capas: capa 2



RMI – Arquitectura en 4 capas: capa 3

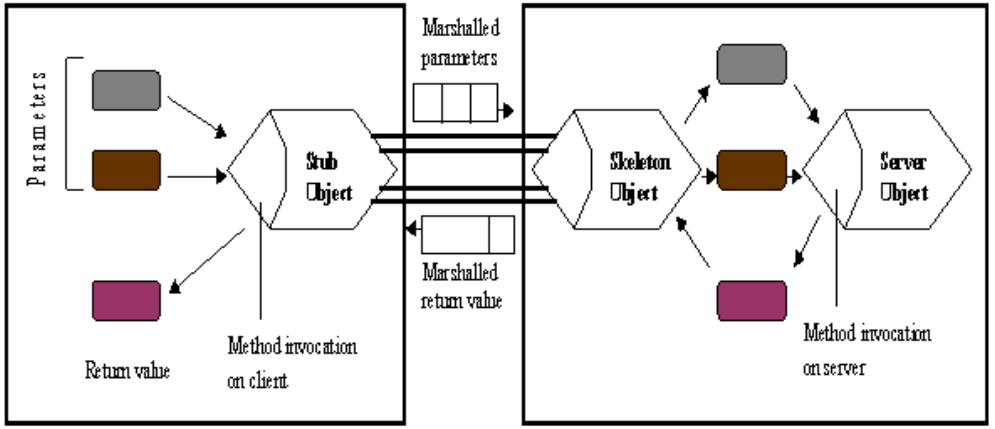
3. Capa de referencias remotas

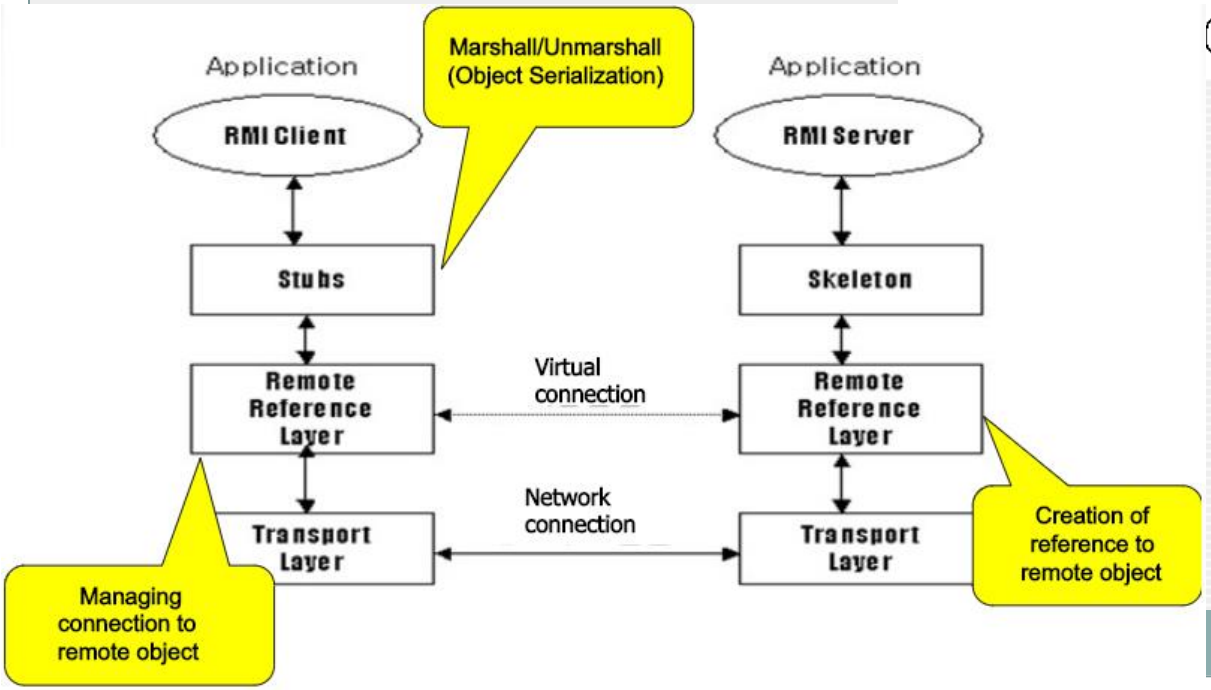
- Esta capa se encarga de la **creación** y **gestión** de las **referencias** a objetos remotos
  - Manteniendo, para ello, una tabla de objetos distribuidos
- Además, convierte las llamadas remotas en peticiones hacia la capa de transporte

RMI – Arquitectura en 4 capas: capa 4

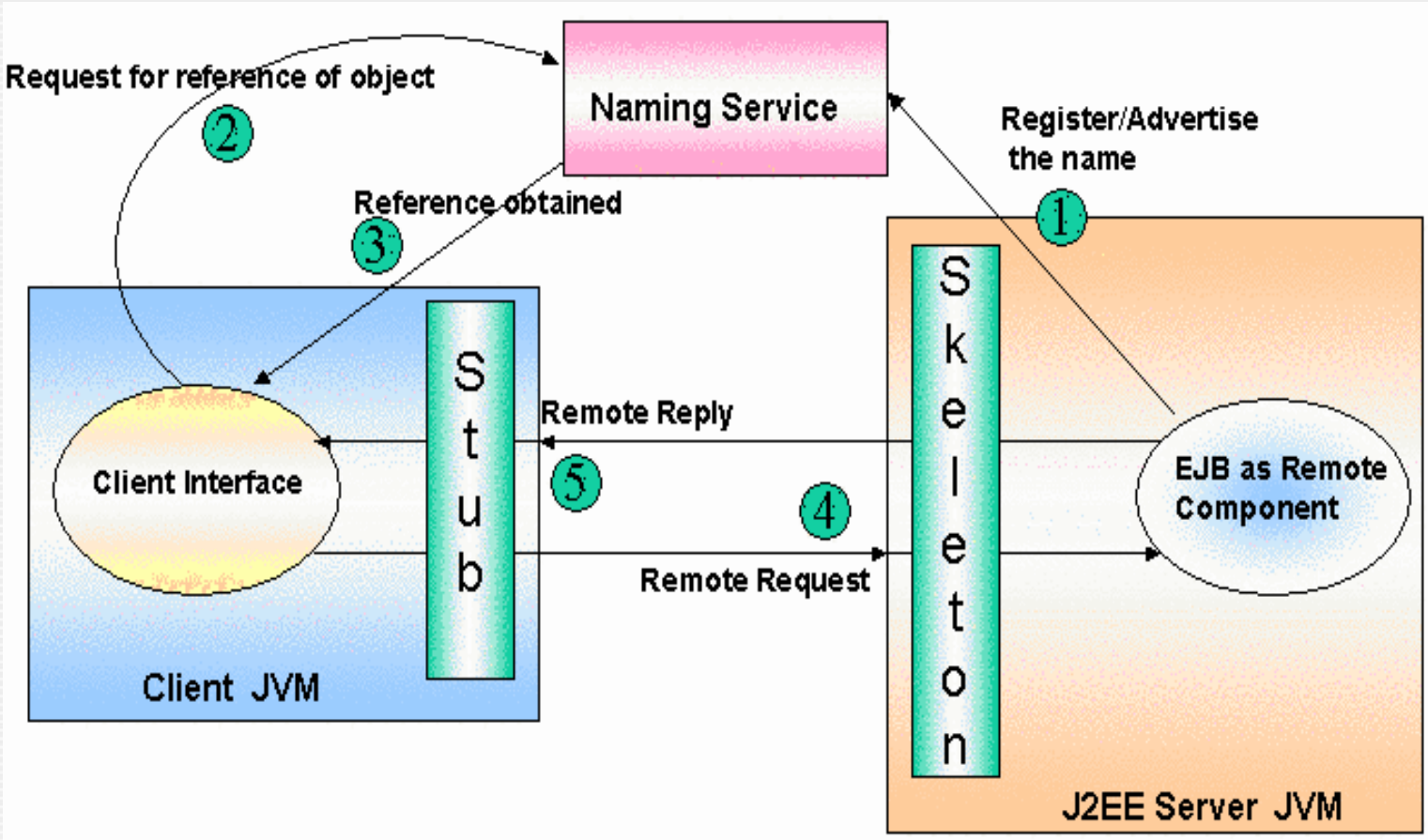
4. **Capa de transporte**
   - Responsable de realizar las conexiones
   - Manejo del transporte de los datos de una máquina a otra
   - El protocolo de transporte subyacente para RMI es **JRMP** (*Java Remote Method Protocol*)
     - Solamente es "comprendido" por programas Java

RMI – Arquitectura en imágenes

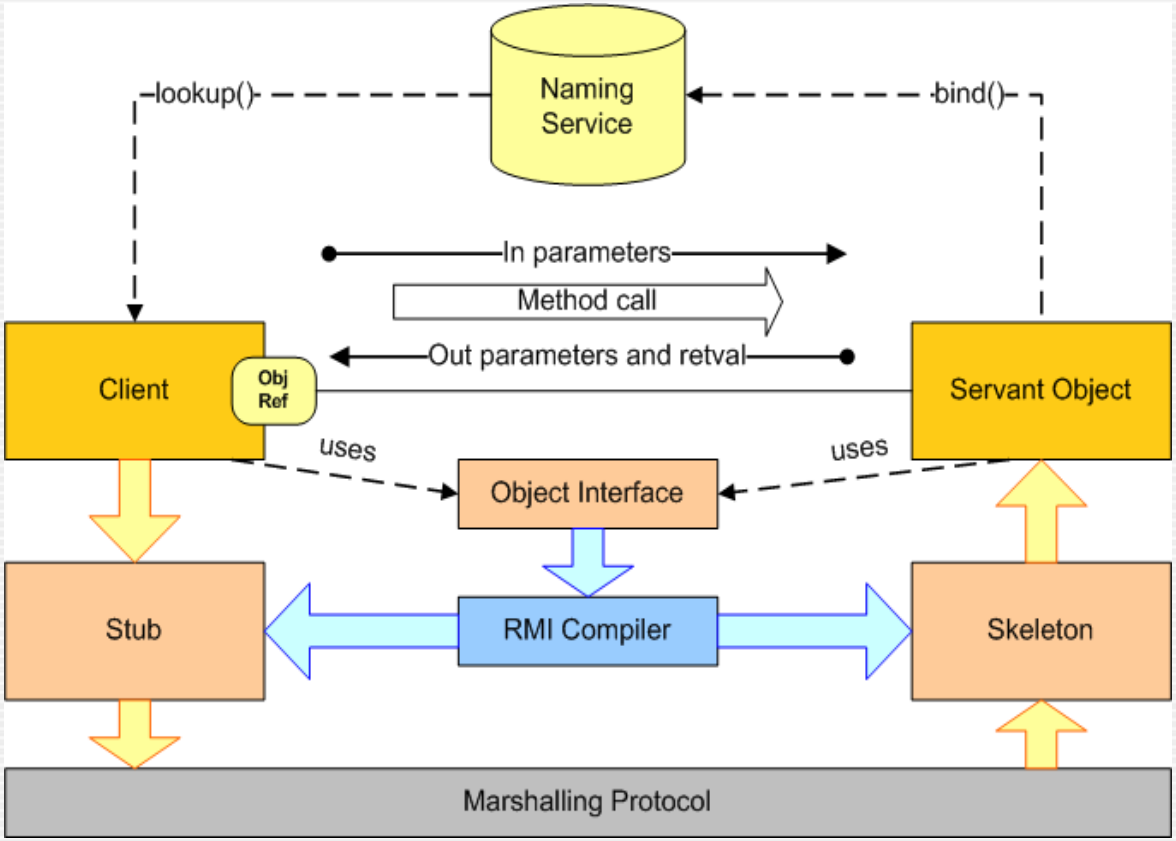




RMI – Arquitectura en imágenes



RMI – Arquitectura en imágenes



RMI – Esquema de una aplicación

- Toda aplicación RMI se descompone en 2 partes:
  - Un **servidor**:
    - Interfaz remota
    - Clase que implementa la interfaz
    - Código principal que crea el objeto remoto y lo registra
  - Un **cliente**:
    - Interfaz remota
    - Código principal que localiza el objeto remoto y llama a sus métodos

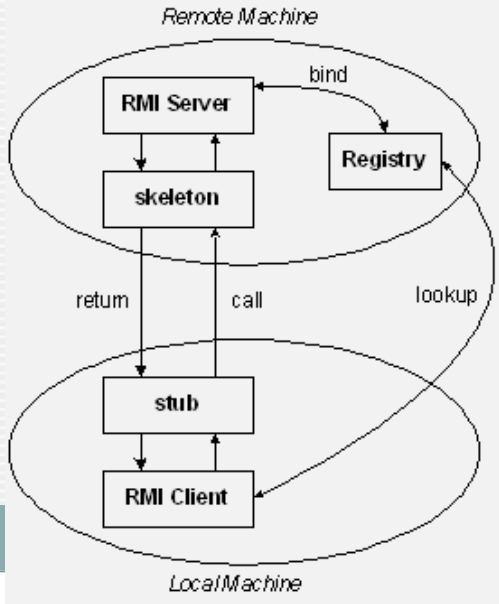
Misma

RMI – Esquema de una aplicación

- Interfaz remota (servidor + cliente):
  - Contiene los métodos remotos (sólo **declaración**) que pueden ser llamados del objeto remoto
  - Restricciones:
    - Debe **heredar** (*extends*) de la interfaz **java.rmi.Remote**
    - Todos los **métodos** deben **lanzar**, al menos, la excepción **java.rmi.RemoteException**
- Clase que implementa la interfaz remota (servidor):
  - Contiene el **código** de los métodos remotos declarados en la interfaz
  - Restricciones:
    - **Implementar** la **interfaz remota**
    - Declarar **constructor** (sin parámetros) que lanza **RemoteException**
    - **Extender** de la clase **UnicastRemoteObject**
      - Para que pueda recibir llamadas remotas, hay que **activar el objeto** de esta clase
  - Puede contener otros métodos adicionales (no remotos)

RMI – Esquema de una aplicación

- Clase principal del servidor:
  - En main, **creamos** el **objeto remoto**
  - Lo **hacemos visible** para clientes mediante **Naming.rebind(…)**
- Clase principal del cliente:
  - Definir la clase que permita **obtener los objetos remotos**
    - Buscándolos en el registro RMI de la máquina remota
      - Para ello usamos **Naming.lookup(…)**



Ejemplo RMI

- Interfaz común (cliente + servidor):

```
package saludaRmi;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface InterfaceSaluda extends Remote
{
     String saludar(String nombre) throws RemoteException; //Método que se publica
}
```

Ejemplo RMI

- Implementación interfaz (servidor):

```
package saludaRmi;

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

public class Saludador extends UnicastRemoteObject implements InterfaceSaluda
{
    public Saludador() throws RemoteException {}   //Constructor

    public String saludar(String nombre) throws RemoteException
    {
       // Implementación del método remoto
       return "Buenos días " + nombre;
    }
}
```

Ejemplo RMI

- Código principal del servidor:

```
package saludaRmi;

import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Servidor
{
    public static void main(String args[])
    {
        try {
            //Crea instancia del objeto que implementa la interfaz (objeto a registrar):
            Saludador obj = new Saludador();
            Registry registry = LocateRegistry.createRegistry(1099); //Arranca rmiregistry local en el puerto 1099
            Naming.rebind("//127.0.0.1/ObjetoSaluda", obj); //Hace visible el objeto para clientes
            System.out.println("El Objeto Saluda ha quedado registrado");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Ejemplo RMI

- Código principal del cliente:

```
package saludaRmi;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.rmi.Naming;

public class Cliente {
    public static void main(String args[]) {
        String respuesta = "";
        try {
            BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Introduce tu nombre:");
            String nombre = entrada.readLine();
            //Localiza el objeto distribuido:
            InterfaceSaluda obj = (InterfaceSaluda) Naming.lookup("//127.0.0.1/ObjetoSaluda");
            respuesta = obj.saludar(nombre);   //Llama al método saludar
            System.out.println(respuesta);
        } catch (Exception e) {
            System.out.println("Excepción : " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Ejercicios

- 1.- Ejecutar los códigos utilizados en esta presentación.
- 2.- Crear una aplicación distribuida con RMI, que permita realizar las siguientes operaciones:
  - Dados dos números, que devuelva como resultado la multiplicación de estos. Por ejemplo: multiplica(2,3)=6
  - Dados dos números, que devuelva como resultado la potencia del primero con respecto al segundo. Por ejemplo: potencia(2,3)=8
- 3.- Crear una aplicación distribuida con RMI, a la que enviemos nuestra fecha de nacimiento y nos devuelva como resultado la edad que tenemos.