Desarrollo de Aplicaciones con Java RMI

Laboratorio de Ingeniería del Software

Contenido

- Antecedentes
- Introducción a los Sistemas Distribuidos
  - Sockets
  - RPC
- Java RMI
  - Características
  - Modelo de Objetos Distribuidos
  - Visión del Sistema RMI
  - Arquitectura RMI
  - Pasos para desarrollar aplicaciones en RMI
  - Ejemplos:
    - P1) Cronometrar transferencia de datos
    - P2) Cliente Descarga Códigos de un servidor
    - P3) P2 + Polimorfismo y Sobrecarga

Un poco de historia

~1990

- James Gosling, Patrick Naughton y Mike Sheridan
  ... (First Person Inc.) http://java.sun.com/people/jag/index.html

..... **OAK** ..... **JAVA**

~1994

- Sun Microsystems

..... Sistemas Distribuidos: CORBA

"A Note on Distributed Computing"
http://www.sunlabs.com/technical-reports/1994/abstract-29.html

¿ Qué buscaba SUN ?

- Las diferencias entre la computación local y la computación distribuida.
- En el caso de CORBA, "no existe" diferencia.
- SUN concluyó que los sistemas distribuidos deben tratar de forma diferente a los objetos locales y a los remotos:
  - Latencias
  - Fallos en las comunicaciones
  - Concurrencia
  - Acceso a memoria

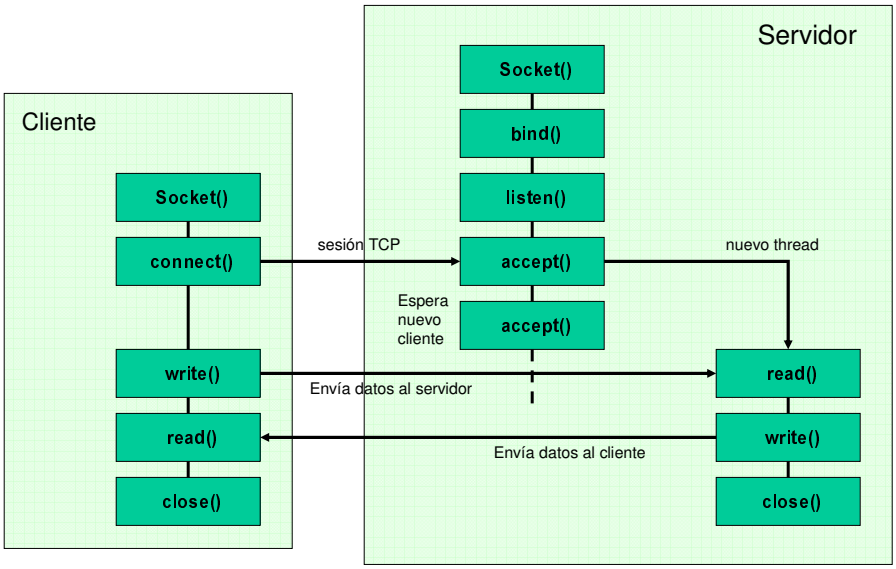
Sistemas Distribuidos

- Los sistemas distribuidos requieren la ejecución de tareas en diferentes máquinas con capacidad de comunicarse entre ellas.
- Desarrollo de Sistemas Distribuidos:
  - Sockets
  - Remote Procedure Calls (RPC)
  - Remote Method Invocation (RMI)

Sockets

- Permite las comunicaciones entre dos máquinas con IP.
- Existen varias alternativas:
  - UDP
  - TCP
    - Java ..... Buffered, Typed, etc.
- Es necesario un protocolo sincronizado entre cliente y servidor DEPENDIENTE de cada aplicación.
- Requiere gran cantidad de operaciones a la hora de un simple intercambio de datos.

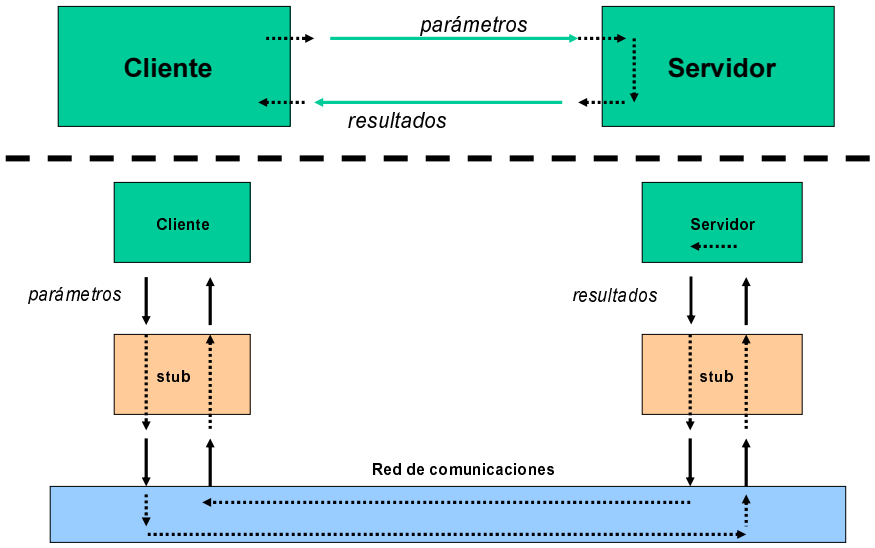
Sockets



Remote Procedure Call (RPC)

- Las RPC permiten la abstracción del sistemas de comunicaciones a una simple llamada a un procedimiento.
- En lugar de trabajar con *sockets*, el programador cree que está haciendo llamadas a métodos locales. Los parámetros de las llamadas son empaquetados y enviados al destino.
- RPC utiliza la codificación XDR el envío y recepción de los datos.
- Principal inconveniente: NO ORIENTADO A OBJETOS

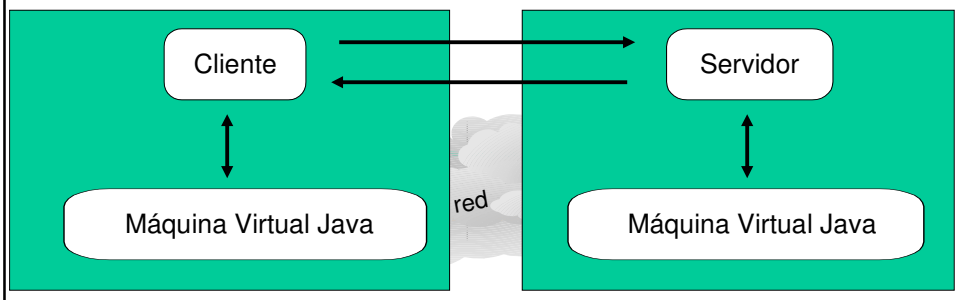
Remote Procedure Call (RPC)



8/48

¿ Qué es Java RMI ?

- Es una especificación desarrollada por JavaSoft desde la implementación del JDK 1.1
  - Permite la ejecución de métodos localizados en objetos remotos que a su vez se ejecutan en una máquina virtual Java diferente.
  - La invocación de métodos no es más que paso de mensajes.



9/48

Características de RMI

- Orientado a Objetos: Java.
- Permite la invocación de métodos de objetos remotos Java entre diferentes máquinas virtuales.
- Incorpora el modelo de objetos distribuido en el modelo Java manteniendo su semántica.
- Uso de las APIs de Java, con todas sus posibilidades.
- Permite un desarrollo sencillo de las aplicaciones distribuidas.
- Permite que un *applet* pueda llamar a un programa Java en el servidor.
- Mantiene el esquema de seguridad de Java.

Modelo de Objetos Distribuidos (RMI)

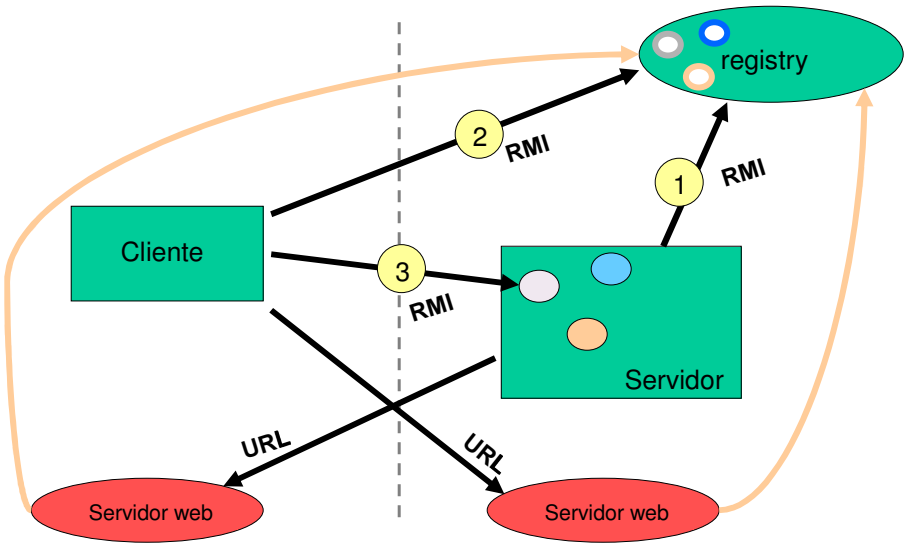
- Aplicaciones de Objetos Distribuidos:
  - Servidor:
    - Crear objetos
    - Permitir referencias a ellos
    - Esperar por peticiones de los clientes
  - Cliente:
    - Obtener referencia a los objetos remotos
    - Invocar a los métodos de los objetos remotos
- RMI ofrece el mecanismo de comunicaciones para el paso de información entre el cliente y el servidor

Modelo de Objetos Distribuidos (RMI)

- Las aplicaciones de Objetos Distribuidos requieren:
  - Localizar objetos remotos:
    - El servidor debe registrar el objeto remoto para que los clientes obtengan sus referencias.
  - Comunicarse con objetos remotos:
    - El sistema RMI lleva a cabo todas las comunicaciones existentes en las llamadas y retornos de resultados entre cliente y servidor.
  - Cargar los bytecodes de las clases de los objetos que son pasados como argumentos o devueltos como resultados.
  - Liberar objetos no referenciados a través del *garbage collector* distribuido.
  - Excepciones nuevas: “Remote Exception”

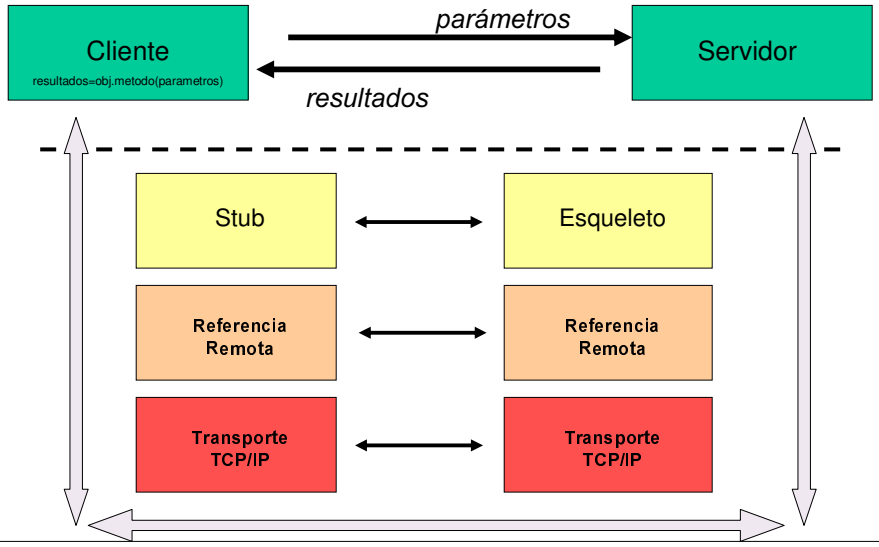
12/48

Visión del sistema RMI



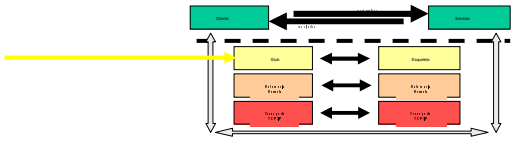
13/48

Arquitectura del Sistema RMI



14/48

Arquitectura:.....Capa *stub* y esqueleto

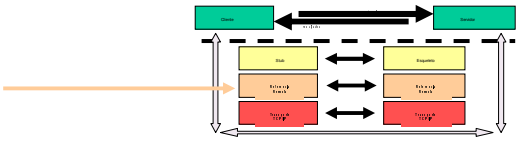


- Características:
  - Los objetos se transmiten entre emisor y receptor a través de una serialización.
  - El *stub* implementa la inicialización de la llamada a la capa de referencia remota, la serialización de los argumentos del método remoto llamado y la deserialización de los resultados.
  - El esqueleto cumple la función complementaria para el programa servidor responsable del objeto.

- El compilador *rmic* genera tanto el *stub* como el esqueleto a partir de la definición del objeto en el programa fuente Java.

15/48

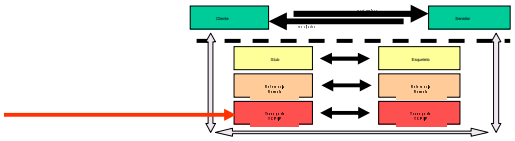
Arquitectura:.....Capa de Referencia Remota



- Responsable de los detalles de transmisión de la llamada a los métodos remotos.
- Implementa varios protocolos de comunicaciones. Estos permiten:
  - Invocación punto a punto.
  - Invocación a un grupo de objetos.
  - Soporte de persistencia y activación de objetos remotos en el servidor.
  - Estrategias de Recuperación de conexión.

16/48

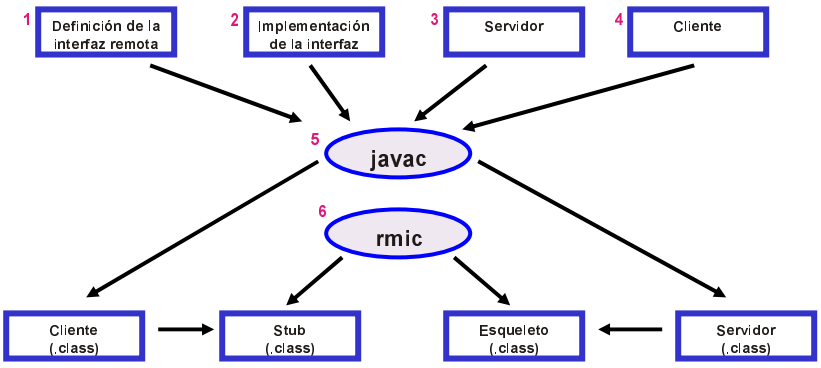
Arquitectura:.....Capa de Transporte



- La capa de transporte se encarga de crear y mantener las conexiones existentes entre las capas de referencia de diferentes máquinas virtuales Java.
- Además mantiene las tablas de correspondencias entre máquinas virtuales y objetos disponibles en una dirección URL dada.

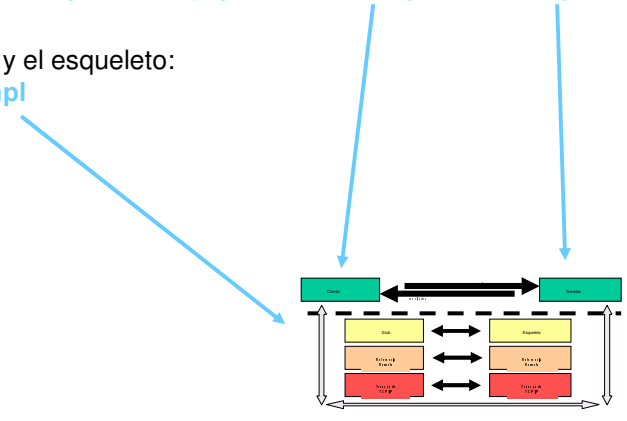
17/48

Pasos para escribir aplicaciones con Java RMI



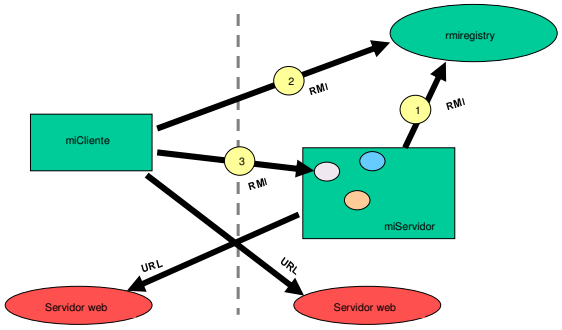
Instrucciones para COMPILAR las aplicaciones

- Compilar Interfaz, Implementación, Servidor y Cliente:
  javac miInterfaz.java miImpl.java miCliente.java miServ.java
- Crear el *stub* y el esqueleto:
  rmic miImpl



Instrucciones para EJECUTAR las aplicaciones

- Ejecutar el servicio de registro de objetos:
  start rmiregistry (windows), **rmiregistry &** (unix)
- Ejecutar servidor:
  java -Djava.security.policy=mijava.policy miServ
- Ejecutar cliente:
  java miCliente



Seguridad en RMI

- Directiva a la máquina virtual **"-Djava.security.policy"** indicando el fichero con la política de seguridad: **"mijava.policy"**
- Ejemplos de fichero de seguridad:

```
grant {
    // permitir todo
    permission java.security.AllPermission;
};
```

```
grant {
   permission java.net.SocketPermission "*:1024-65535","connect, accept";
   permission java.net.SocketPermission "*:80", "connect";
}
```

Ejemplos Prácticos

P1) Medir tiempos en transferencia de datos

P2) Cliente descarga códigos de un servidor

P3) P2 + Polimorfismo y Sobrecarga

P1) Medir tiempos en transferencia de datos

- **OBJETIVO:**
  - Transferencia de datos entre dos ordenadores haciendo uso de RMI.
- **PASOS**:
  - Crear el servicio para el cliente ---> interfaz
  - Implementar los servicios ---------> implements ...
  - Crear el servidor
  - Crear el cliente
- **ESPECIFICACIÓN**:
  - Tomar los tiempos de transferencia de enteros entre dos máquinas en un rango entre 1 y 1048576 enteros.

Esquema de la práctica

1) Interfaz Remota

```
// TransferRMI ..... Interfaz

public interface ITransferRMI extends java.rmi.Remote {

    public int initSend () throws java.rmi.RemoteException;

    public int[] getData(int size) throws java.rmi.RemoteException;

}
```

2) Implementación de la Interfaz

```
public class TransferRMIImpl extends UnicastRemoteObject implements ITransferRMI {

  public TransferRMIImpl(String name) throws RemoteException  {
     try  {
        Naming.rebind(name, this);
     } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
     }
  }
  public int initSend() throws RemoteException {   return 1;   }
  public int[] getData(int size) throws RemoteException {
    int k;
    int[] buffer = new int[size];
    for (k=0; k<tam; k++) buffer[k] = k;
    return buffer;
  }
} // de la clase TransferRMIImpl
```


3) Codificar el Servidor

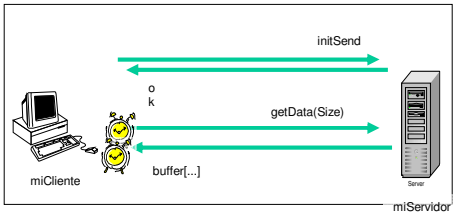
```
import java.rmi.*;
import java.rmi.server.*;

public class TransferRMIServer {
  public static void main(String args[]) {
    try   {
      // Create TransferRMIImpl
      TransferRMIImpl ObjImpl = new TransferRMIImpl("Transfer_Obj");
      System.out.println("CountRMI Server preparado.");
    } catch (Exception e) {
      System.out.println("Exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
} // de la clase TransferRMIServer
```

4) Codificar el Cliente

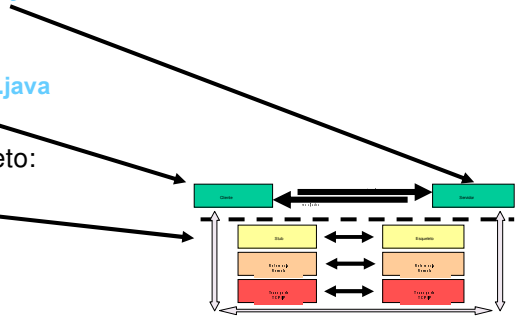
```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class TransferRMIClient {
  public static void main(String args[]) {
    int[] buffer = new int[1];
      .....
      // Localizamos el objeto remoto Obj
      TransferRMI Obj = (TransferRMI)Naming.lookup("rmi://"+ args[0] + "/" +
                                                   "Transfer_Obj");
      for (i=0; i<Size;i++) {
        int ok = Obj.initSend();
        // Iniciar cronómetro
        buffer = Obj.getData(size);
          // Parar cronómetro
        }
   } // del main
} // de la clase TransferRMIClient
```



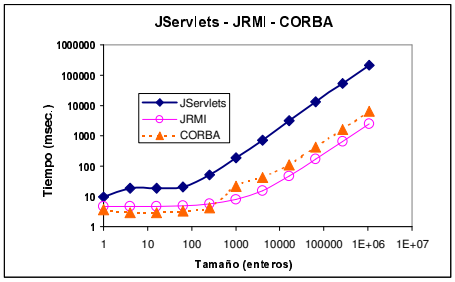
COMPILAR

- Compilar Interfaz y objetos tareas:
  javac ITransferRMI.java TransferRMIImpl.java
- Compilar servidor:
  javac TransferRMIServer.java
- Compilar el Cliente:
  javac TransferRMIClient.java
- Crear el *stub* y el esqueleto:
  rmic Server



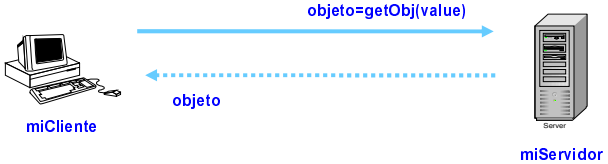
EJECUTAR

- Ejecutar el servicio de registro de objetos:
  start rmiregistry (windows), **rmiregistry &** (unix)
- Ejecutar servidor:
  java -Djava.security.policy=mijava.policy TransferRMIServer
- Ejecutar cliente: host INI FIN x REP
  java TransferRMIClient 193.145.100.161 1 1048576 2 10

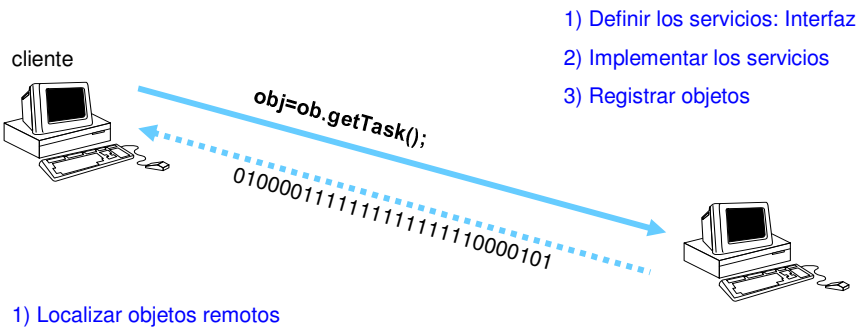


Objetos viajeros

- Una de las características más interesantes de Java RMI es la capacidad por descargar software en tiempo de ejecución.
- Los valores de los parámetros del cliente al servidor y los valores de los resultados del servidor al cliente son serializados. Todos los tipos primitivos y aquellos con capacidad de serialización pueden viajar por la red.
- Los objetos que implementan la clase "*java.io.serializable*" también pueden viajar por la red.



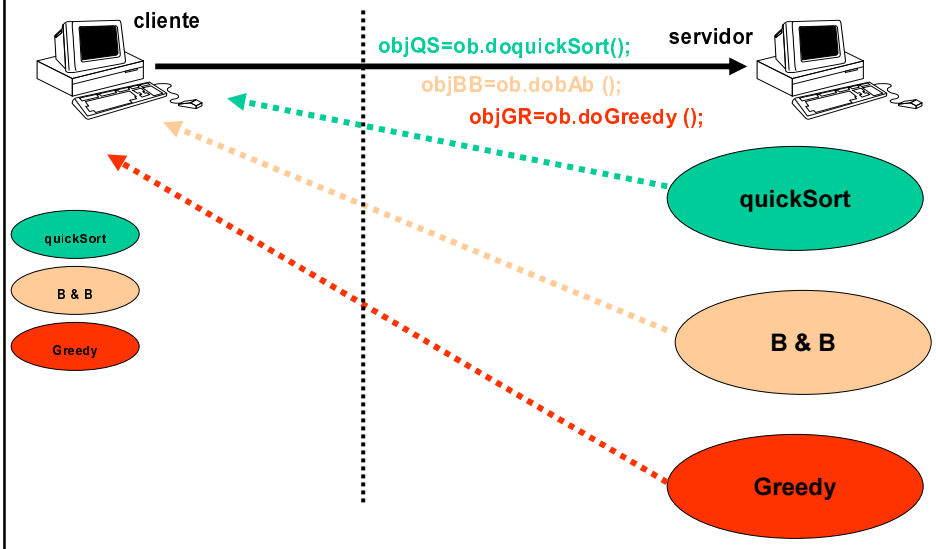
Visión de la descarga de código



P2) Cliente descarga códigos de un servidor

- **OBJETIVO:**
  - Paso de objetos entre cliente y servidor para la ejecución de tareas en el cliente que originalmente están en el servidor.
- **PASOS**:
  - Crear el servicio para el cliente ---> interfaz con las tareas
  - Implementar los servicios ---------> implements ...
  - Crear el servidor
  - Crear el cliente
- **ESPECIFICACIÓN**:
  - Ejecutar la aplicación.
  - Modificarla para incorporar nuevas tareas.

Ejecución de diferentes tareas

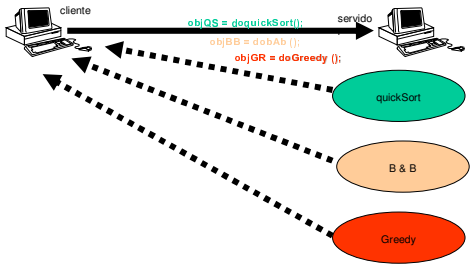


34/48

1) Interfaz Remota (InterfazTarea.java)

```
// Interfaz de Tarea InterfazTarea (InterfazTarea.java)

public interface InterfazTarea extends java.rmi.Remote {
   quickSort doquickSort() throws java.rmi.RemoteException;
   bAb       dobAb() throws java.rmi.RemoteException;
   Greedy    doGreedy() throws java.rmi.RemoteException;
}
```



35/48

2) Implementación de las tareas

```
// Objeto quickSort (quickSort.java)
public class quickSort implements java.io.Serializable {
  public void computo() {
    System.out.println("......Realizando un QuickSort !!");
  }
}
```

```
// Objeto bAb (bAb.java)
public class bAb implements java.io.Serializable {
  public void computo() {
    System.out.println("......Realizando un Branch and Bound !!");
  }
}
```

```
// Objeto Greedy (Greedy.java)
public class Greedy implements java.io.Serializable {
  public void computo() {
    System.out.println("......Realizando un Greedy !!");
  }
}
```


3) Implementación Interfaz en el Server (Server.java)

```
public class Server extends UnicastRemoteObject implements InterfazTarea  {
        public static final void main(String[] args) {
                try {
                        Server runner = new Server();
                        Naming.rebind("jobs_server",runner);
                } catch (Exception e) {
                        System.exit(1);
                }
        }
        public quickSort doquickSort() {
                return (new quickSort());
        }
        public bAb dobAb() {
                return (new bAb());
        }
        public Greedy doGreedy() {
                return (new Greedy());
        }
}
```

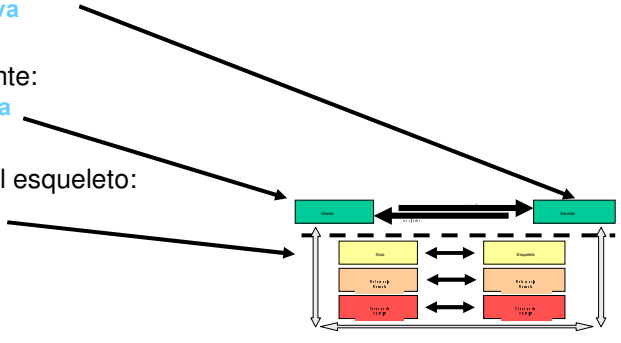
4) Codificar el Cliente (Client.java)

```
public class Client {
    InterfazTarea objTarea;
    public static final void main(String[] args) {
        Client miCliente = new Client();
        miCliente.miproceso();
    }
    public Client() {
        try { objTarea = (InterfazTarea)Naming.lookup("rmi://host/jobs_server");}
        catch(Exception e) { System.out.println("Error:  " + e.getMessage()); }
    }
    public void miproceso() {
        try {
            quickSort miquickSort = objTarea.doquickSort();
            miquickSort.computo();
            bAb mibAb = objTarea.dobAb();
            mibAb.computo();
            Greedy miGreedy = objTarea.doGreedy();
            miGreedy.computo();
        } catch (Exception e) { System.out.println("Error: " + e.getMessage()); }
    }
}
```

38/48

COMPILAR

- Compilar Interfaz y objetos tareas:
  javac InterfazTareas.java quickSort.java bAb.java Greedy.java
- Compilar servidor:
  javac Server.java
- Compilar el Cliente:
  javac Client.java
- Crear el *stub* y el esqueleto:
  rmic Server



39/48

EJECUTAR

- **Ejecutar el servicio de registro de objetos:**
  start rmiregistry (windows), rmiregistry & (unix)
- **Ejecutar servidor:**
  java -Djava.security.policy=mijava.policy Server
- **Ejecutar cliente:**
  java Client

```
Servidor creado !!
Objeto registrado ok !!
Servidor preparado !!
Envio objeto tipo quickSort() al cliente !!  ->  Creado objeto cliente !!
Envio objeto tipo bAb() al cliente !!        ->  .....Realizando un QuickSort !!
Envio objeto tipo Greedy() al cliente !!     ->  .....Realizando un Branch and Bound
                                                 .....Realizando un Greedy !!
```


P3) P2 + Polimorfismo y Sobrecarga

- Ejemplo anterior, si los objetos **quickSort**, **bAb** y **Greedy** hubiesen sido extendidos desde una clase superior, **Tarea**; desde las llamadas al método remoto, nos podría devolver en el mismo parámetro uno u otro objeto.

```
// Interfaz de Tarea InterfazTarea (InterfazTarea.java)

public interface InterfazTarea extends java.rmi.Remote {
   Tarea  doquickSort() throws java.rmi.RemoteException;
   Tarea  dobAb() throws java.rmi.RemoteException;
   Tarea  doGreedy() throws java.rmi.RemoteException;
}
```

Polimor... y Sobrecar... "*REMOTO*"

```
// Objeto Tarea del que se derivan las demás tareas (Tarea.java)

import java.io.*;
public class Tarea implements java.io.Serializable {
  String msg;
  public Tarea() {
    super();
  }
  public Tarea(String msg) {
    this.msg = msg;
  }
  public void computo() {
    System.out.println("Ejecutando la tarea " + msg);
  }
}
```


Polimor... y Sobrecar...: Implementar tareas como extensión de Tarea

```
// Objeto quickSort (quickSort.java)
public class quickSort extends Tarea implements java.io.Serializable {
  public void computo() {
    System.out.println("......Realizando un QuickSort !!");
  }
}
```

```
// Objeto bAb (bAb.java)
public class bAb  extends Tarea implements java.io.Serializable {
  public void computo() {
    System.out.println("......Realizando un Branch and Bound !!");
  }
}
```

```
// Objeto Greedy (Greedy.java)
public class Greedy  extends Tarea implements java.io.Serializable {
  public void computo() {
    System.out.println("......Realizando un Greedy !!");
  }
}
```

Polimor... y Sobrecar...: Implementación Interfaz en el Server

```
public class Server extends UnicastRemoteObject implements InterfazTarea  {
        public static final void main(String[] args) {
                try {
                        Server runner = new Server();
                        Naming.rebind("jobs_server",runner);
                } catch (Exception e) {
                        System.exit(1);
                }
        }
        public Tarea doquickSort() {
                return (new quickSort());
        }
        public Tarea dobAb() {
                return (new bAb());
        }
        public Tarea doGreedy() {
                return (new Greedy());
        }
}
```


Polimor... y Sobrecar...: Codificar el Cliente

```
public class Client {
    InterfazTarea objTarea;
    public static final void main(String[] args) {
        Client miCliente = new Client();
        miCliente.miproceso();
    }
    public Client() {
        try { objTarea = (InterfazTarea)Naming.lookup("rmi://host/jobs_server");}
        catch(Exception e) { System.out.println("Error:  " + e.getMessage()); }
    }
    public void miproceso() {
        try {
            Tarea miTarea = objTarea.doquickSort();
            miTarea.computo();
            Tarea miTarea = objTarea.dobAb();
            miTarea.computo();
            Tarea miTarea = objTarea.doGreedy();
            miTarea.computo();
        }  catch (Exception e) { System.out.println("Error: " + e.getMessage()); }
    }
}
```

Códigos

P1) Medir tiempos en transferencia de datos

./Ejercicios_RMI/Tiempos_en_Transferencia_Datos/....

P2) Cliente descarga códigos de un servidor

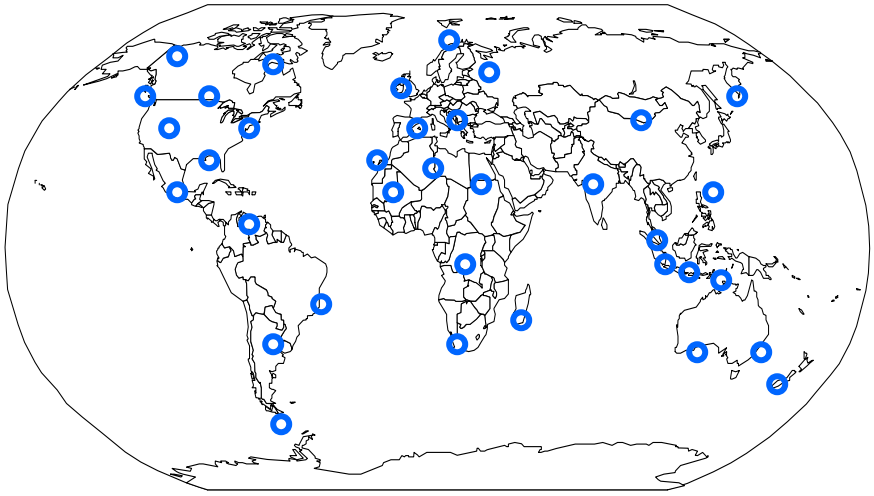
./Ejercicios_RMI/Carga_Remota_de_Codigo/Tareas_simple/....

P3) P2 + Polimorfismo y Sobrecarga

./Ejercicios_RMI/Carga_Remota_de_Codigo/Tareas_extendido/....

Mundo Java

- Máquina Virtual Java



Referencias bibliográficas

- Java RMI Specification: http://java.sun.com/rmi/
- Java Serialization: http://java.sun.com/serialization/
- Información on-line de Sun (*Remote Method Invocation Specification*). www.sun.com
- Orfali R. & Harkey D. Client/Server Programming with JAVA and CORBA. Ed. Wiley, segunda edición.
- D. Ayers, H.Bergsten, M. Bogovich, J. Diamond, M. Ferris, M. Fleury, A. Halberstadt, Paul Houle, P. Mohseni, A. Patzer, R. Phillips, S. Li, K. Vedati, M. Wilcox, and S. Zeiger, Professional Java Server Programming. Ed. Wrox Press, primera edición.
- Daniel J. Berg & J. Steven Fritzinger, Advanced Techniques for Java Developers, Ed. Wiley, primera edición. 1998.
- J. Jaworski. Java 1.2. al descubierto. Ed. Prentice Hall, primera edición. 1999.