Invocación de Métodos Remotos: prácticas de laboratorio

Coromoto León Hernández, Gara Miranda Valladares
Dpto. de Estadística, Investigación Operativa y Computación
Universidad de La Laguna
c/Astrofísico Fco. Sánchez s/n - 38271 La Laguna – S/C de Tenerife
(cleon|gmiranda)@ull.es

Resumen

En este trabajo se presenta una propuesta de prácticas de laboratorio para el tema de Invocación de Métodos Remotos utilizando el lenguaje de programación Java, en el contexto de una asignatura de Programación de Sistemas Distribuidos. En primer lugar, se propone una práctica de iniciación al modo de uso y a continuación, la implementación de un servicio remoto que permita resolver problemas utilizando la técnica de Ramificación y Acotación, siempre y cuando el usuario especifique el problema utilizando un formato determinado.

1. Introducción

La asignatura optativa "Programación de Sistemas Distribuidos" de seis créditos se oferta en el tercer curso de la Ingeniería Técnica en Informática de Sistemas en la Universidad de La Laguna.

El contenido de la asignatura cuenta con temas de Comunicación entre Procesos, Paradigma Cliente/Servidor, Interfaces de Programación de Sockets (tanto orientados como no orientados a conexión) y Objetos Distribuidos (Llamadas a Procedimiento Remoto, Invocación de Métodos Remotos, CORBA) [1][6]. La Tabla 1 muestra el conjunto de prácticas de laboratorio que se han programado asociadas a estos temas.

Java [3] es el lenguaje de programación elegido para desarrollar los contenidos prácticos de la asignatura. Una de las principales razones de esta elección es la facilidad de aprendizaje del mismo si se conoce C++ y la gran cantidad de material docente disponible.

La programación imperativa precede a la programación orientada a objetos. En ella, un procedimiento o función es una estructura de control que proporciona la abstracción correspondiente a una acción. Una llamada a procedimiento convencional es una llamada a un procedimiento que reside en el mismo sistema que el que la invoca. En el modelo de llamada a procedimiento remoto, un proceso realiza una llamada a un procedimiento de otro proceso, que posiblemente reside en un sistema remoto. Desde su introducción a principios de los años 80, el modelo de llamadas a procedimiento remoto (*Remote Procedure Call* – RPC) ha sido muy utilizado en aplicaciones distribuidas. Sin embargo, puesto que el lenguaje elegido para realizar las prácticas de la asignatura es Java, en lugar de RPC se estudiará el paradigma de Invocación a Métodos Remotos (*Remote Method Invocation* – RMI) que es una implementación orientada a objetos del modelo RPC [4]. Java-RMI es una herramienta exclusiva para programas Java, aunque debido a su relativa simplicidad, constituye un buen comienzo para estudiantes que están empezando a utilizar objetos distribuidos.

En RMI, un *Servidor de Objetos* exporta un *Objeto Remoto* y lo registra en un *Servicio de Nombres*. El objeto proporciona métodos remotos que un *Cliente* puede invocar utilizando una sintaxis similar a las de invocación de métodos locales.

Práctica	**Semanas**
Creación de Hilos (*Threads*)	1
Sincronización de Hilos (*Threads*)	1
Direcciones IP y Nombres de dominio	1
Serialización de objetos	1
Comunicación no orientada a conexión	2
Comunicación orientada a conexión	2
Comunicación en grupos	1
Invocación de Métodos Remotos	2
CORBA	2
Servicio de resolución mediante Ramificación y Acotación	1
Total	14

Tabla 1. Propuesta de prácticas de laboratorio

El contenido del trabajo se estructura de la siguiente forma: en la sección 2 se describe el funcionamiento del paradigma de Invocación a Métodos Remotos, en la sección 3 se presenta el enunciado de las prácticas de laboratorio asociadas al tema y finalmente aparecen las conclusiones y los trabajos futuros.

2. Invocación a métodos remotos en Java

El mecanismo de *Invocación a Métodos Remotos* permite hacer algo que parece sencillo. Si se tiene acceso a un objeto en una máquina distinta, se podrá llamar a los métodos de ese objeto remoto. Esto implica que:

- De algún modo, los parámetros del método deben pasarse a la otra máquina.
- El objeto debe ser informado para que ejecute el método.
- El valor obtenido debe ser devuelto.

Las clases que proporciona Java-RMI permiten manipular de forma transparente todos estos detalles. Considérense las siguientes definiciones:

- *Objeto Cliente*: objeto cuyos métodos efectúan la llamada remota.
- *Objeto Remoto*: objeto localizado en el servidor de Objetos.

La definición de un objeto se utiliza para una única llamada a un método. Es posible invertir los papeles en cualquier punto del camino. Por tanto, el servidor de una llamada previa puede convertirse en cliente cuando llame a un método remoto de un objeto que reside en otra máquina.

Cuando un *Objeto Cliente* quiere invocar a un método de un objeto remoto, llama a un método ordinario que está encapsulado en un objeto denominado *registro* (*stub*). El objeto *stub* reside en la máquina del cliente, no en el servidor. El *stub* empaqueta como un bloque de bytes los parámetros utilizados en el método remoto (usando el mecanismo de serialización). El objetivo de esta operación es convertir los parámetros a un formato susceptible de ser enviado de una JVM (*Java Virtual Machine*) a otra. El objeto *stub* en el *Cliente* construye un bloque con la siguiente información y la envía al *Servidor*:

- Una descripción del método que será llamado.
- Los parámetros codificados.

En el *Servidor* el objeto receptor realiza las siguientes acciones por cada método remoto invocado:

- Decodifica los parámetros.
- Localiza al objeto que ha sido llamado.
- Encuentra el método requerido.
- Captura y ordena los valores a devolver, o la excepción de la llamada.
- Envía un paquete con los datos del resultado codificados al *stub* del Cliente.

El *stub* del *Cliente* decodifica el valor de retorno, o la excepción, procedente del *Servidor*. Este valor se convierte en el valor de retorno de la llamada al *stub*. Si el método remoto lanza una excepción, el *stub* la propaga en el ambiente en que tuvo lugar la llamada. Este proceso aunque complejo es completamente transparente para el programador. Además, los diseñadores de objetos remotos en Java trataron de darles el mismo aspecto y funcionamiento que los locales.

3. Prácticas de laboratorio

Las prácticas de laboratorio asociadas al tema son dos: una inicial para afianzar los conceptos y mostrar la metodología de desarrollo y otra de aplicación de los mismos.

3.1. Aplicación simple: suma de dos arrays

El ejercicio consiste en implementar una aplicación Cliente/Servidor usando Java-RMI en la cual el servicio que ofrece el Objeto Remoto es un método que suma los elementos de dos arrays.

En el programa *Servidor* se ha de tener una instancia de un *Objeto Remoto*. El programa *Cliente* ha de proporcionar dos arrays de enteros e invocar al método remoto para que realice la suma y devuelva el array resultado [5].

Se han de implementar las siguientes clases:

- `Arith.java` (definición de la interfaz)
- `ArithImpl.java` (implementación de la interfaz)
- `ArithServer.java` (implementación del servidor)
- `ArithClient.java` (implementación del cliente)

El programa cliente tiene que manipular ciertos objetos, pero no tiene copia de ellos puesto que residen en el servidor. Sin embargo, como tiene que saber cómo trabajar con ellos, su forma de funcionamiento la encuentra en una interfaz que es compartida por el cliente y el servidor y que reside en ambas máquinas.

Todas las interfaces de *Objetos Remotos* deben derivar de la interfaz `Remote` que está definida en el paquete `java.rmi`. Todos los métodos declarados en estas interfaces pueden lanzar una `RemoteException` puesto que siempre es posible que una llamada remota falle. Así pues, Java obliga a capturar una `RemoteException` en cada llamada a un método remoto, además de tener que especificar una acción adecuada en el caso de que dicha llamada no se produzca.

El servidor debe implementar la clase que indica cómo funcionan los métodos de la interfaz remota. Para que esta clase sea un servidor de objetos remotos hay que hacer que extienda a la clase `UnicastRemoteObject`, que es una clase concreta del paquete `java.rmi.server` que hace que los objetos sean accesibles remotamente.

Para acceder a un *Objeto Remoto* que se encuentra en un servidor, el cliente necesita un objeto `stub` local. ¿Cómo puede un cliente hacerse con un objeto de este tipo? La librería RMI de Sun ofrece un *Servicio de Registro de Nombres* para localizar al objeto servidor. El programa servidor registra objetos en ese servicio, y el cliente obtiene los `stub` a partir de esos objetos. Para registrar un objeto se debe suministrar al servicio de registro una referencia al objeto y un nombre (que debería ser único):

```
Naming.bind("nombre", obj);
```

El cliente obtiene un `stub` para acceder a un *Objeto Remoto* especificando el nombre del servidor y del objeto de la siguiente forma:

```
... =(TypeCast) Naming.lookup(
   "rmi://yourserver.com/nombre");
```

Los programas cliente que usen RMI deben cargar un administrador de seguridad para controlar las actividades de los `stubs` que se cargan dinámicamente.

`RMISecurityManager` es un administrador de seguridad que se instala con la siguiente instrucción:

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

El programa cliente obtiene la referencia a un objeto `Arith` que se ha dado de alta en el *Registro de Nombres* RMI e invoca al método `add()` con dos arrays. Por defecto, `RMISecurityManager` prohíbe a todo el código del programa establecer conexiones de red. Pero son necesarias para acceder al registro RMI y para contactar con los objetos del servidor. Para ello se ha de proporcionar un archivo de política de seguridad e indicarle al administrador de seguridad que lea de ese archivo:

```
System.setProperty(
   "java.security.policy",
   "client.policy");
```

Los pasos a seguir para poner a funcionar la aplicación son los siguientes:

- Compilar los ficheros fuente de las clases interfaz, implementación de la interfaz, cliente y servidor:
  ```
  javac Arith*.java
  ```
- Compilar con `rmic` la clase de implementación para obtener el `stub`:
  ```
  rmic -v1.2 ArithImpl
  ```
- Inicializar el Registro de Nombre RMI con:
  ```
  rmiregistry &
  ```
- Ejecutar el servidor en una terminal:
  ```
  java ArithServer
  ```
- Ejecutar al cliente en otra terminal:
  ```
  java ArithClient
  ```

Asegurarse de que el fichero `client.policy` está en el directorio actual.

Además de la implementación anterior se proponen los siguientes ejercicios:

- Añada código al método `add()` de la clase `ArithImpl` de forma que haya un retardo de cinco segundos antes de que el método termine. Esto tiene el efecto de alargar de manera artificial la latencia de cada

invocación del método. Compile y arranque el servidor. En pantallas separadas arranque dos o más clientes. Observe la secuencia de eventos en las pantallas. ¿Se puede saber si el servidor de objetos ejecuta las llamadas a los métodos concurrente o iterativamente? Explíquelo.

- Implemente una aplicación Cliente/Servidor, utilizando el API de `Sockets`, en la cual el cliente envíe dos arrays de enteros al servidor, éste los sume y devuelva el array resultado al cliente. Utilizar la clase serializable `DataObj` como interfaz entre el cliente y el servidor. Implementar el servidor en una clase `ArithServer`. El servidor será un servidor simple que espera por una conexión. Cuando la conexión se establezca, leerá los objetos que le envía el cliente. Realizará la suma y devolverá el objeto resultado. Implementar el cliente en una clase `ArithClient`. El cliente será un cliente simple. Ha de definir dos objetos arrays con los que realizará su solicitud al servidor. Cuando recibe el resultado lo muestra por la pantalla.
- Utilice el método `currentTimeMillis()` de la clase `System` para establecer un cronómetro en los clientes de los ejercicios de Sockets y RMI. ¿Qué se puede decir acerca del tiempo que consume cada una de las implementaciones?

3.2. Aplicación compleja: servicio de resolución de problemas mediante ramificación y acotación

La técnica de Ramificación y Acotación es un método general que permite resolver un amplio rango de problemas de optimización combinatoria. Este tipo de paradigma pertenece a los denominados métodos enumerativos que se basan en la exploración del espacio de soluciones completo.

La técnica de Ramificación y Acotación consiste en explorar un grafo, normalmente acíclico o un árbol, buscando la solución óptima de un problema dado. En cada nodo, se calcula una cota del posible valor de aquellas soluciones que pudieran encontrarse más adelante en el grafo. Si la cota muestra que cualquiera de estas soluciones va a ser necesariamente peor que la mejor solución encontrada hasta el momento no se necesita seguir explorando esta parte del grafo. La Ramificación y Acotación utiliza cálculos auxiliares para decidir en cada momento qué nodo debe explorarse a continuación y una lista con prioridad para almacenar los nodos que se han generado y que se encuentran aún sin examinar. El orden en que se exploran los nodos puede variar según la implementación. Es típico usar una exploración en profundidad o una exploración en anchura, aunque también se puede realizar una exploración en la que se elija en primer lugar el nodo con mayor valor o realizar la decisión en base a otros criterios [2].

La implementación de un esquema genérico de resolución de problemas mediante esta técnica algorítmica conlleva el uso del siguiente conjunto de *interfaces* Java:

- `Problem`: define la interfaz mínima necesaria para definir un problema. Se debe sobrescribir el método `generateSubProblem()` que a partir del problema original debe generar el subproblema raíz del árbol de búsqueda.
- `SubProblem`: define la interfaz mínima necesaria para representar cada uno de los subproblemas en que se irá dividiendo el problema original. Los métodos que se han de implementar son:
  - `Solution solve(Problem p)`: devuelve la mejor solución encontrada para el subproblema actual.
  - `void branch(Problem sp, Solver s)`: genera cada uno de los subproblemas en los que el subproblema actual se ramifica, llamando al método `insert` del objeto s para cada uno de ellos.
  - `double lower_bound(Problem p)`: establece una cota inferior del valor de las soluciones que se conseguirán a partir del subproblema.
  - `double upper_bound(Problem p)`: establece una cota superior del valor de la soluciones que se conseguirán a partir del subproblema.
- `Solution`: esta interfaz debe ser implementada por la clase que representa la solución al problema. No presenta ningún método.

Se considerará como caso de estudio el Problema de Mochila 0-1. Su formulación es la siguiente:

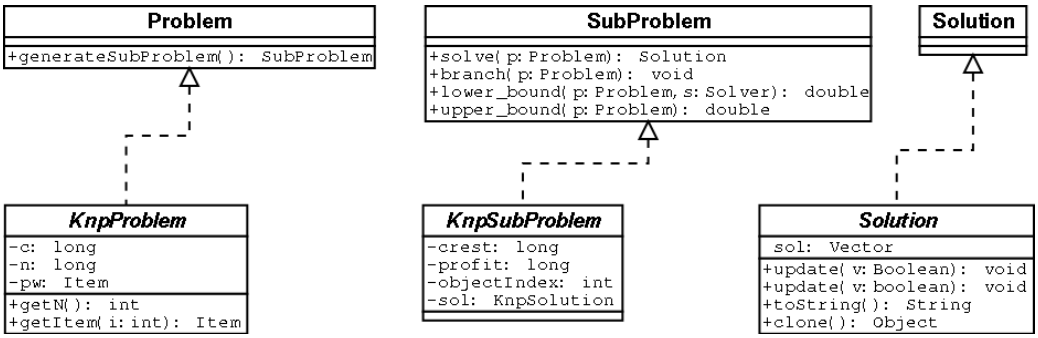

Figura 1. Diagrama UML para las clases que representan un problema

Dada una mochila de capacidad *C* y *N* objetos con pesos p_i y beneficios b_i para i = 1,...,N, se trata de guardar en la mochila aquellos objetos que maximicen el beneficio y no sobrepasen la capacidad. Se crean clases que implementan a las interfaces `Problem`, `SubProblem` y `Solution` para este problema concreto. La Figura 1 muestra el diagrama UML del conjunto de clases que componen el programa.

La clase `KnpProblem` contiene la definición de la capacidad, el número de objetos y el vector con los pesos y los beneficios. Es la que se muestra en la Figura 2.

La definición de la solución como un vector de booleanos que indican si se incluye o no un objeto se hace en la clase `KnpSolution` que aparece en la Figura 3.

```
public class KnpProblem
          implements Problem {
  private long c;
  private long n;
  private Item [] pw;
  ...
  public SubProblem
    generateSubProblem(){
    KnpSubProblem sp = new
    KnpSubProblem(c, 0, 0,
          new KnpSolution());
    return sp;
  }
}
```

Figura 2. Definición de la clase del problema

La clase `KnpSubProblem` contiene la definición de los subproblemas. Su implementación es la que aparece en al Figura 4. Se trabaja con: la capacidad restante, el beneficio acumulado, el objeto en estudio y la solución hasta el momento (esto es, los objetos que se han insertado o no hasta llegar al que está en estudio). Nótese que implementa todos los métodos de la interfaz `SubProblem`.

```
import java.util.*;

public class KnpSolution
 implements Solution, Cloneable {
  Vector sol;

  KnpSolution(){
    sol = new Vector();
  }

  public void update(Boolean v){
    sol.add(v);
  }

  public void update(boolean v){
    sol.add(new Boolean(v));
  }

  public String toString(){
  ...
  }

  public Object clone(){
  ...
  }
```

Figura 3. Definición de la clase solución

```
public class KnpSubProblem implements SubProblem {
  private long crest, profit; //Capacidad restante y beneficio actual
  private int objIndex;       //objeto en estudio
  private KnpSolution sol;    //solución
  ...
  public double upper_bound(Problem p){
    KnpProblem pm = (KnpProblem)(p);
    long weigth = 0, pft = profit, lastObject = pm.getN();
    for (int i = objIndex; i < lastObject; i++){
      if ((weigth + pm.getItem(i).getWeigth()) <= crest){
        weigth += pm.getItem(i).getWeigth(); pft += pm.getItem(i).getProfit();
      } else break;
    }
    if (i < lastObject){return (pft + (pm.getItem(i).getProfit()*
                               (crest - weigth)/pm.getItem(i).getWeigth()));
    }else return pft;
  }

  public double lower_bound(Problem p){
    KnpProblem pm = (KnpProblem)(p);
    long weigth = 0, pft = profit, lastObject = pm.getN();
    for (int i=objIndex; ((i < lastObject) && (weigth < crest)); i++){
      if ((weigth + pm.getItem(i).getWeigth()) <= crest){
        weigth += pm.getItem(i).getWeigth(); pft += pm.getItem(i).getProfit();
    } }
    return pft;
  }

  public Solution solve(Problem p){
    KnpProblem pm = (KnpProblem)(p);
    long weigth = 0, lastObject = pm.getN();
    KnpSolution sol2 = (KnpSolution)sol.clone();
    for (int i= objIndex; ((i < lastObject) && (weigth < crest)); i++){
      if ((weigth + pm.getItem(i).getWeigth()) <= crest){
        weigth += pm.getItem(i).getWeigth(); sol2.update(true);
      } else{ sol2.update(false);}
    }
    for (; i < lastObject; i++)sol2.update(false);
    return sol2;
  }

  public void branch(Problem pr, Solver s) {
    KnpProblem pm = (KnpProblem)(pr);
    if (objIndex < pm.getN()){
      KnpSubProblem spNo = new KnpSubProblem(crest, objIndex + 1,
                                    profit, (KnpSolution)sol.clone());
      s.insert(spNo); spNo.sol.update(false);
      long cap = pm.getItem(objIndex).getWeigth();
      if (cap <= crest){
        KnpSubProblem spYes = new KnpSubProblem(crest-cap, objIndex+1,
                              profit+pm.getItem(objIndex).getProfit(),
                              (KnpSolution)sol.clone());
        spYes.sol.update(true); s.insert(spYes);
    } }
  }
}
```

Figura 4. Definición de la clase del subproblema

```
 1  import java.util.*;
    ...
 2  Stack spList;
    ...
 3  public Solution run(Problem pbm) {
 4    double bestActual = -1;
 5    SubProblem actualsp;
 6    Solution sol = null;
 7
 8    spList.push(pbm.generateSubProblem());
 9
10    while(!spList.empty()){
11      double upper;
12      actualsp = (SubProblem)spList.pop();
13      if ((upper = actualsp.upper_bound(pbm)) > bestActual){
14        double lower;
15        if ((lower = actualsp.lower_bound(pbm)) > bestActual){
16          bestActual = lower;
17          sol = actualsp.solve(pbm);
18        }
19        if (upper != lower)
20          actualsp.branch(pbm, this);
21        }
22    }
23    return sol;
24  }
25  public void insert(SubProblem sp) {
26    spList.push(sp);
27  }
```

Figura 5. Esquema genérico de Ramificación y Acotación

La Figura 5 muestra un esquema genérico para resolver problemas de maximización mediante la técnica de Ramificación y acotación. Se utiliza la clase `java.util.Stack` para definir la estructura `spList` en la que se va a almacenar el recorrido del árbol de búsqueda de la solución (línea 2). El método `run()` es el encargado de realizar la búsqueda (líneas 3-24). En primer lugar se inserta en la lista de subproblemas, `spList`, el problema original (línea 8). En el bucle principal, mientras la lista no esté vacía (línea 10), se extrae un problema de la lista (línea 12), se calcula la cota superior del mismo (línea 13) y si es mejor que la mejor solución hasta el momento, se ramifica invocando al método `branch()` (línea 20). Además, se calcula la cota inferior (línea 15) y si es mejor que la mejor solución hasta el momento se sustituye por la que se acaba de calcular. El método `insert()` (líneas 25-27) es utilizado por el método `branch()` para insertar en `spList` los nuevos problemas que se generen. Estos métodos están implementados en la clase `Solver`.

El uso conjunto de todas las clases se implementa a través de una clase `Main` en la que se puede especificar el siguiente ejemplo:

Capacidad = 104;
Número de items = 8;
Pesos = { 25, 35, 45, 5, 25, 3, 2, 2};
Beneficios = {350, 400, 450, 20, 70, 8, 5, 5};
Solución = {1,3,4,5,7,8}
Beneficio máximo = 900.

Los ejercicios que se proponen al estudiante son:

- Compilar y ejecutar el ejemplo del Problema de la Mochila.
- Utilice el método `currentTimeMillis()` de la clase `System` para establecer un cronómetro en la clase `Main` que permita conocer el tiempo que se consume en la

resolución del problema. ¿Cuánto tiempo consume la implementación local?

- Implementar una aplicación Cliente/Servidor usando Java RMI en la cual el servicio que ofrece el objeto remoto es la resolución mediante la técnica de Ramificación y Acotación de un problema especificado. En el cliente se localizará y preguntará por el objeto remoto que resuelve un problema dado. Esto se implementará con las siguientes clases:
  - `Solver` (definición de la interfaz)
  - `SolverImpl` (implementación de la interfaz)
  - `SolverServer` (implementación del servidor)
  - `SolverClient` (implementación del cliente)
- Utilice el método `currentTimeMillis()` de la clase `System` para establecer un cronómetro en el cliente. ¿Cuánto tiempo consume la implementación distribuida?
- ¿Qué conclusión puede dar sobre los dos tipos de implementación?

4. Conclusión

Se ha presentado un conjunto de prácticas de laboratorio para cubrir el tema de Invocación a Métodos Remotos en una asignatura con contenidos de Programación de Sistemas distribuidos. La principal novedad es la propuesta de implementación de un servicio que proporciona la posibilidad de resolver problemas utilizando la técnica de Ramificación y Acotación, siempre que el usuario especifique el problema utilizando un formato determinado. Este esquema es genérico y se puede extender a otras técnicas algorítmicas como Divide y Vencerás o Programación Dinámica.

Para la evaluación de la propuesta se ha realizado una encuesta a los alumnos. El número total de respuesta fue 24. Una de las preguntas planteadas fue: ¿Considera la práctica final 'Servicio de resolución mediante Ramificación y Acotación' más compleja que las anteriores? Si / No. Los resultados de la encuesta muestran que el 96% de los alumnos no la consideran más compleja que las anteriores. Otra de las preguntas de la encuesta pedía que se valorara el grado de interés por cada práctica en una escala que iba de nada interesante a muy interesante. La Tabla 2 muestra los resultados obtenidos sumando como porcentaje de interés todos lo que estaban por encima del interesante por razones de espacio.

Práctica	**%interés**
Creación de Hilos (*Threads*)	100
Sincronización de Hilos (*Threads*)	96
Direcciones IP y Nombres de dominio	92
Serialización de objetos	87
Comunicación no orientada a conexión	96
Comunicación orientada a conexión	100
Comunicación en grupos	87
Invocación de métodos remotos (*Java – RMI*)	96
CORBA (*Java-IDL*)	100
Servicio de resolución mediante Ramificación y Acotación	92

Tabla 2. Resultados de la encuesta sobre interés de las prácticas de laboratorio

5. Agradecimientos

Este trabajo se ha realizado en el marco del plan Nacional de I+D+I bajo el proyecto con número de referencia TIN2005-08818-C04-04. El trabajo de Gara Miranda Valladares ha sido financiado con la beca FPU-AP2004-2290.

Referencias

[1] G. Coulouris, G., Dollimore, J., Kindberg, T. *Sistemas Distribuidos: Conceptos yDiseño*, Addison-Wesley, 3ra. Edicion, 2001.

[2] Dorta, I., Dorta, P., León, C., Rojas, A. *Utilización de Software en la Docencia de Técnicas Algorítmicas*. JENUI'2001. Páginas 190-195.

[3] Java 2 Standard Edition (J2SE) http://java.sun.com/j2se/

[4] Liu, M. L. *Computación Distribuida. Fundamentos y Aplicaciones.* Addison-Wesley, 1rd. Edición, 2004.

[5] Mahmoud, Qusay H. *Distributed Programming with Java*. Manning, 1999.

[6] Tanenbaum, A, van Steen, M. *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.