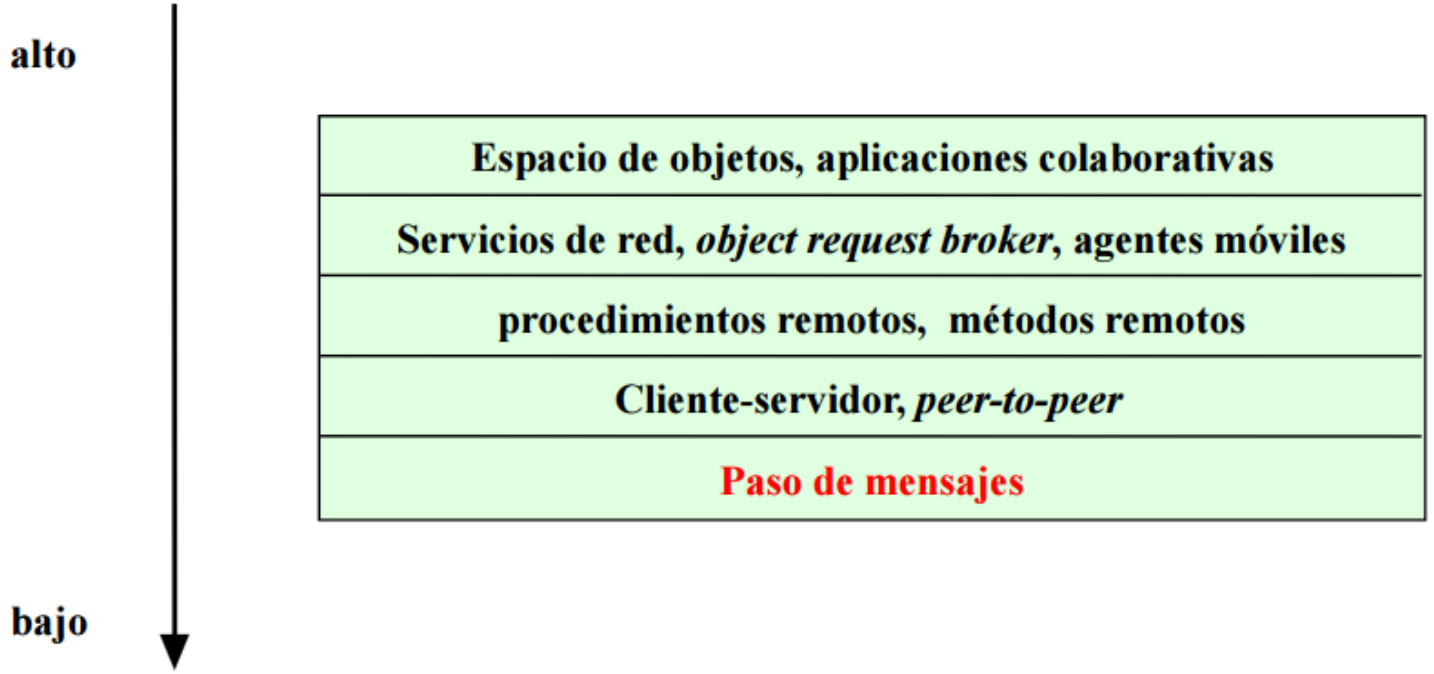
SISTEMAS DISTRIBUIDOS

PROGRAMA
DE INGENIERIA DE SISTEMAS

ING. DANIEL EDUARDO PAZ PERAFÁN

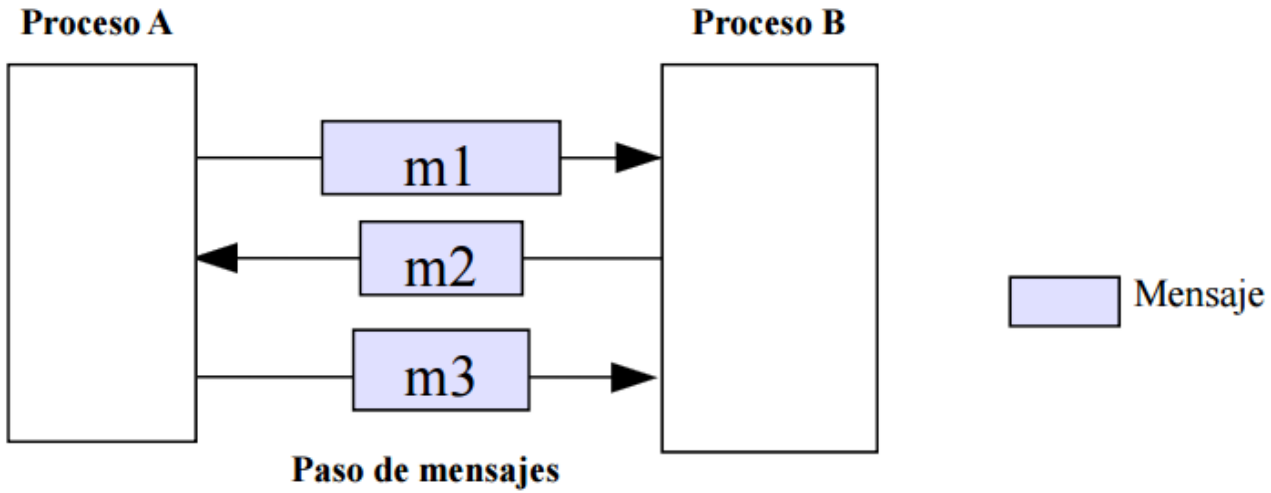
SISTEMAS DE OBJETOS DISTRIBUIDOS



Paradigma de paso de mensajes

Paradigma fundamental para aplicaciones distribuidas

- Un proceso envía un mensaje de solicitud
- El mensaje llega al receptor, el cual procesa la solicitud y devuelve un mensaje en respuesta
- Esta respuesta puede originar posteriores solicitudes por parte del emisor



Paradigma de paso de mensajes

- Las operaciones básicas para soportar el paradigma de paso de mensajes son enviar y recibir
  - Protocolos más comunes: IP y UDP
- Para las comunicaciones orientadas a conexión también se necesitan las operaciones conectar y desconectar
  - Protocolo más común: TCP
- Operaciones de Entrada/Salida que encapsulan el detalle de la comunicación a nivel del sistema operativo
  - Ejemplo: el API de sockets

Paradigma de llamadas a procedimientos remotos

Se basa en el paradigma procedimental.

Un problema es la abstracción de las entidades del modelo de un dominio y su modelado bajo el paradigma estructurado.

- Identifican entidades y sus características, se modelan como estructuras.
- Las acciones se modelan como funciones.

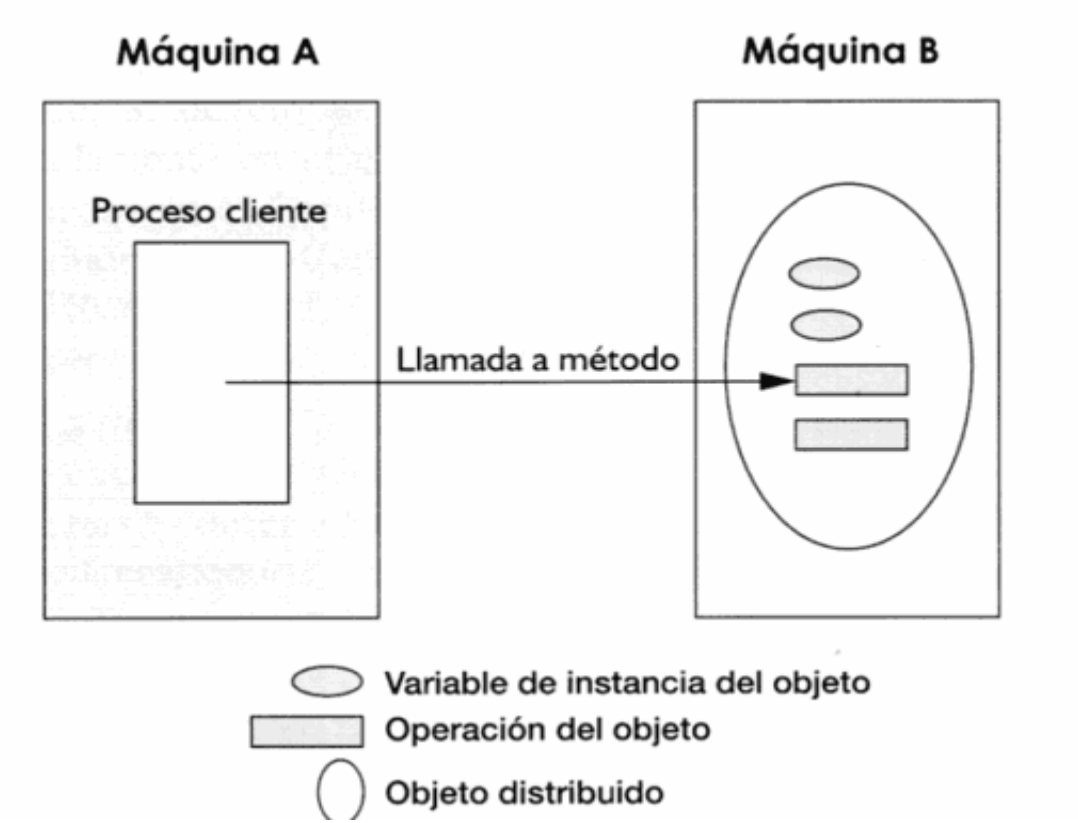
SISTEMAS DE OBJETOS DISTRIBUIDOS

- Ofrece una mayor abstracción.
- Cada objeto encapsula: estado, comportamiento.
- Orientado a acciones.

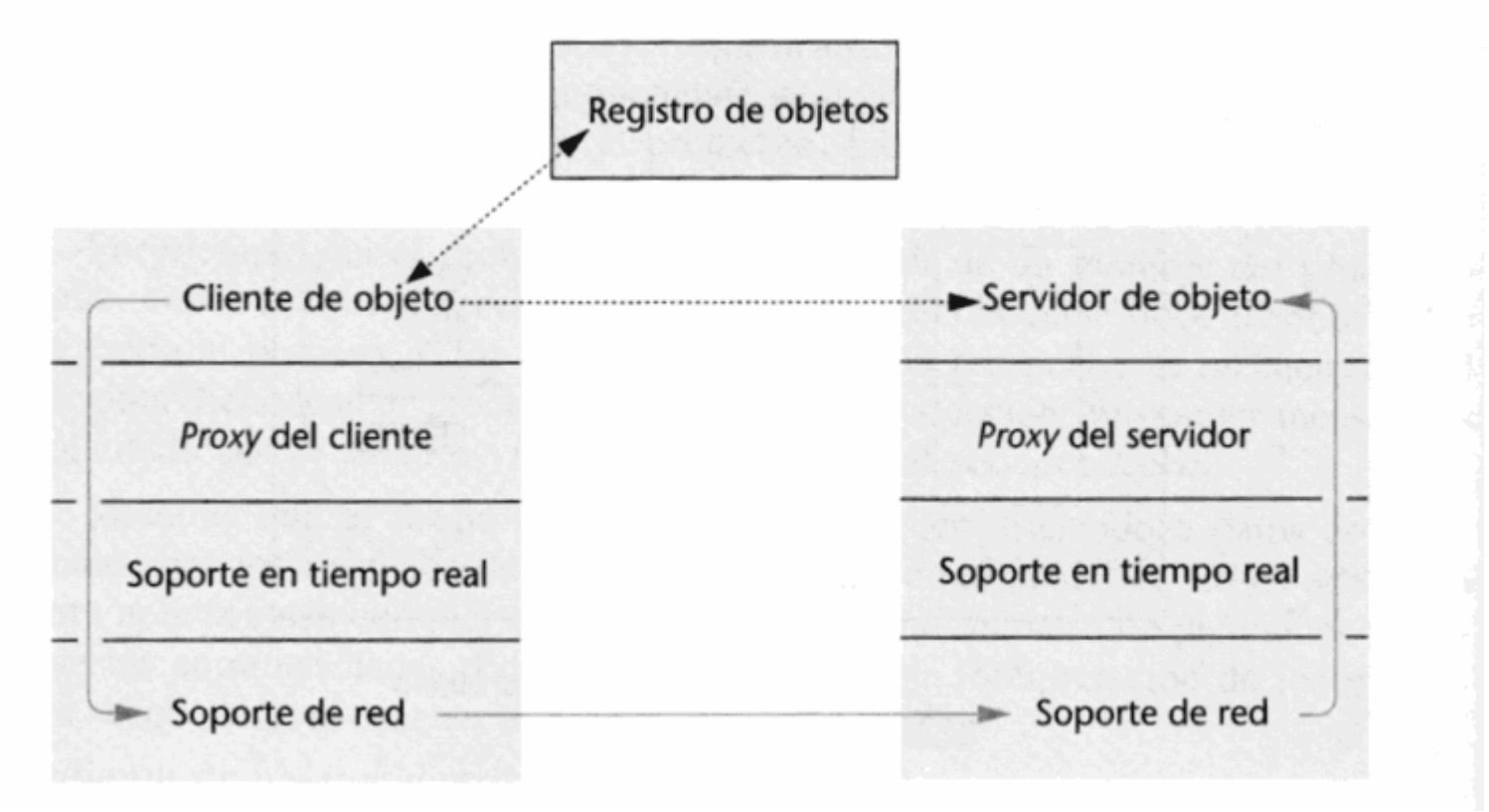
Tener en cuenta:

- Objeto Local(OL): métodos invocados por un O.L
- Objeto Distribuido(OD):métodos invocados por O.L u objetos remotos.
- Método remoto: Metodos de un OD.

SISTEMAS DE OBJETOS DISTRIBUIDOS



ARQUITECTURA BASICA DE OBJETOS DISTRIBUIDOS



APLICACIONES QUE HAN ADOPTADO SISTEMA DE OBJETOS DISTRIBUIDOS

-Java RMI(Remote Method Invocation).
-CORBA(Common Object Request Broker Architecture).
-DCOM(Distributed Component Object Model).
-SOAP(Simple Object Access Protocol).

CASO DE ESTUDIO: JAVA RMI

- Un objeto remoto u objeto distribuido es aquel cuyos métodos pueden ser invocados desde otra JVM, potencialmente ubicada en otro computador.
- Un objeto de este tipo es descrito por una o más interfaces remotas, las cuales son interfaces Java que declaran métodos de objetos remotos.
- La invocación de un método remoto (RMI) es la acción de invocar un método de una interface remota sobre un objeto remoto.

CASO DE ESTUDIO: JAVA RMI

INTRODUCCIÓN A JAVA

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principio de los años 90´s

Java desarrolla un código “neutro” que no depende del tipo de maquina, el cual se ejecuta sobre una “máquina virtual”, denominada Java Virtual Machine (JVM).

La JVM interpreta el código neutro (independiente del procesador) convirtiéndolo a código propio de la máquina concreta.

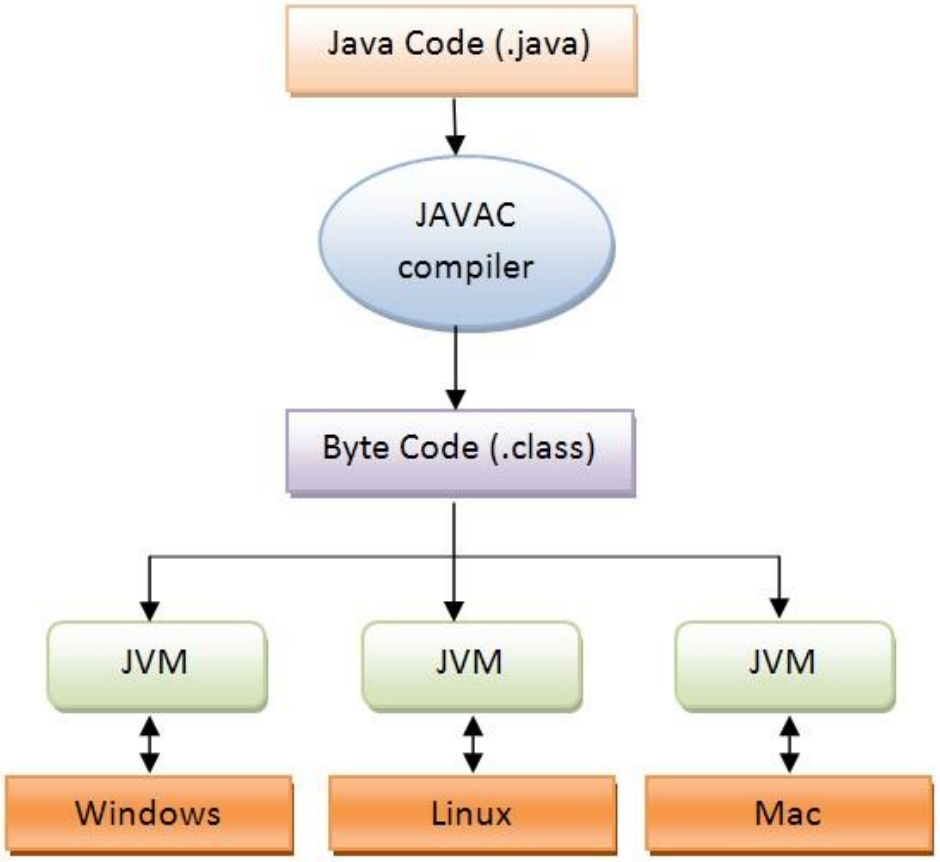
CASO DE ESTUDIO: JAVA RMI

Características de JAVA

- Paradigma netamente orientado a objetos, soportando las características de encapsulación, herencia y polimorfismo.
- Toma lo mejor de otros lenguajes orientados a objetos, como C y C++, sin embargo, el lenguaje tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, como la manipulación directa de punteros
- Tipado estáticamente. Todas las variables deben corresponder a un tipo de dato.
- Tiene un sistema automático de asignación y liberación de memoria (recolector de basura)
- El compilador Java traduce cada fichero fuente de clases a código de bytes (Bytecode), que puede ser interpretado por todas las máquinas

CASO DE ESTUDIO: JAVA RMI

MAQUINA VIRTUAL DE JAVA



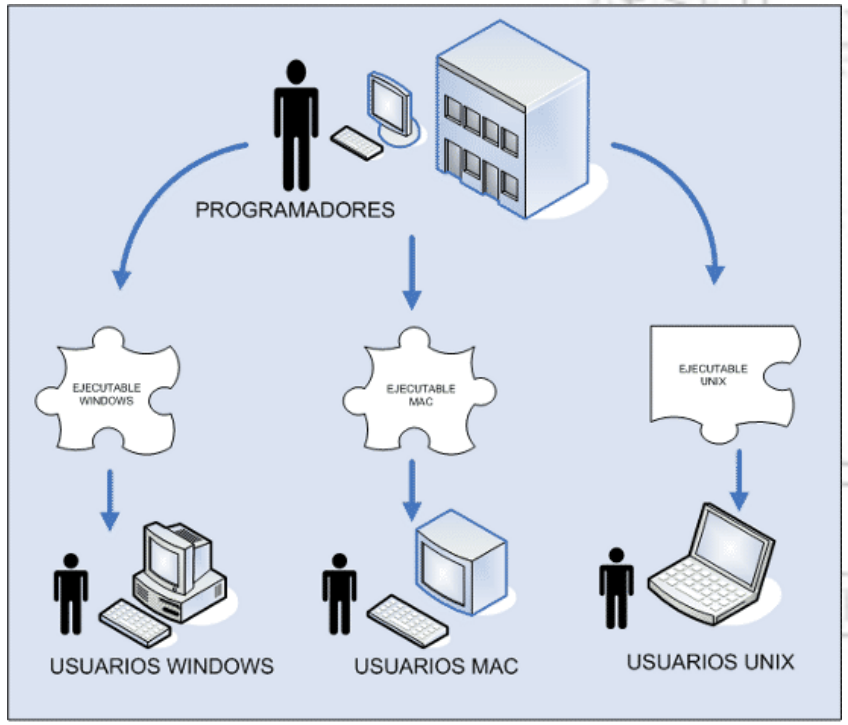
En el proceso de compilación se traduce el código fuente a byte code.

La JVM interpreta el código intermedio y lo traduce a lenguaje maquina, propio de la arquitectura HW de la computadora.

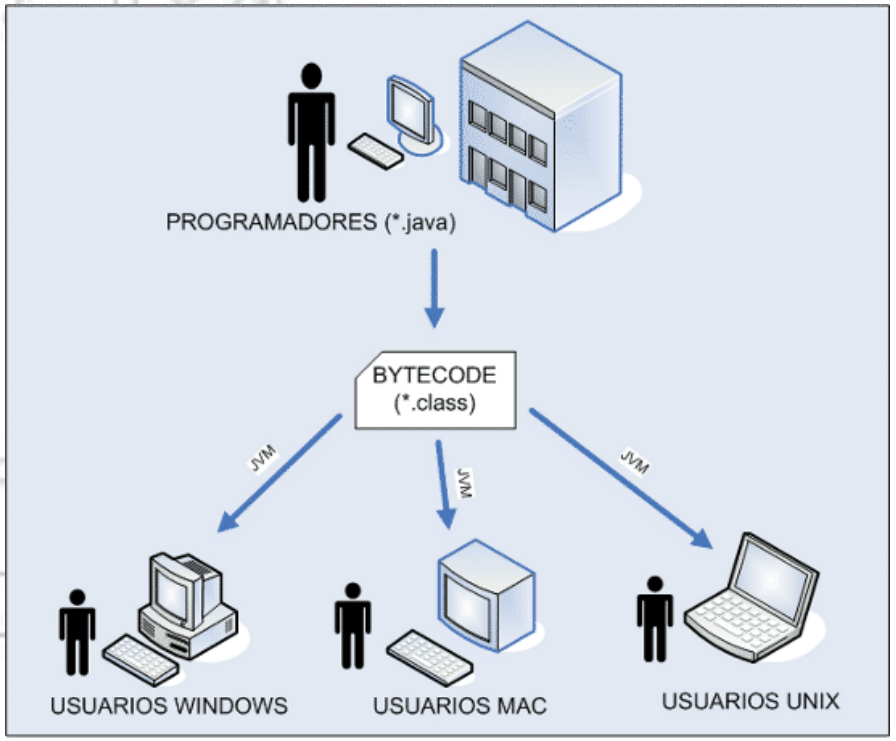
Java se hizo independiente del hardware y del sistema operativo en que se ejecutaba.

CASO DE ESTUDIO: JAVA RMI

MAQUINA VIRTUAL DE JAVA



ESQUEMA COMPILADO



ESQUEMA COMPILADO – INTERPRETADO JAVA

CASO DE ESTUDIO: JAVA RMI

JRE (Entorno de ejecución de Java) y JDK (Java Development Kit).

El JRE es la implementación de la Máquina virtual de Java que realmente ejecuta los programas de Java.

El JDK es un kit de desarrollo de software que se utiliza para escribir aplicaciones con el lenguaje de programación Java.

Los archivos respectivos que se encargan de estas tareas son:

El compilador Java --- > javac.exe. Se encarga de compilar el código fuente y pasarlo a bytecode.

El intérprete Java --- > java.exe. Se encarga de interpretar los archivos .class (bytecode) a código maquina.

CASO DE ESTUDIO: JAVA RMI

- La meta principal de los diseñadores de RMI fue la de 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.

CASO DE ESTUDIO: JAVA RMI

- Un objeto remoto u **objeto distribuido** es aquel cuyos métodos pueden ser invocados desde otra JVM, potencialmente ubicada en otro computador.
- Un objeto de este tipo es descrito por una o más interfaces remotas, las cuales son interfaces Java que declaran métodos de objetos remotos.
- La invocación de un método remoto (RMI) es la acción de invocar un método de una interface remota sobre un objeto remoto.

CASO DE ESTUDIO: JAVA RMI

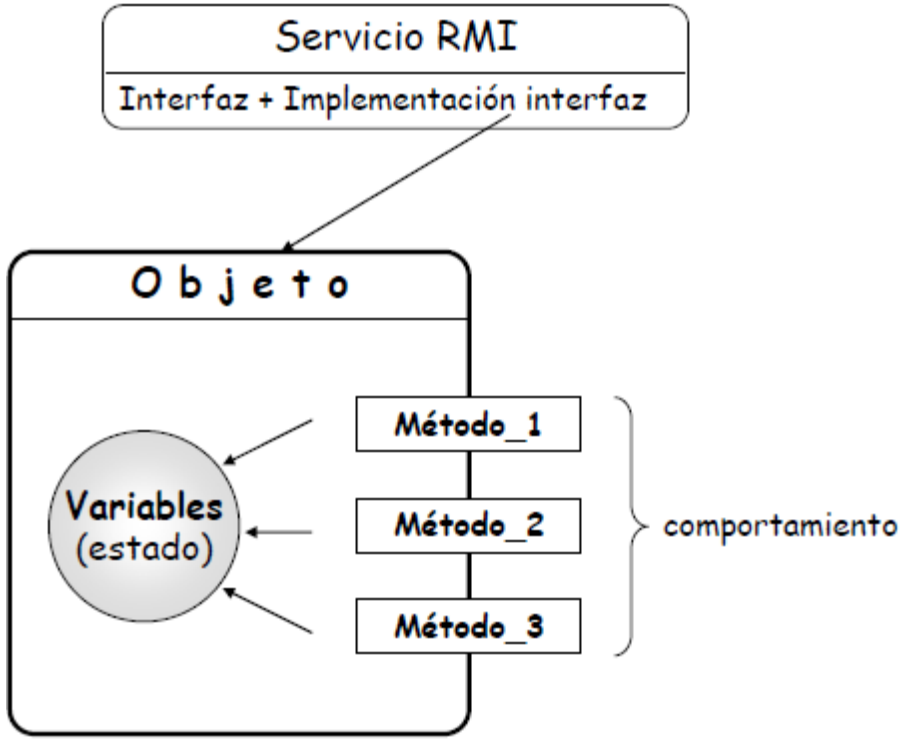
INTERFACE

Se definen un conjunto de operaciones remotas

CLASE

Implementan las operaciones de la interface

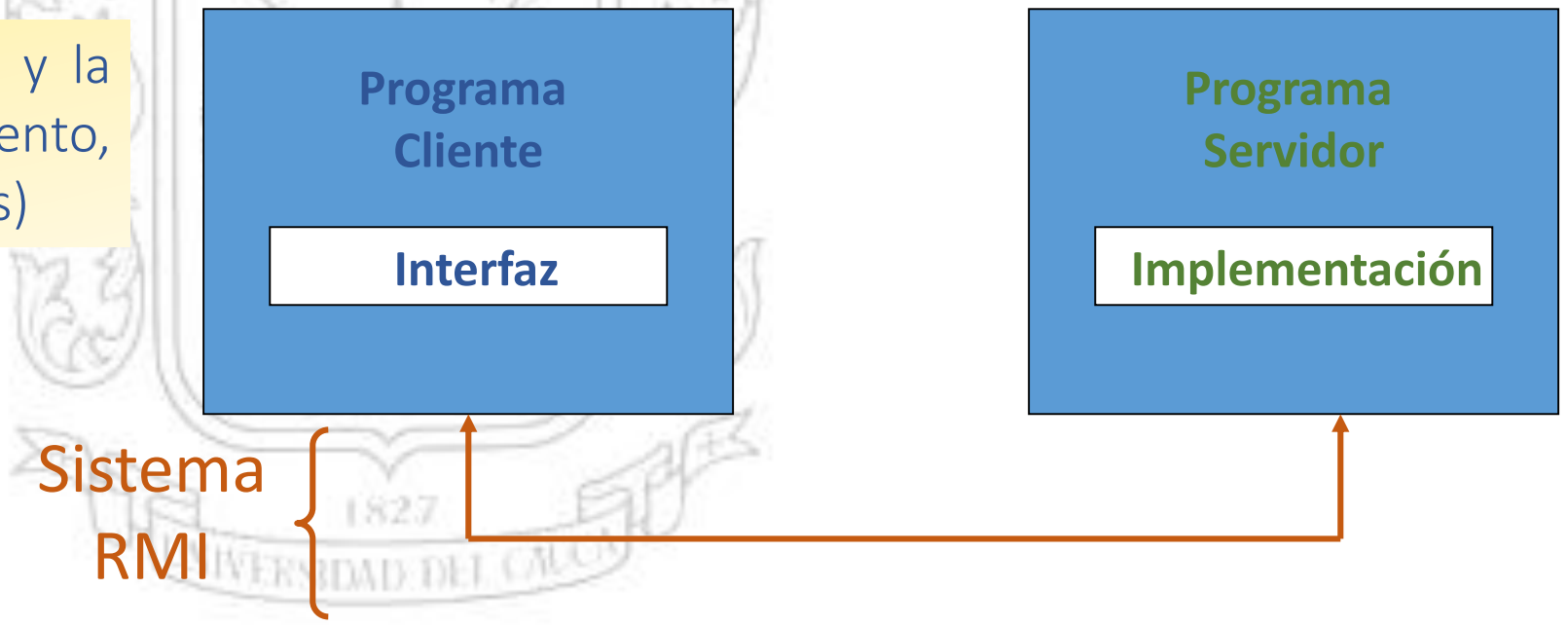
CASO DE ESTUDIO: JAVA RMI



- Un objeto tiene un identificador único dentro de su MV

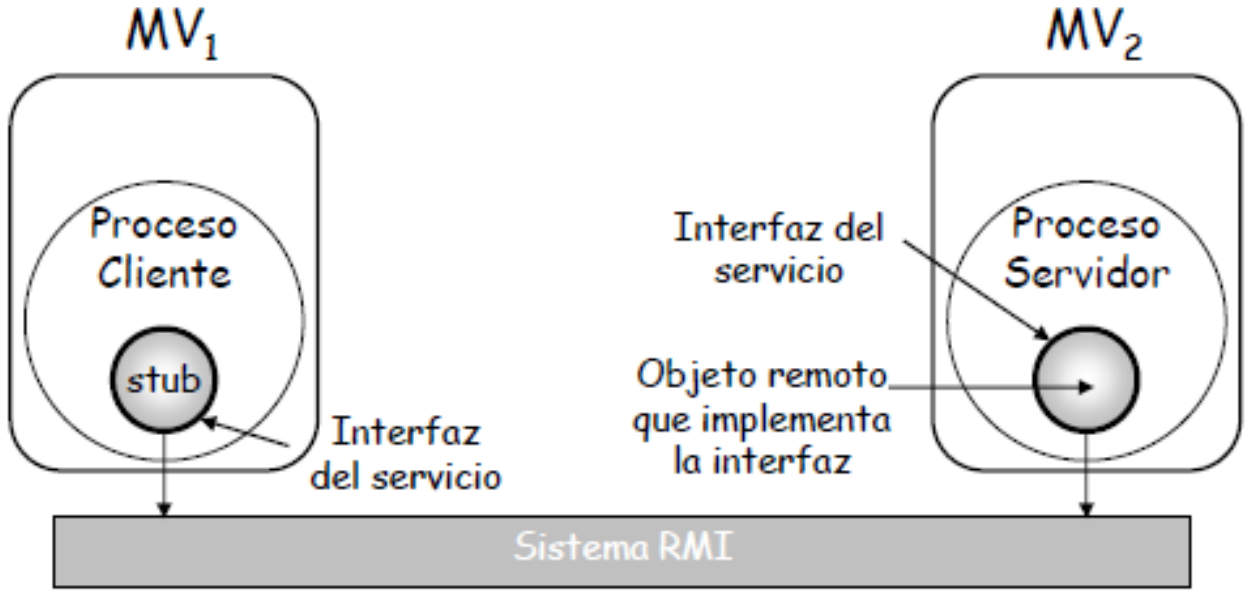
CARACTERÍSTICAS DE JAVA RMI

La definición del comportamiento y la implementación del comportamiento, están separadas. (interfaces - clases)



CASO DE ESTUDIO: JAVA RMI

interface remota.

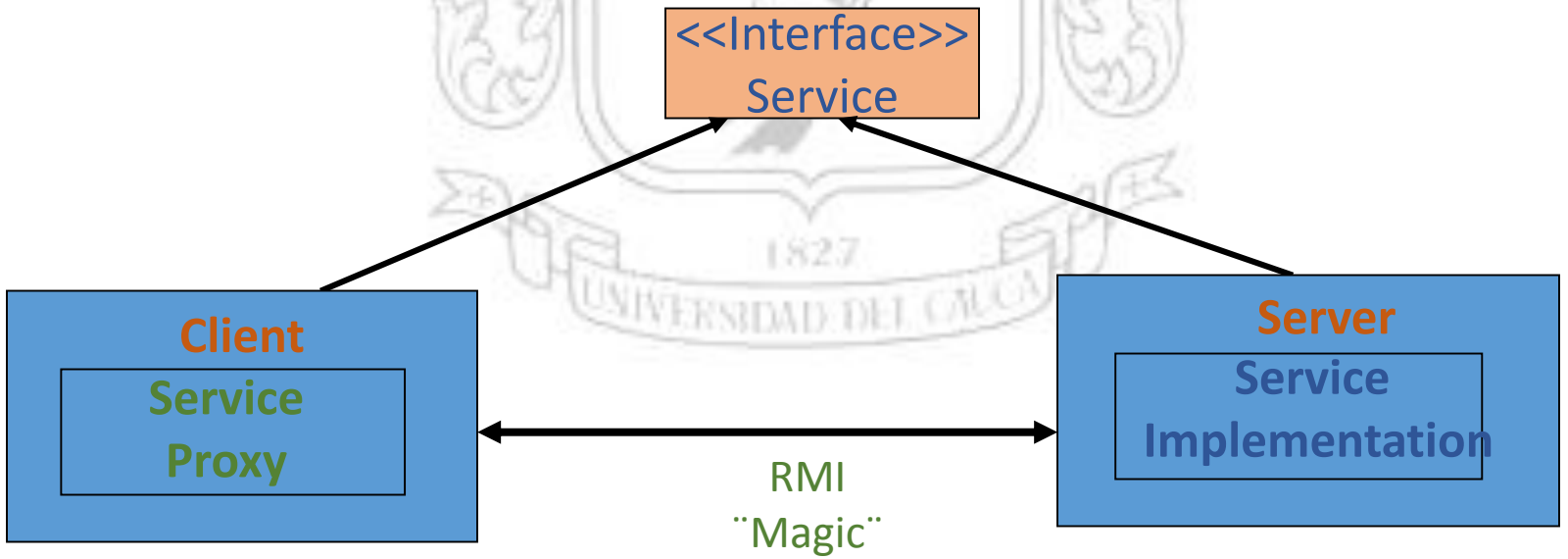


Un objeto es descrito por una o mas interfaces remotas

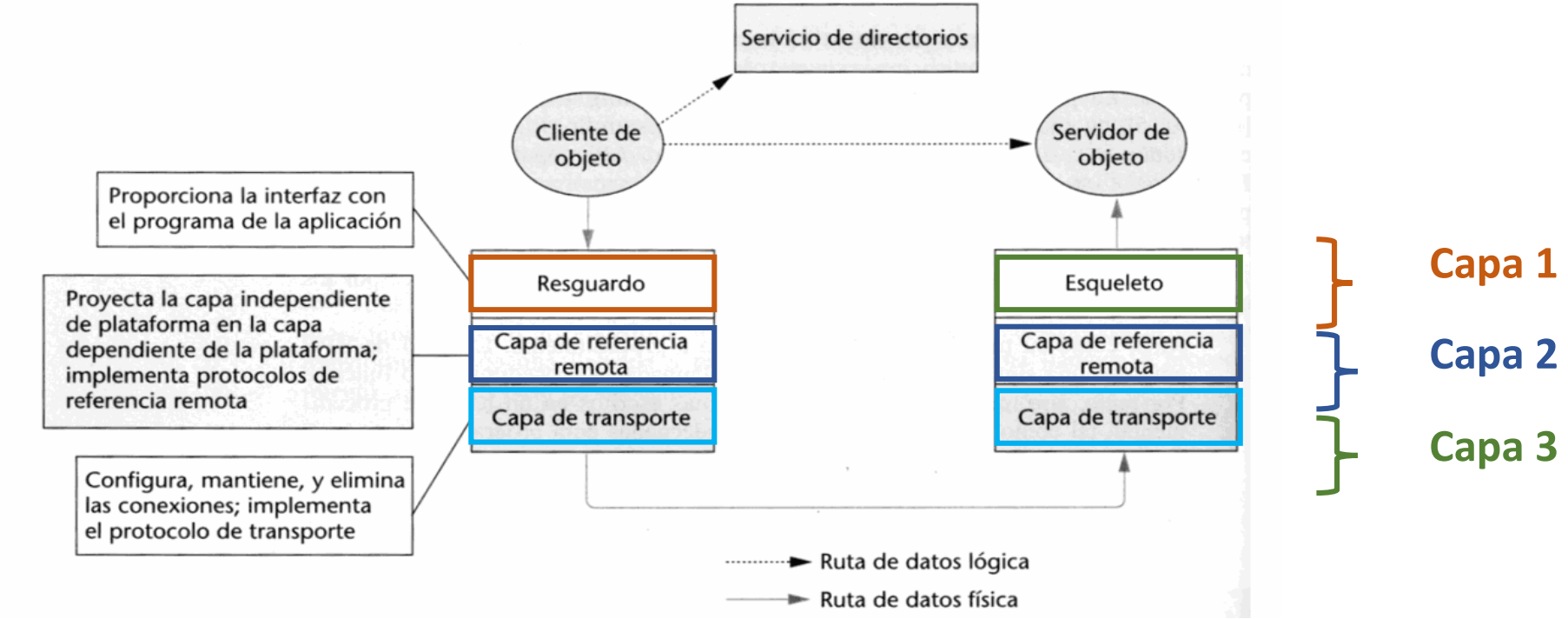
La invocación no se realiza sobre el método del objeto remoto sino sobre su interface remota.

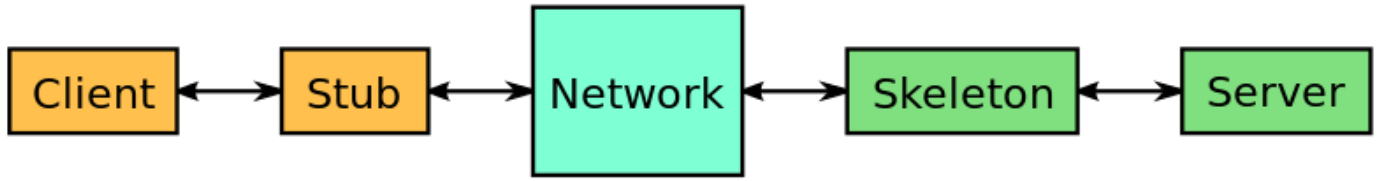
CASO DE ESTUDIO: JAVA RMI

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.



ARQUITECTURA DE JAVA RMI

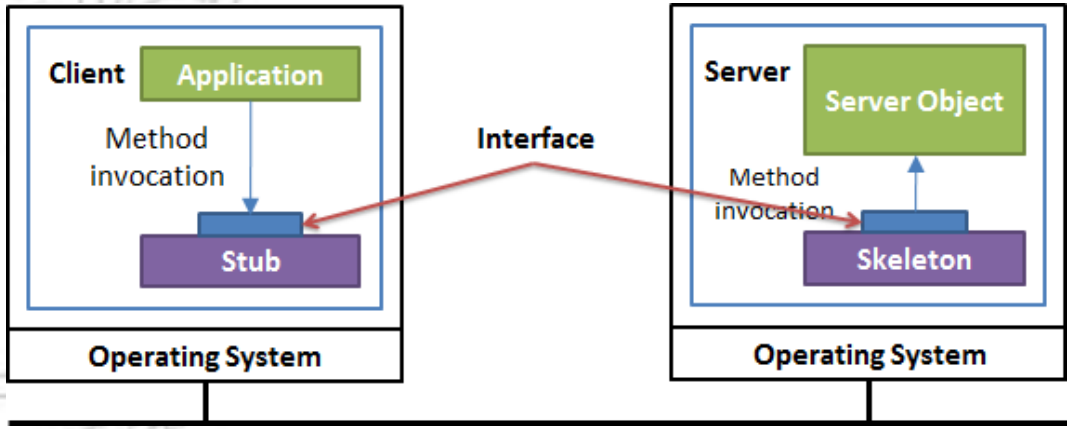




CARACTERÍSTICAS DE JAVA RMI

Capa stubs & skeleton

- Su objetivo conseguir la transparencia de localización.
- Se basa en el patrón de diseño Proxy
- El skeleton es obsoleto en la versión Java 2SDK



CARACTERÍSTICAS DE JAVA RMI

Stub

- Inicia una conexión con la JVM remota que almacena el objeto remoto.
- Realiza el aplanado(serialización) de los parámetros para la JVM remota
- Espera por el resultado de la invocación del método.
- Realiza el desaplanado (deserialización) al valor del resultado o la excepción en caso de error.
- Retorna el valor al llamador

CARACTERÍSTICAS DE JAVA RMI

Skeleton

- Desaplana los parámetros para el método remoto
- Invoca el método sobre la actual implementación del objeto remoto.
- Realiza el Aplanado del valor del resultado para el llamador
- El JDK contiene la herramienta **rmic** que **genera** el stub y skeleton

CARACTERÍSTICAS DE JAVA RMI

Nivel de los Stubs y Skeletons

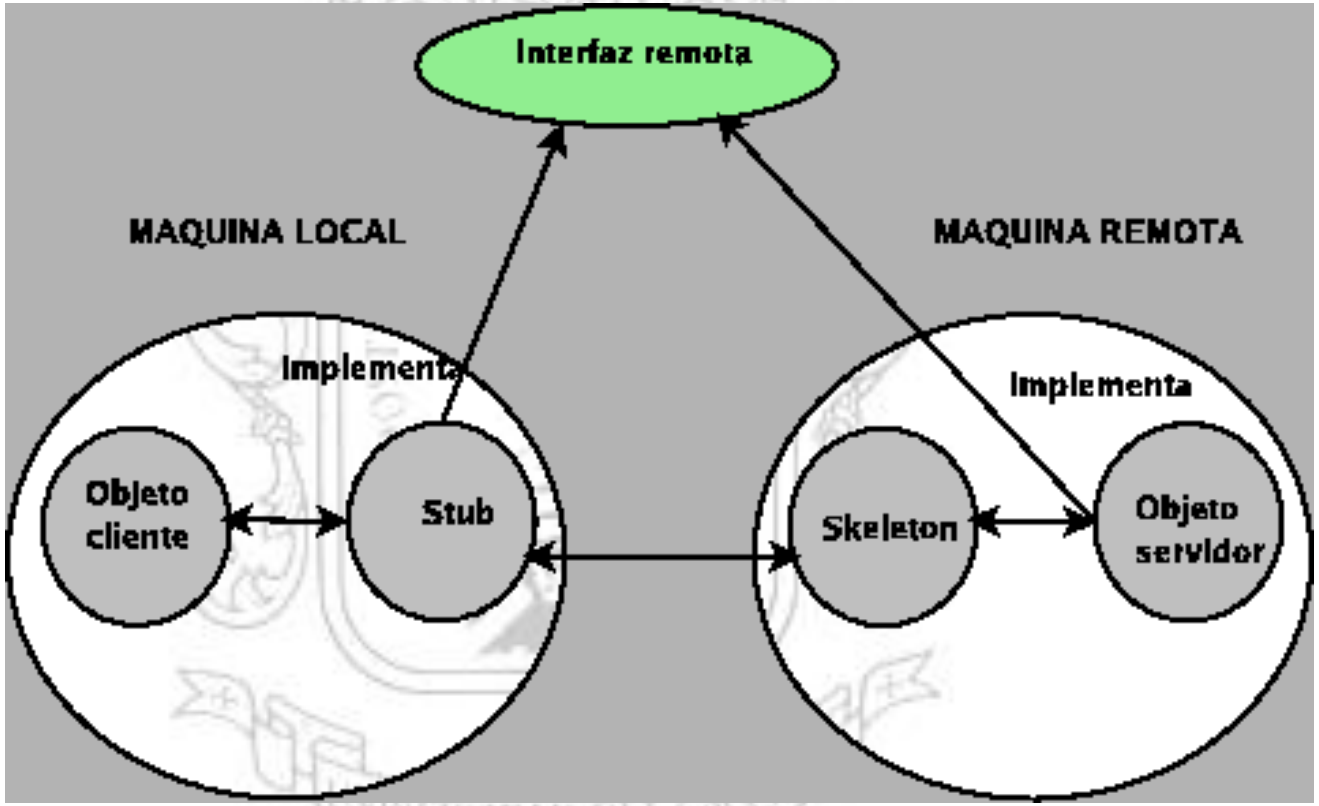
- En este nivel RMI usa el patrón de diseño de Proxy.
- En este patrón un objeto en un contexto es representado por otro (el proxy) en un contexto diferente.
- El proxy sabe cómo dirigir las invocaciones a métodos entre los distintos objetos participantes.

CARACTERÍSTICAS DE JAVA RMI

Capa de referencia remota

- Define y soporta la semántica de la invocación de una conexión RMI.
- 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.

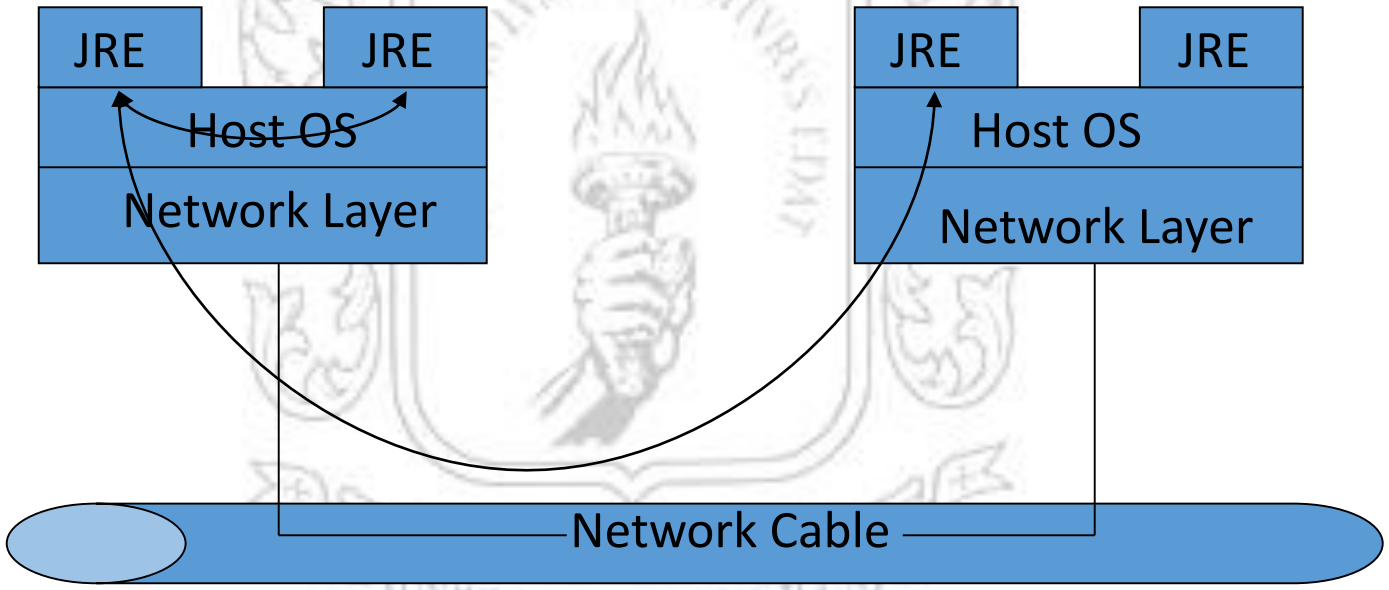
CARACTERÍSTICAS DE JAVA RMI



Relaciones entre elementos RMI

CARACTERÍSTICAS DE JAVA RMI

Capa de Transporte



Se basa en conexiones TCP/IP

CARACTERÍSTICAS DE JAVA RMI

Capa de Transporte

- Permite el establecimiento y control de un canal de comunicación virtual orientado a la transmisión de flujo de bytes entre dos o más procesos.
- Por encima de TCP/IP, RMI usa un wire level protocol propiedad de Sun, conocido como *Java Remote Method Protocol* (JRMP).
- El inconveniente de que el protocolo JRMP sea propiedad de Sun es que no es abierto, por lo que tiene ciertas dificultades para comunicarse con otros *middleware* como, por ejemplo, CORBA.

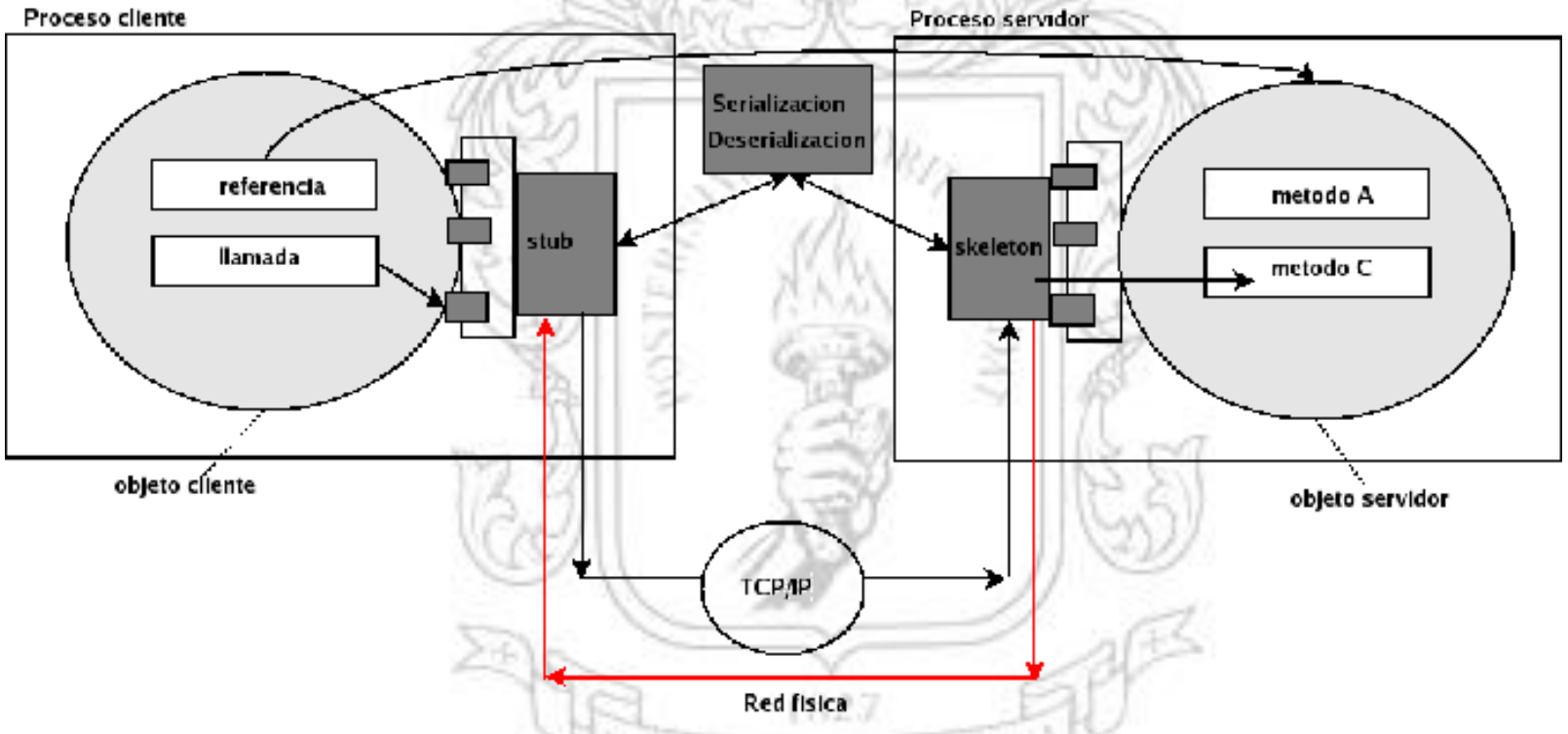
CARACTERÍSTICAS DE JAVA RMI

Capa de Transporte

- Sun ofrece la posibilidad de usar un protocolo abierto como el IIOP (Internet InterORB Protocol) en lugar de JRMP.

- RMI sobre IIOP permite a objetos remotos de java permite comunicarse con objetos CORBA que pueden estar escritos en un lenguaje de programación diferente a java.

CARACTERÍSTICAS DE JAVA RMI



Comunicación basada en RMI

Reflection en JAVA RMI

Deseamos escribir un programa que en tiempo de ejecución:

- Determine de qué tipo es un atributo de una clase: int, String, float, boolean
- Indicar si una variable de una clase es public, private, final o static
- Cuántos parámetros le llegan a una función de una clase.
- Permita utilizar un método privado de un objeto de una clase.
- Acceder directamente a una variable privada de otra clase?

<Reflection API>
Java

Reflection en JAVA RMI

La reflexión es comúnmente utilizada por programas que requieren la capacidad de examinar o modificar el comportamiento en tiempo de ejecución de aplicaciones que se ejecutan en la máquina virtual de Java

Mediante la Java Reflection API un programador puede inspeccionar y manipular clases, sus métodos y atributos en tiempo de ejecución, sin conocer a priori (en tiempo de compilación) los nombres de las clases específicas con las que está trabajando, la definición de atributos o implementación de sus métodos.

Reflection en JAVA RMI

La reflexión es comúnmente utilizada por programas que requieren la capacidad de examinar o modificar el comportamiento en tiempo de ejecución de aplicaciones que se ejecutan en la máquina virtual de Java

Mediante la Java Reflection API un programador puede inspeccionar y manipular clases, sus métodos y atributos en tiempo de ejecución, sin conocer a priori (en tiempo de compilación) los nombres de las clases específicas con las que está trabajando, la definición de atributos o implementación de sus métodos.

Reflection en JAVA RMI

Investigación:

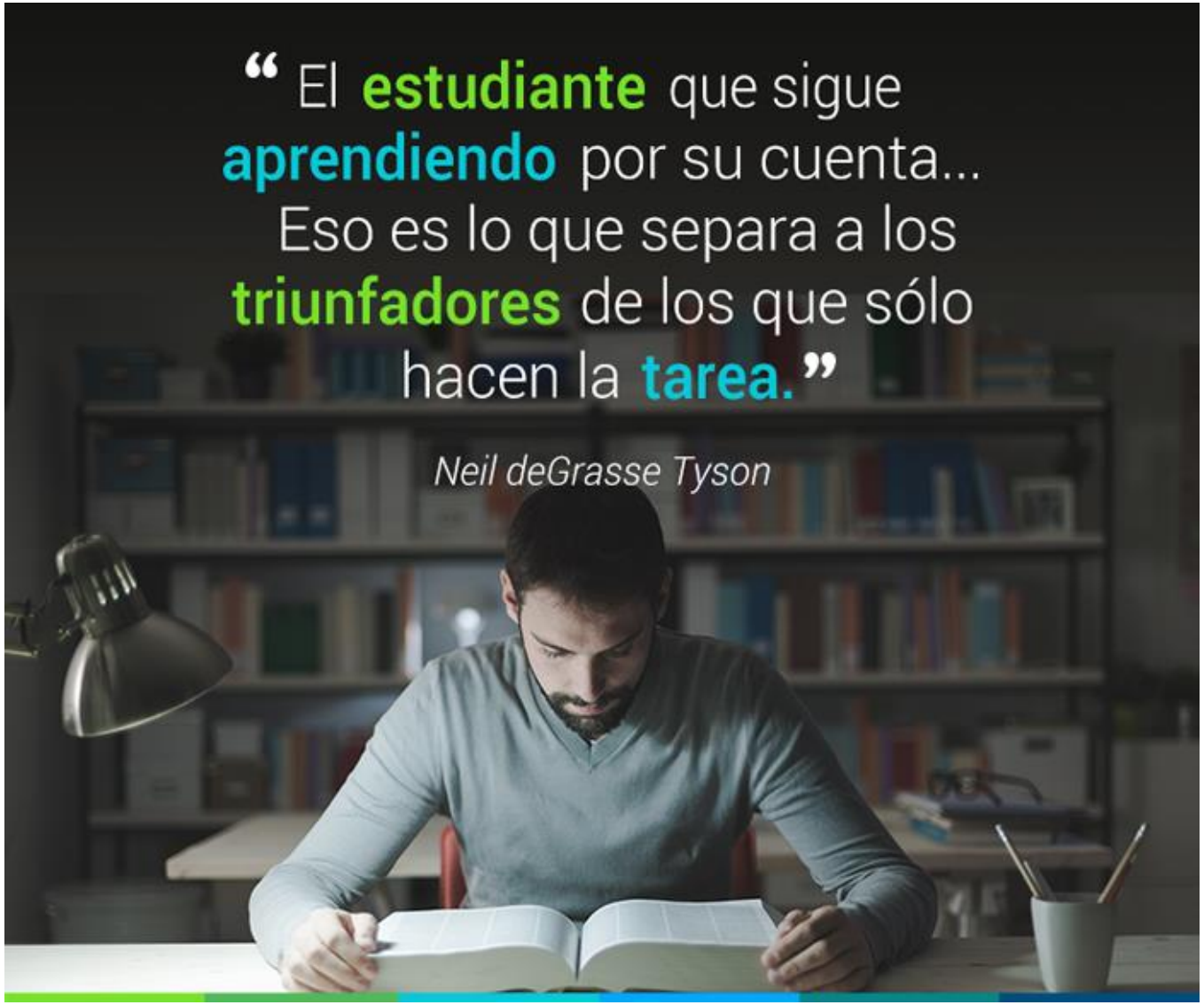
Crear una clase que tenga atributos públicos, privados y de diferente tipo.

La clase debe manejar un constructor por defecto y uno parametrizado.

La clase debe tener varios métodos públicos y privados implementados que reciban varios parámetros, realicen operaciones y retornen un valor.

Crear un ejemplo que permita en tiempo de ejecución, utilizando únicamente el api de Java Reflection :

a) Obtener el tipo de dato de los atributos, sus nombres, valores y el modificador de acceso (public, private) de la clase declarada.
b) Obtener todos los métodos de la clase y mostrar: modificador de acceso, nombre del método, cantidad de parámetros, tipo de dato de cada parámetro y tipo de dato de retorno del método.
c) Acceder a los atributos privados y obtener el tipo de dato, sus nombres y valores
d) Cambiar el valor de un atributo privado, sin acceder a un método publico.
e) Invocar un método privado.
f) Crear una instancia de la clase y utilizar la instancia.


“ El estudiante que sigue aprendiendo por su cuenta... Eso es lo que separa a los triunfadores de los que sólo hacen la tarea.”
Neil deGrasse Tyson

Registro de objetos remotos en Java RMI

Localización de objetos remotos

Como un cliente encuentra un servicio remoto RMI?

Java permite el uso de diferentes servicios de nombrados para registrar un objeto distribuido.

- **Interfaz de nombrado y directorios de java (JNDI) Java Naming and Directory Interface**
- **Java.rmi.registry.Registry. Proporciona el servicio de nombrado rmiregistry que es un servicio de directorios. Utiliza el puerto tcp por defecto 1099.**

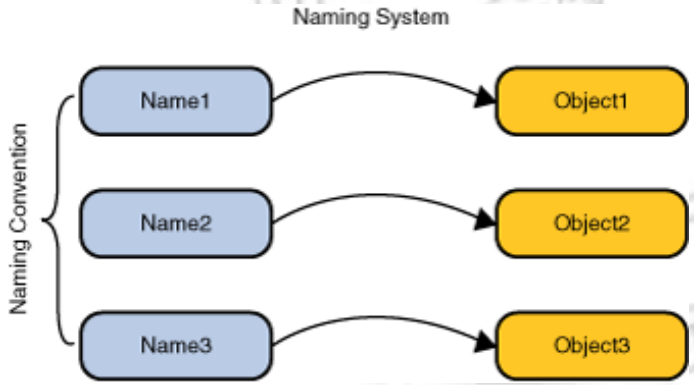
Deben estar ubicados en un host o un puerto bien conocido

Registro de objetos remotos en Java RMI

Servicio de nombrado

Una facilidad fundamental en cualquier Sistema de computo es el servicio de nombrado, el cual significa que a un objeto se le asocia un nombre.

Un servicio de nombrado permite buscar un objeto a partir de su nombre



Identificador **Objeto**

El Sistema de nombrado determina la sintaxis que el nombre del identificador debe seguir.

Registro de objetos remotos en Java RMI

Lanzar el N_S

El N_S de RMI es un objeto remoto que sirve como un servicio de directorios para los clientes manteniendo en un mapa los nombres de otros objetos remotos.

El comportamiento se define en la interfaz:

Java.rmi.registry.Registry

JDK implementa la interfaz con el nombre Registry

Registro de objetos remotos en Java RMI

Lanzar el N_S

JDK proporciona una herramienta para la ejecución del rmiregistry

El N_S **rmiregistry** puede ser lanzado usando programación:

a) LocateRegistry.getRegistry(int puerto): retorna un stub si el N_S ya esta ejecutandose.

b) LocateRegistry.createRegistry(int puerto): crea un objeto remoto n_s y retorna una referencia para su acceso

El N_S rmiregistry puede ser lanzado desde consola utilizando el comando:

rmiregistry [port]

Corre un objeto remoto de registro, en un puerto especifico del host actual.

Ejemplo Registro de objetos remotos en Java RMI

Método que permite arrancar un servidor de registro RMI. Primero intenta obtener una referencia a un objeto remoto Registry. Si se lanza una excepción en la cual se establece que no existe una instancia del objeto remoto Registry se crea una nueva instancia.

```
public static void arrancarNS(int numPuertoRMI) throws RemoteException
{
        try
        {
                Registry registro = LocateRegistry.getRegistry(numPuertoRMI);

        }
        catch(RemoteException e)
        {
                System.out.println("El registro RMI no se localizó en el puerto" + numPuertoRMI);

                Registry registro = LocateRegistry.createRegistry(numPuertoRMI);
                System.out.println("El registro se ha creado en el puerto: " + numPuertoRMI);
        }

}
```

Registro de objetos remotos en Java RMI

La clase Naming proporciona métodos para obtener y almacenar referencias de los objetos remotos mediante URLs, en el servicio de nombrado rmiregistry. Sus métodos más habituales son:

- public static void **bind**(String name, Remote object) throws AlreadyBoundException,MalformedURLException,RemoteException
- public static void **rebind**(String name, Remote object) throws RemoteException, MalformedURLException
- public static void **lookup**(String name) throws NotBoundException,MalformedURLException,RemoteException

Registro de objetos remotos en Java RMI

Registro del objeto servidor en el n_s

El servidor de objetos registra el objeto servidor en el servicio RMI Registry por medio de los métodos de la clase estática **java.rmi.Naming**

- public static void **bind**(String name, Remote object) throws AlreadyBoundException,MalformedURLException,RemoteException
- public static void **rebind**(String name, Remote object) throws RemoteException, MalformedURLException

Aceptan la URL de la forma: rmi://<host_name>[:<service_port>]/<service_name>

Ejemplo Registro de objetos remotos en Java RMI

El servicio de nombres se encarga de registrar la referencia al objeto remoto.

El método rebind permite almacenar una referencia al objeto remoto en el N_S.

```
public static void RegistrarObjetoRemoto(Remote objetoRemoto, String dirIP, int numPuerto, String nombreObjeto)
{
        String UrlRegistro = "rmi://"+dirIP+":"+numPuerto+"/"+nombreObjeto;
        try
        {
                Naming.rebind(UrlRegistro, objetoRemoto);
                System.out.println("Se realizó el registro con la direccion: " +UrlRegistro);
        } catch (RemoteException e)
        {
                System.out.println("Error en el registro: " + e.getMessage());
        } catch (MalformedURLException e)
        {
                System.out.println("Error url inválida: " + e.getMessage());
        }

}
```

Consulta de objetos remotos en Java RMI

Consulta de objeto servidor en el cliente.

El cliente accede al servicio RMI Registry por medio de los métodos de la clase estática java.rmi.Naming, mediante el método:

- public static Remote **lookup**(String name) throws NotBoundException,MalformedURLException,RemoteException

Aceptan la URL de la forma: **rmi://<host_name>[:<service_port>]/<service_name>**

Ejemplo de consulta de objetos remotos en Java RMI

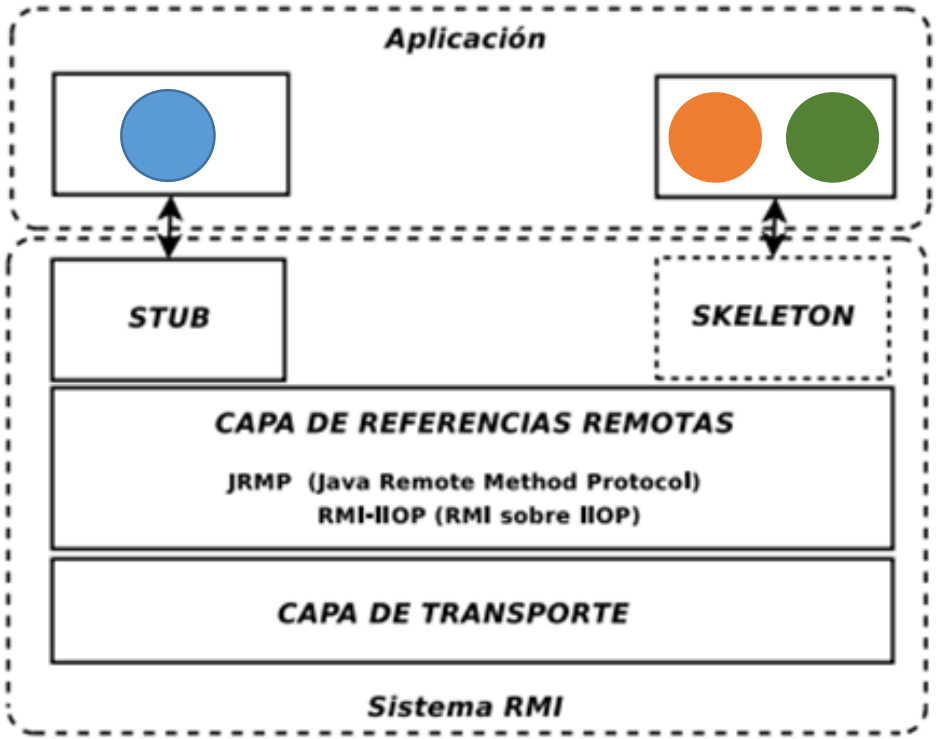
El servicio de nombres se encarga de registrar, almacenar y distribuir esas referencias remotas

Clientes reciben una copia serializada del stub al hacer lookup() en el servicio de nombres

```
public static Remote obtenerObjRemoto(int puerto, String dirIP, String nameObjReg)
{
    String URLRegistro;
    URLRegistro  = "rmi://" + dirIP + ":" + puerto + "/"+nameObjReg;
    try
    {
        return Naming.lookup(URLRegistro);
    }
    catch (Exception e)
    {
        System.out.println("Excepcion en obtencion del objeto remoto"+ e);
        return null;
    }
}
```

Registro de objetos remotos en Java RMI

CO: Cliente de objetos



SO: Servidor de objetos

OS: Objeto remoto

CO: Cliente de objetos (Aplicación que interactúa con el usuario)

SO: Servidor de objetos (Aplicación que instancia un OS y lo registra en el NS)

OS: Objeto servidor (Objeto que implementa un interfaz remota)

Registro de objetos remotos en Java RMI

Servidor de objetos

a) En un host, un servidor crea un servicio remoto, construyendo primero un objeto local que implementa el servicio (interface).

b) Luego, se exporta el objeto a RMI. Cuando el objeto es exportado, RMI crea un servicio que espera que los clientes se conecten para solicitarlo (listening service).

c) Después de exportar el objeto, el servidor lo registra en el RMI registry con un nombre público.

Consulta objetos remotos en Java RMI

Cliente de objetos

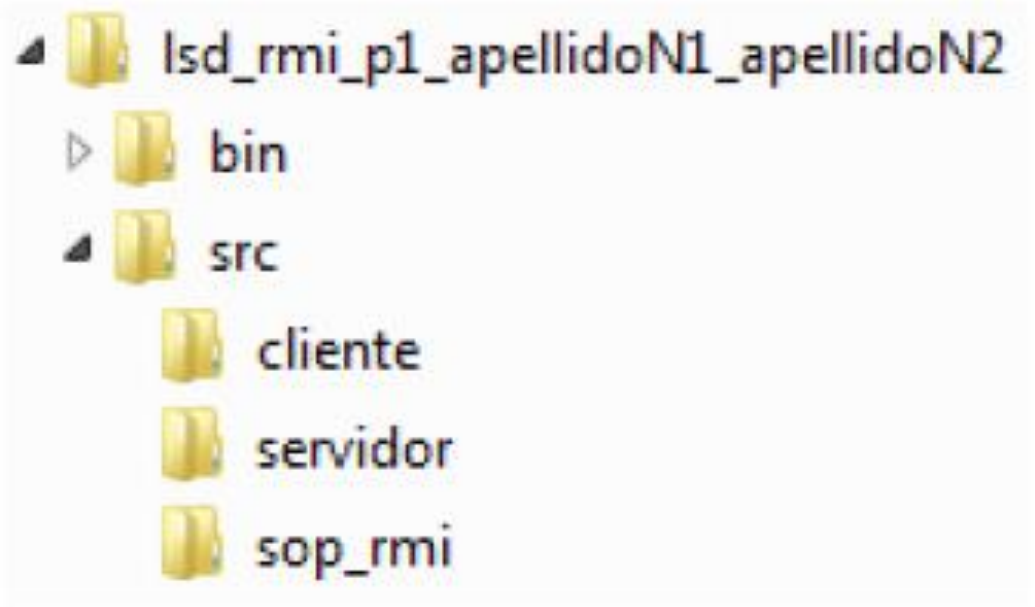
1. Establecer el gestor de seguridad SecurityManager (opcional) Necesario si se van a utilizar objetos remotos residentes en una máquina física distinta Establecerlo antes de cualquier invocación RMI (incluida llamada a RMIregistry) System.setSecurityManager(newSecurityManager());

2. Obtener las referencias al objeto remoto (stubs) consultando el servicio de nombres (rmiregistry) (método lookup(nombre))

3. Invocar los métodos remotos llamando a los métodos del stub con los parámetros precisos

Desarrollar una aplicación en Java RMI

Antes de iniciar, estructurar una carpeta de trabajo donde se ubicaran los archivos fuente (directorio 'src') y los archivos binarios (bytecode) (directorio bin)



Pasos para desarrollar la aplicación

Pasos durante el desarrollo

a) Definir una interface remota.
b) Implementar la interface remota
c) Escribir el servidor de objetos (Crea el objeto local, se pone en escucha y se registra en el n_s)
d) Escribir el cliente de objetos que utilizara el objeto remoto (Busca la referencia en el n_s)

Pasos durante la ejecución

Generar los stub y skeleton (utilizar rmic únicamente si la aplicación va a ejecutarse sobre una versión anterior a java 1.2)

a) Lanzar el rmiregistry (servidor de nombres)
b) Ejecutar el servidor de objetos (Crea el objeto local, se pone en escucha y se registra en el n_s)
c) Ejecutar el cliente de objetos

Ejemplo aplicación Java RMI

Primer paso: Definir la interface remota

```
package sop_rmi;

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

//Hereda de la clase Remote, lo cual la convierte en interfaz remota
public interface GestorUsuariosInt extends Remote
{
    //cabecera del primer método remoto
    public boolean registrarUsuario(int identificacion, String nombres, String apellidos) throws RemoteException;
    //cada definición del método debe especificar que puede lanzar la excepción java.rmi.RemoteException

    //cabecera del segundo método remoto
    public int consultarCantidadUsuarios() throws RemoteException;
    //cada definición del método debe especificar que puede lanzar la excepción java.rmi.RemoteException
}
```

Ejemplo aplicación Java RMI

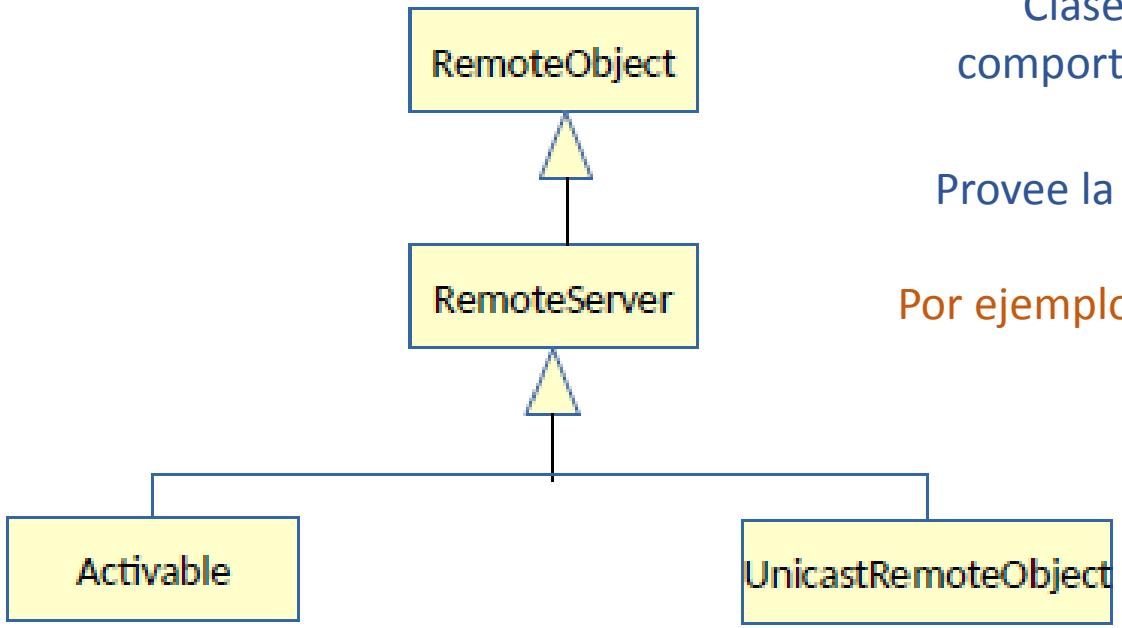
Segundo paso: Implementar la interface remota

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;

/*
Clase que implementa la interface remota GestorUsuariosInt
*/

public class GestorUsuariosImpl extends UnicastRemoteObject implements GestorUsuariosInt
{
    ArrayList<Usuario> misUsuarios;

    public GestorUsuariosImpl() throws RemoteException
    {...4 lines }

    @Override
    public int consultarCantidadUsuarios() throws RemoteException
    {...4 lines }

    @Override
    public boolean registrarUsuario(int identificacion, String nombres, String apellidos) throws RemoteException
    {...13 lines }

}
```

Una implementación de la interface no tiene porque extender de **UnicastRemoteObject**

Por tanto necesitamos exportar explícitamente el objeto remoto llamando a uno de los métodos **UnicastRemoteObject.exportObject**

Ejemplo aplicación Java RMI



Clase abstracta que describe el comportamiento de un objeto remoto

Provee la semántica básica para soportar la referencia remota
Por ejemplo: Crear y exportar un Obj remoto a un particular puerto

La clase explícitamente permite que un objeto se exporte a si mismo, utilizando el método exportObjetc().
Exportar es la habilidad de un objeto remoto de aceptar peticiones.
Involucra escuchar en un socket TCP. Múltiples objetos pueden escuchar en el mismo puerto.

Ejemplo aplicación Java RMI

- **Unicast:** Indica que se envía una petición de un proceso a un proceso individual.

java.rmi.server.UnicastRemoteObject;

La comunicación es punto a punto entre procesos, basada en JRMP . Las referencias al objeto remoto únicamente son validas mientras el proceso servidor este vivo.

java.rmi.activation.Activatable;

La clase activable provee soporte para objetos remotos que requieran persistencia, y que puedan ser activados por el Sistema.

javax.rmi.PortableRemoteObject

Permite la funcionalidad básica para funcionar bajo el protocolo RMI-IIOP.

Ejemplo aplicación Java RMI

Tercer paso: Compilar el código fuente correspondiente a la interface y a al lcase que la implementa.

Compilar el código fuente del objeto de implementación con la herramienta *javac* (Compilador java).

```
javac -d ../bin sop_rmi/*.java
```

Ejemplo aplicación Java RMI

Cuarto paso: Implementar el Servidor de Objetos (Ver ejemplo analizado en clase)

Quinto Paso: Implementar el Cliente de Objetos (Ver ejemplo analizado en clase)

Sexto paso: Compilar las clases del servidor y cliente

- Compilar las clases del Servidor de Objetos y el Cliente de Objetos con la herramienta javac. Ubicados en el directorio 'src' el comando sería

Para compilar el servidor de objetos:
javac *-d ../bin servidor/*.java*
Para compilar el cliente de objetos:
javac *-d ../bin cliente/*.java*

Ejemplo aplicación Java RMI

Séptimo paso: Iniciar el registro

Si el registro es lanzado por consola

Tener en cuenta que para lanzar el N_S se debe ubicar en el directorio 'bin'. Porque el rmiregistry necesita conocer los bytecode de las clases que implementan los servicios remotos. Para evitarlo se puede utilizar la variable CLASSPATH para fijar la ruta a los archivos bytecode.

```
rmiregistry [#pto]
```

Ejemplo aplicación Java RMI

Octavo paso: Iniciar la aplicación

a. Ejecutar el servidor:

Comando general:

java Nombre_clase_servidor_obj <dir_IP_NS> <#pto>

Por ejemplo, si la clase pertenece al paquete 'servidor':

java servidor.Servidor localhost 2020

b. Ejecutar el cliente:

Comando general:

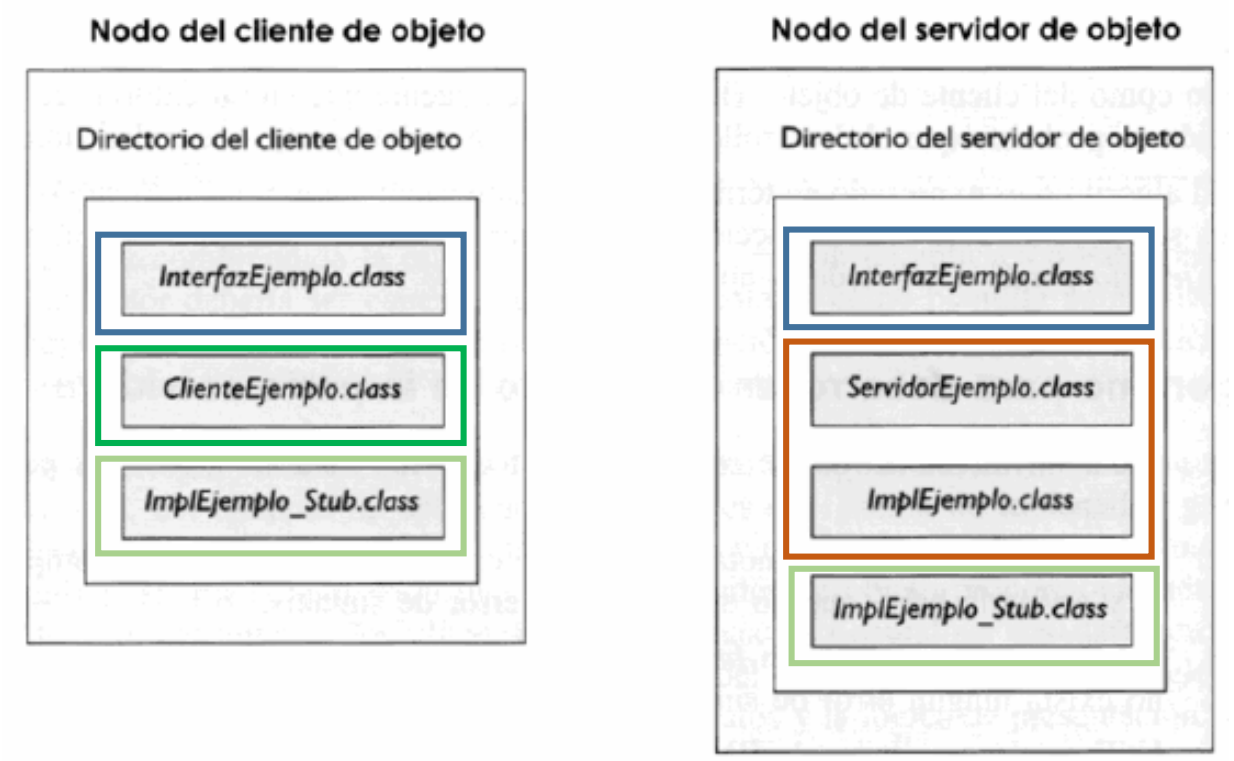
java Nombre_clase_cliente_obj <dir_IP_NS> <#pto>

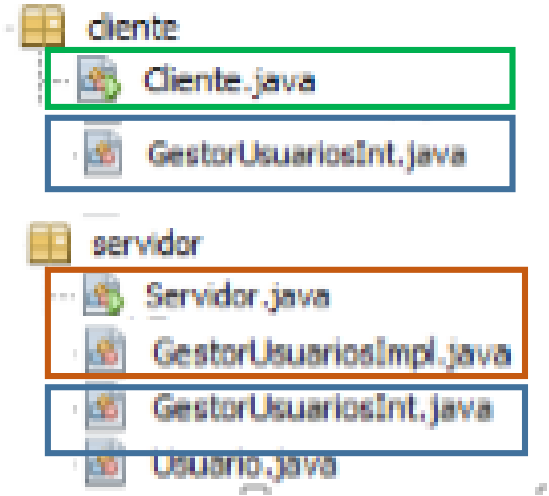
Por ejemplo, si la clase pertenece al paquete 'cliente':

java cliente.Cliente localhost 2020

Ejemplo aplicación Java RMI

Organización de los archivos que componen la aplicación.





PASO DE PARÁMETROS EN JAVA RMI

Concepto de paso por valor

- Al pasar una variable a un método mediante valor lo que se pasa es una copia de la información contenida en esa variable.
- Por lo que tendremos dos instancias diferentes de la misma variable, una que esta en el medio en el que se envió y otra que esta en el medio donde fue enviada,
- Si se modifica la información de la variable enviada, esta solo será cambiada en ese ámbito

PASO DE PARÁMETROS EN JAVA RMI

Concepto de paso por referencia

Cuando se utiliza este mecanismo para pasar argumentos a un método, se proporciona al método una referencia directa a la variable pasada como argumento

Es decir se pasa un apuntador a la dirección en memoria en la que se localiza la variable, por lo que al modificar la información mediante el apuntador en el método al que fue enviada, la variable será modificada en todos los ámbitos ya que en realidad modificamos la variable original

PASO DE PARÁMETROS EN JAVA RMI

En Java solo existe el paso de parámetros por copia, denominado paso por valor.

Al manejar objetos en Java, las variables mantienen una referencia al objeto, por lo tanto cuando pasamos un objeto como parámetro se está realizando una copia de la referencia. Así tenemos dos variables diferentes apuntando al mismo objeto

PASO DE PARÁMETROS EN JAVA RMI

- Un objeto remoto en Java debe implementar la interfaz *java.rmi.Remote* permite que la máquina virtual lo reconozca como tal
- En el lenguaje de programación Java, el paso de argumentos es sólo por valor, sean tipos primitivos u objetos.
- Cuando se utiliza la expresión paso por referencia se refiere al paso por valor de referencias remotas
- La copia por valor se realiza mediante la serialización de objetos

PASO DE PARÁMETROS EN JAVA RMI

El paso de argumentos y los valores de retorno que admiten los métodos remotos tiene las siguientes características:

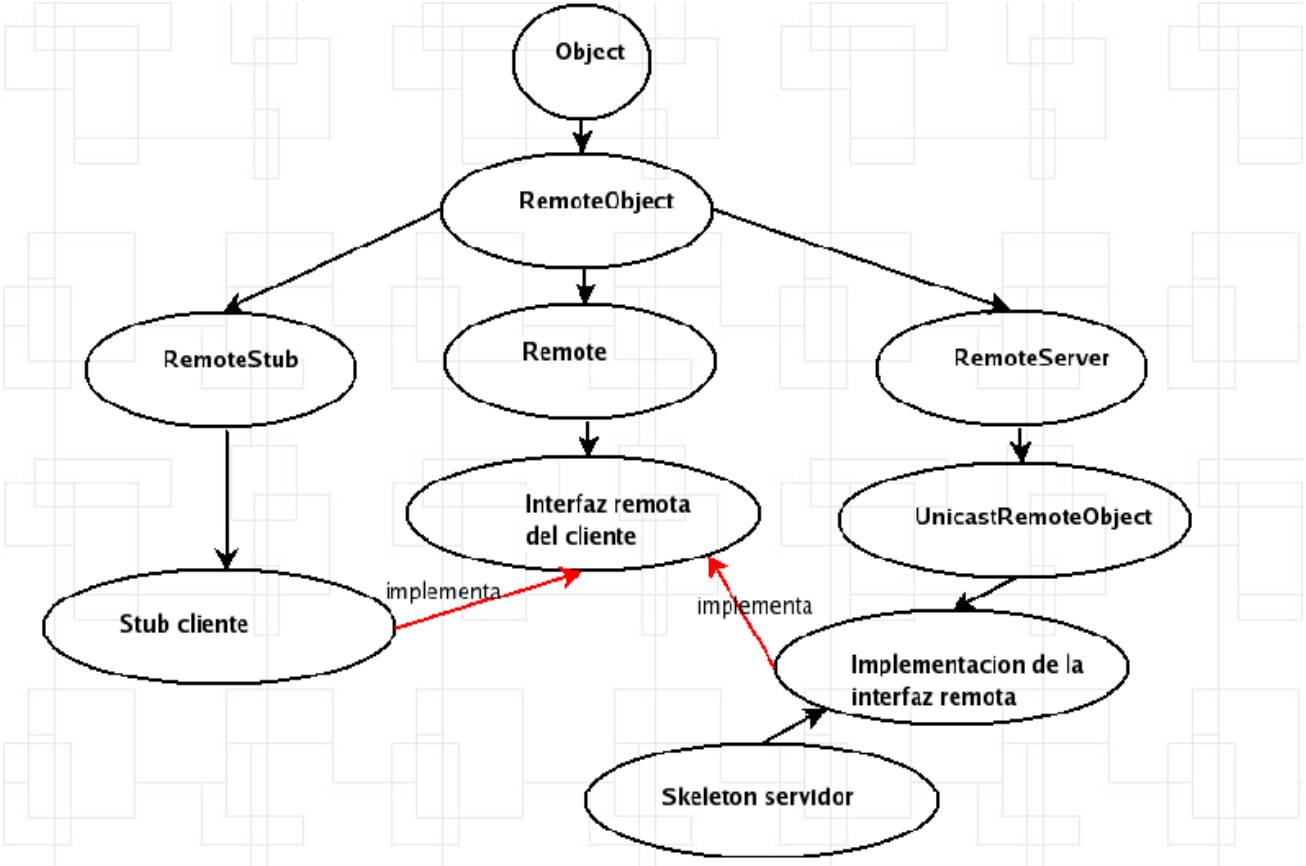
a) Los tipos de datos primitivos se pasan por valor. Es decir, se copian del espacio de direcciones de una máquina virtual a otra.

b) Los objetos (no remotos) cuyas clases implementen la interfaz *java.io.Serializable* se pasan por valor.

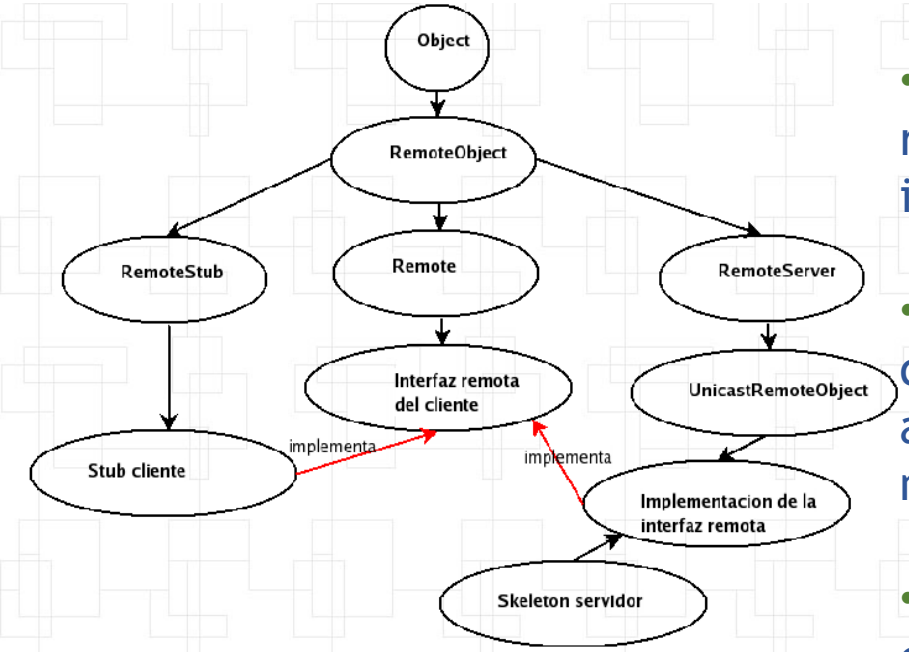
c) Los objetos remotos que estén a la espera de peticiones llegadas desde los clientes no se transmiten sino que, en su lugar, se envían las referencias remotas a los mismos (instancias de un *stub*).

d) Los objetos (no remotos) que no implementan *java.io.Serializable* no pueden enviarse a un objeto remoto.

JERARQUÍA DE CLASES EN JAVA RMI



JERARQUÍA DE CLASES EN JAVA RMI

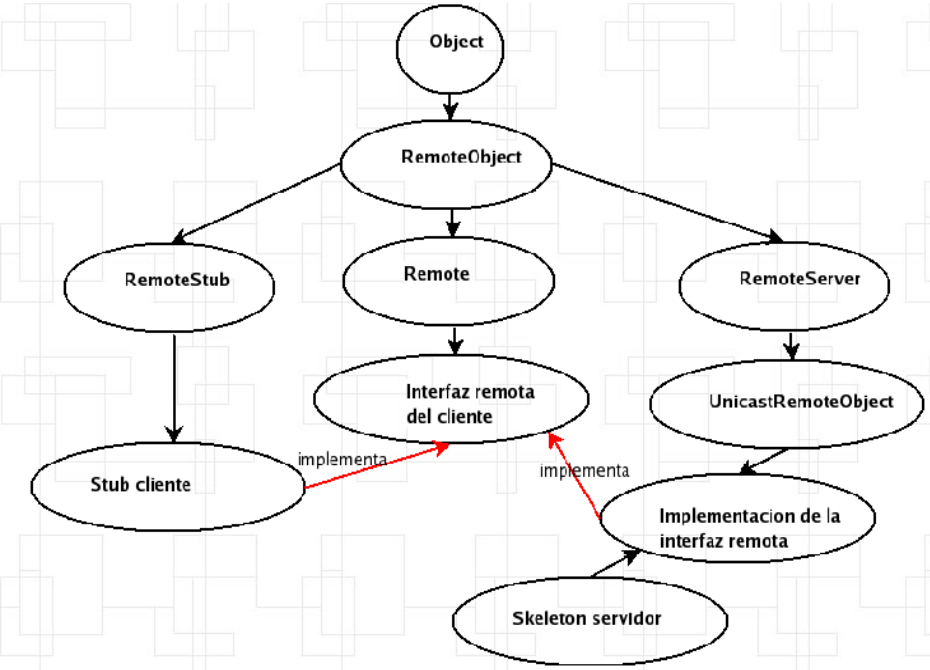


- **Clase *RemoteObject***: Implementa la clase *java.lang.Object* (clase raíz de todas las clases Java) para objetos remotos y la interfaz *java.rmi.Remote*.

- **Clase *RemoteServer***: Hereda de *RemoteObject*. Es la clase raíz de la que heredan todas las implementaciones de objetos cuyos métodos son accesibles remotamente, y proporciona la semántica básica para el manejo de referencias remotas.

- **Clase *RemoteStub***: Hereda de *RemoteObject*. Es la clase raíz de la que heredan todos los *stubs* de los clientes.

JERARQUÍA DE CLASES EN JAVA RMI



- Clase *RMIClassLoader*: Incluye métodos estáticos para permitir la carga dinámica de clases remotas. Si un cliente o servidor de una aplicación RMI necesita cargar una clase desde un host remoto, llama a esta clase.

- Clase *RMISocketFatory*: Usada por RMI para obtener sockets cliente y servidor para las llamadas RMI.

- Clase *UnicastRemoteObject*: Subclase de *RemoteServer*. Permite una comunicación punto a punto. Permite que un objeto se exporte a si mismo.

- *Clase activable:* provee soporte para objetos remotos que requieran persistencia, y que puedan ser activados por el Sistema.

MANEJO DE EXCEPCIONES EN JAVA RMI

- Bajo un entorno de red es necesario que los fallos puedan ser notificados para realizar su posterior manejo.

- Por ejemplo: Los clientes deben lanzar (throw) la excepción java.rmi.RemoteException para casos de error general.

- Existen otras subclases para condiciones específicas:

    - Exceptions During Remote Object Export
    - Exceptions During RMI Call
    - Exceptions or Errors During Return
    - Naming Exceptions
    - Other Exceptions

MANEJO DE EXCEPCIONES EN JAVA RMI

- Al ejecutar el método bind puede lanzarse las siguientes excepciones: AlreadyBoundException, MalformedURLException, RemoteException
- Al ejecutar el método rebind puede lanzarse las siguientes excepciones: RemoteException, MalformedURLException
- Al ejecutar el método lookup puede lanzarse las siguientes excepciones: NotBoundException,MalformedURLException,RemoteException
- Al serializar los argumentos de un método a la respuesta aun método puede lanzarse la excepción: MarshalException
- Al conectarse con un host remoto, si la conexión se rechaza puede lanzarse la excepción: ConnectException.

El gestor de seguridad de RMI

- En versiones superiores a Java 2, una aplicación Java debe obtener primero, antes de ejecutarse, información acerca de sus privilegios o permisos para acceder a los recursos del sistema
- Estos permisos se definen en Java mediante la definición de una política de seguridad y una aplicación puede obtener una política a través de un fichero ".policy".
- Se debe definir una política de seguridad para Java RMI mediante el establecimiento de la propiedad java.security.policy a un fichero específico, por ejemplo, politicas.policy

POLÍTICAS DE SEGURIDAD EN JAVA RMI

En el ejemplo siguiente ejemplo, se da, tanto al servidor como al cliente, todos los permisos ya que se utiliza la política de seguridad definida en el archivo “miArchivo.policy” que contiene una única regla:

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

La política de seguridad puede ser más restrictiva, como se pone de manifiesto en el siguiente “java.policy” que incluye las siguientes reglas:

```
grant {
    permission java.io.filePermission “/tmp/*”, “read”, “write”;
    permission java.net.SocketPermission “host.dominio.com:999”,”connect”;
    permission java.net.SocketPermission “*:1024-65535”,”connect,request”;
    permission java.net.SocketPermission “*:80”,”connect”;
};
```

POLÍTICAS DE SEGURIDAD EN JAVA RMI

1. Permite a cualquier aplicación Java, leer/escribir cualquier fichero que esté en el directorio /tmp (incluidos subdirectorios).

2. Se permite a todas las clases Java establecer una conexión de red con la máquina “host.dominio.com” a través del puerto 999.

3. Permite a todas la clases permiso para conectarse o aceptar conexiones a través de puertos no privilegiados mayores que 1024 (puertos 1024-65535) y desde o hacia cualquier host (carácter comodín *).

4. Permite a todas las clases que se ejecutan en la JVM local conectarse al puerto 80 de cualquier otra máquina (conexiones http).

POLÍTICAS DE SEGURIDAD EN JAVA RMI

La clase java.lang.SecurityManager establece un control de seguridad sobre aplicaciones RMI. Los permisos que establece el gestor de seguridad se encuentran en el archivo de políticas de seguridad creado. Automáticamente son fijados al gestor de seguridad, por medio del AccessController.checkPermission.

Se puede establecer un gestor de seguridad de la siguiente manera:

```
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
}
```

CALLBACK EN RMI

Se presenta cuando el servidor solicita un llamado remoto a un cliente. Por ejemplo un servidor necesita notificar a otros procesos la ocurrencia de un evento

¿Como un cliente puede enterarse de la ocurrencia de un evento en el proceso servidor?

POLLING

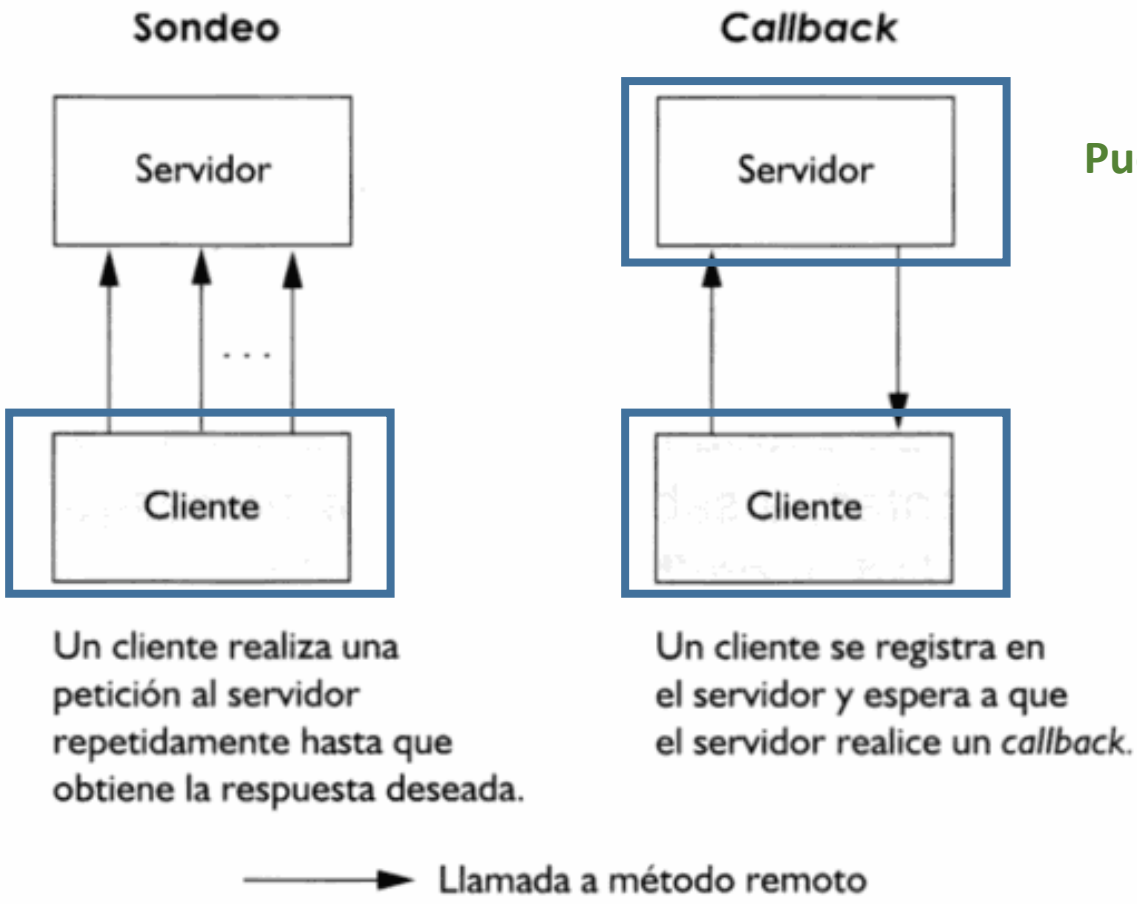
Un cliente invoca de forma repetitiva un método remoto

CALLBACK

Un objeto remoto cliente se registra en un objetor servidor remoto. Cuando ocurre un evento el objeto servidor le informa al cliente

CALLBACK EN RMI

Inicia la comunicación

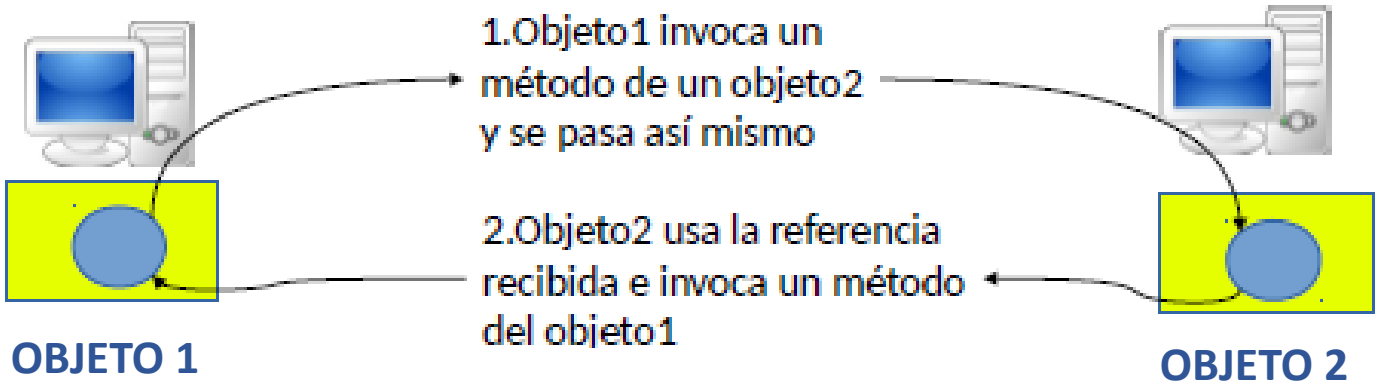


Puede iniciar la comunicación

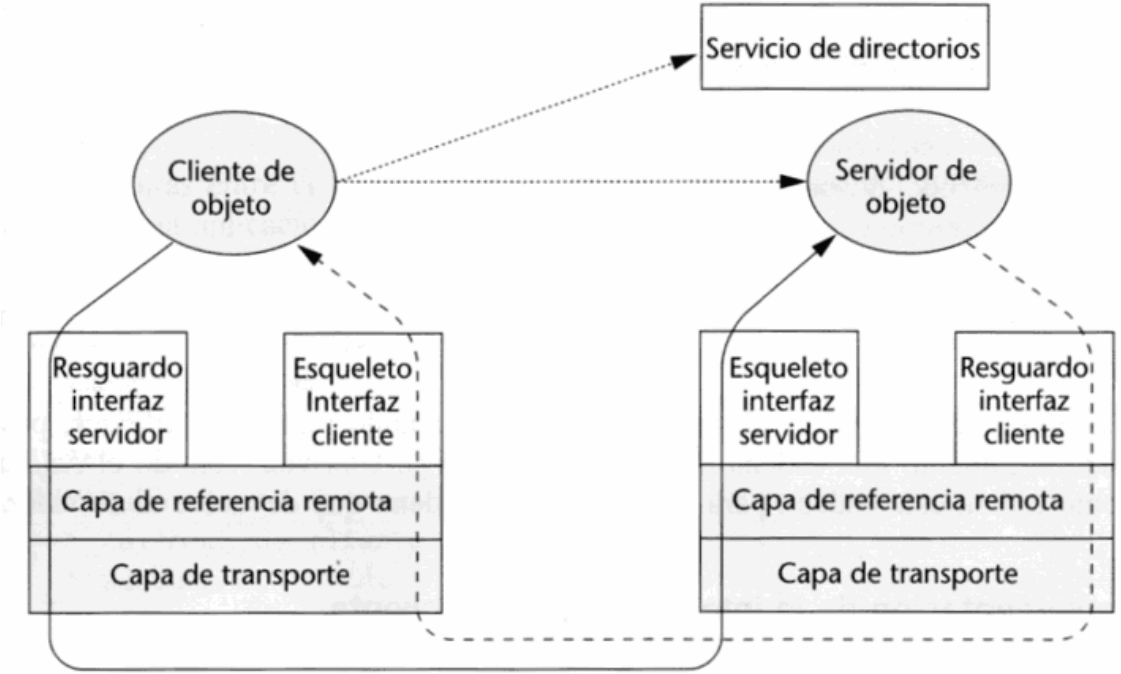
Inicia la comunicación

CALLBACK EN RMI

Cada cliente interesado en la ocurrencia de un evento se debe registrar en el servidor de objeto.



ARQUITECTURA DE CALLBACK EN RMI



- **Con un callback las invocaciones de los métodos remotos se vuelven bidireccionales**
- **Son necesarios dos objetos remotos. (Ver ejemplo explicado en clase)**

Sincronización del acceso a la estructura que almacena la referencia a los clientes

El método registrar usuario modifica una estructura común.

```
public class ServidorCllbckImpl extends UnicastRemoteObject implements ServidorCllbckint {

    private List<UsuarioCllbckInt> usuarios;

    public ServidorCllbckImpl() throws RemoteException
    {
        super();
        usuarios= new ArrayList<UsuarioCllbckInt>();
    }
```

Como el método se puede ejecutar concurrentemente es importante que la estructura común se proteja con exclusión mutua, la cual se consigue por medio de métodos sincronizados.

Sincronización del acceso a la estructura que almacena la referencia a los clientes

El método registrar usuario modifica una estructura común, esta marcado como un método synchronized.

- Cada vez que un hilo intenta acceder a un bloque sincronizado de un objeto le pregunta al objeto si no hay algún otro hilo que este ejecutando ese bloque de código.
- Si hay otro hilo ejecutando ese bloque de código, entonces el hilo actual es suspendido y puesto en espera hasta que el otro hilo termine. Cuando el hilo termina entonces el nuevo hilo entra a ejecutar el bloque.
- Si hay dos bloques synchronized que hacen referencia a distintos objetos (por más que ambos utilicen el mismo nombre de variable), la ejecución de estos bloques **no** será mutuamente excluyente.

CARGA DINÁMICA DE CLASES

Permite que el código ubicado en una maquina sea compilado en otra.

Como sabe el sistema desde donde cargar las clases:

- CLASSPATH: Variable de entorno que permite establecer donde buscar las clases a cargar.
- Codebase: Establece una URL donde se buscan las clases a cargar.

La propiedad java.rmi.server.codebase es usada para especificar una URL (sftp://,ftp://,http://)

Si la propiedad java.rmi.server.useCodebaseOnly es fijada a true, las clases son cargadas de acuerdo a la URL especificada en codebase

CARGA DINÁMICA DE CLASES

- Permite la carga de clases desde servidores FTP o HTTP (Proceso de carga de applets)
- La carga de clases remota se hace a través de RMIClassLoader
- El control de la carga de clases se hace fijando ciertas propiedades de la JVM

  java [D<PropertyName>=<PropertyValue>]+<ClassFile>

CARGA DINÁMICA DE CLASES

Configuraciones del sistema RMI

Closed: No hay carga dinámica, las clases son localizadas en la JVM y ubicadas vía CLASSPATH

Dinámica del lado cliente: la clase principal (C.O) es cargada de acuerdo al CLASSPATH, las otras clases de acuerdo al codebase

Dinámica del lado servidor: la clase principal (S.O) es cargada de acuerdo al CLASSPATH, las otras clases de acuerdo al codebase

CARGA DINÁMICA DE CLASES

Configuraciones del sistema RMI

Bootstrap cliente: Todo el código cliente es cargado de acuerdo al codebase, vía un pequeño programa en el cliente(bootstrap loader), que busca las clases desde un servidor central.

Bootstrap servidor: Todo el código servidor es cargado de acuerdo al codebase, vía un pequeño programa en el servidor(bootstrap loader), **que busca las clases desde un servidor central.**

Bootstrap cliente y servidor: En esta configuración todo el código del cliente, y todo el código del servidor es cargado de acuerdo al codebase, vía un pequeño programa en la máquina cliente y la máquina servidora (bootstrap loader), **que busca las clases desde un servidor central.**

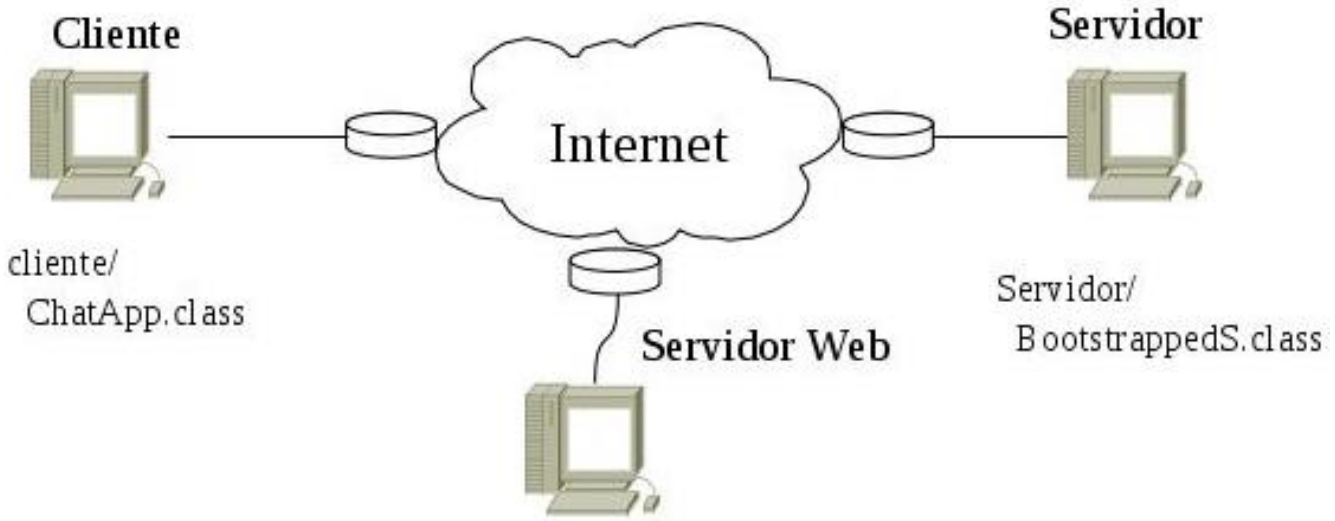
.

CARGA DINÁMICA DE CLASES

Mediante un gráfico donde aparezca el nodo cliente y el nodo servidor y otros nodos que considere necesarios indicar la distribución de archivos(.class), colocar nombres, que debe tener cada nodo si se aplicará las siguientes configuraciones:

- Carga Dinámica de Clases en Java RMI(con API jdk1.1), teniendo en cuenta que el cliente usa el modo de configuración dinámica del lado del cliente y el servidor usa el modo de configuración 'Bootstrapped' en una aplicación.
- Carga Dinámica de Clases en Java RMI(con API jdk1.1), teniendo en cuenta que el cliente usa el modo de configuración dinámica del lado del cliente y el servidor usa el modo de configuración dinámica del lado del servidor en una aplicación.
- Carga Dinámica de Clases en Java RMI(con API jdk1.4), teniendo en cuenta que el cliente usa el modo de configuración 'Bootstrapped y el servidor usa el modo de configuración ‘dinámica del lado del servidor’ en una aplicación.
- Carga Dinámica de Clases en Java RMI(con API jdk1.8), teniendo en cuenta que el cliente usa el modo de configuración 'Bootstrapped y el servidor usa el modo de configuración ‘'Bootstrapped ’ en una aplicación.

CARGA DINÁMICA DE CLASES



ClienteImpl_stub.class
ClienteImpl_skel.class

rmic –v1.2 NombreClaseImpl

ServidorImpl_stub.class
ServidorImpl_skel.class

RECOLECTOR DE BASURA DISTRIBUIDO

• La gestión de la memoria (asignación y liberación) es uno de los principales problemas que existen a la hora de desarrollar software sin fallos.

• Normalmente es sencillo saber cuándo hay que crear un nuevo objeto, pero suele ser más difícil saber cuándo hay que liberarlo.

• En lenguajes de bajo nivel (C, C++, etc), este problema es difícil de resolver

• Los lenguajes más modernos (Java, C#, etc) poseen un mecanismo que permite “liberar” la memoria de un objeto de manera automática cuando este ya no es necesario, por medio de un servicio denominado recolector de basura

RECOLECTOR DE BASURA DISTRIBUIDO

- En un entorno local un recolector de basura es un servicio que elimina todos aquellos objetos que no poseen una referencia activa hacia ellos.

- En un entorno distribuido se debe controlar los objetos remotos “huerfanos”, es decir aquellos objetos remotos que no son referenciados por ningún cliente de objetos.

- Cuando un cliente de objetos obtiene un stub de un objeto remoto, se crea una referencia al objeto remoto.

RECOLECTOR DE BASURA DISTRIBUIDO

- Las referencias a los objetos (*stubs*) pueden estar en otros procesos
- Cada instancia de un objeto está asociada a un ObjID
- Idea:
  - El RMI runtime mantiene una "cuenta" de los *stubs* que se han generado con un ObjID determinado
  - Cada vez que un *stub* es recolectado en un cliente (deja de ser alcanzable), se notifica al runtime del servidor, que resta una unidad
  - Cuando esta cuenta llega a 0, el objeto RMI se puede recolectar
- Problemas:
  - ¿Qué pasa si un cliente tiene un fallo de *crash*?
  - ¿Qué pasa si hay un fallo de partición en la red?
- Solución:
  - Java RMI propone el uso del "arrendamiento" (*leasing*)
  - El leasing permite solucionar los problemas anteriores

RECOLECTOR DE BASURA DISTRIBUIDO

- El Recolector de Basura Distribuido es un servicio RMI (igual que el Registry)

- Se puede construir una referencia al mismo partiendo de un stub (el DGC que controla un servidor está en la misma JVM que el propio servidor)

- La interfaz DGC tiene solo dos métodos
- public void clean(…)
  - Se utiliza para "liberar" un lease que se posee sobre un ObjID
  - Podemos no invocarlo y el lease se liberará cuando expire
- public Lease dirty(…)
  - Se utiliza para adquirir un *lease* sobre un ObjID
  - El RMI *runtime* del cliente invoca este método de manera transparente y periódica cuando tenemos un *stub* asociado a ese ObjID que es alcanzable
- Cada JVM tiene una sola instancia del servicio DGC

REFERENCIAS

- M.L.Liu, Computación Distribuida: Fundamentos y Aplicaciones, Pearson Educación, S.A Madrid, 2004, ISBN:84-7829-066-4.
- *Andrew S. Tanenbaum. Sistemas Operativos, Diseño e implementación.*
- *Subrahmanyam Allamaraju, et al. Professional Java Server Programming J2EE 1.3 Edition. Wrox Press Ltd.*
- *jGuru: Remote Method Invocation (RMI)http://java.sun.com/developer/OnlineTraining/rmi/RMI.html*