

Introducción a esquemas de nombres, redes, clientes y servicios con Java

Luis Daniel Benavides Navarro
Escuela Colombiana de Ingeniería
Arquitectura Empresarial

28 de enero de 2020

1. Reconocimiento

Parte de los contenidos y códigos de este taller están basados en los contenidos de los tutoriales de Java que se encuentran en:

<https://docs.oracle.com/javase/tutorial/networking/index.html>.

2. Conceptos básicos de redes

Los programas que se comunican a través de internet utilizan generalmente dos protocolos: el Transmission Control Protocol (TCP) o el User Datagram Protocol (UDP). En java en general usted utiliza clases ya implementadas en el paquete `java.net`.

2.1. TCP

El Transmission Control Protocol (TCP) es un protocolo basado en conexión que provee una conexión confiable entre dos computadores. TCP en particular mantiene el orden de los paquetes de datos y garantiza que todos los datos se entreguen.

2.2. UDP

El User Datagram Protocol (UDP) es un protocolo que envía los datos en paquetes llamados datagramas, no provee garantía de entrega ni de orden de entrega. Este protocolo no está basado en conexión.

2.3. Que son los puertos

En general los computadores tiene una sola conexión a internet y todos los datos que llegan y salen utilizan esta conexión física. Sin embargo, el computador

puede tener múltiples aplicativos que utilizan la red. Para separar la información que es enviada a una aplicativo específico se asigno un número lógico a cada aplicación. Este número es denominado el puerto y , como veremos más adelante, es utilizado para enviar datos a aplicaciones específicas en computadores remotos.

Los protocolos de TCP y UDP utilizan estos puertos para enviar los datos que llegan a las aplicaciones correctas. Recuerde que cada aplicación que espera datos de la red se le asigna un puerto para que puede escuchar los datos que llegan a un puerto determinado.

Los puertos se representan con un entero de 16 bits y tienen un rango de 0 hasta 65.535. Los puertos de 0- 1023 están restringidos para aplicaciones específicas por ejemplo el 80 es para el servidor web.

2.4. Clases que soportan el trabajo con redes en Java

Algunas de las clases que utilizan TCP en Java son: URL, URLConnection, Socket y ServerSocket. Todas están en el paquete java.net.

Algunas de las clases que utilizan UDP en Java son: DatagramPacket, DatagramSocket y MulticastSocket . Todas están en el paquete java.net.

3. Trabajando con URLs

URL es la abreviación de Uniform Resource Locator, y es básicamente una dirección para localizar recursos en internet. Una idea clara de como son las URLs la encontramos en nuestro navegador de internet. Así, la forma general de una URL es la siguiente:

```
<protocolo>://<servidor>:<puerto>/<dirección del recurso en el servidor>
```

un ejemplo concreto es:

```
http://ldbn.escuelaing.edu.co:80/index.html
```

En java se puede crear un URL de varias maneras:

```
URL personalSite = new URL("http://ldbn.escuelaing.edu.co:80/");
```

Este código crea un objeto de tipo URL que lo asigna a la variable personalSite. También puede crear una URL relativa a otra de la siguiente manera:

```
URL misPublicaciones = new URL(personalSite, "publications_bib.html");
```

Todos los constructores de la URL lanzan excepciones MalformedURLException, por lo que es necesario colocarlos dentro de un bloque try-catch.

```
try {
    URL myURL = new URL(. . .)
} catch (MalformedURLException e) {
    e.printStackTrace();
}
```

```

1 import java.io.*;
2 import java.net.*;
3
4 public class URLReader {
5
6     public static void main(String[] args) throws Exception {
7         URL google = new URL("http://www.google.com/");
8
9         try (BufferedReader reader
10              = new BufferedReader(new InputStreamReader(google.openStream()))) {
11             String inputLine = null;
12             while ((inputLine = reader.readLine()) != null) {
13                 System.out.println(inputLine);
14             }
15         } catch (IOException x) {
16             System.err.println(x);
17         }
18     }
19 }
```

Figura 1: clase que lee datos de internet

3.1. Leyendo los valores de un objeto URL

El programador puede usar varios métodos para leer la información de un objeto URL: getProtocol, getAuthority, getHost, getPort, getPath, getQuery, getFile, getRef.

EJERCICIO 1

Escriba un programa en el cual usted cree un objeto URL e imprima en pantalla cada uno de los datos que retornan los 8 métodos de la sección anterior.

3.2. Leyendo páginas de internet

Para leer páginas de internet debe crear flujos de datos (streams) y leer como si lo hiciera del teclado. El ejemplo siguiente lee datos de internet y los presenta en la pantalla (fig. 1).

EJERCICIO 2

Escriba una aplicación browser que pregunte una dirección URL al usuario y que lea datos de esa dirección y que los almacene en un archivo con el nombre resultado.html.

Luego intente ver este archivo en el navegador.

4. Sockets (enchufes)

Los sockets son los puntos finales del enlace de comunicación entre dos programas ejecutándose en la red. Cada socket está vinculado a un puerto específico,

así la capa que implementa el protocolo TCP puede saber a qué aplicación enviar los mensajes. En general un servidor es un proceso que se ejecuta y tiene un socket, vinculado a un puerto, que está esperando solicitudes de clientes externos. Los sockets son una abstracción de más bajo nivel que las URLs y sirven para implementar protocolos de comunicación cliente-servidor.

El protocolo cliente servidor consiste en un programa cliente que hace solicitudes a un programa servidor que atiende dichas solicitudes.

Java provee dos clases para manejar la comunicación por medio de sockets: `Socket` y `ServerSocket`. Ambas clases se encuentran en el paquete `java.net`.

NOTA: Una idea clara para entender los sockets es imaginar que son los enchufes donde se conectan las aplicaciones para comunicarse.

4.1. Como usar los sockets desde el cliente

Vamos a utilizar sockets para crear un pequeño aplicativo cliente servidor. El aplicativo consiste en un cliente que envía mensajes y un servidor que responde con el mismo mensaje pero con una cadena “Respuesta:” al principio del mismo. El servidor también imprime en pantalla los mensajes que recibe.

Antes de ver el código del cliente es importante ver que para obtener una conexión se usa el código:

```
miSocket = new Socket("127.0.0.1", 35000);
```

donde “127.0.0.1” es el host local y 35000 es el puerto. Estas sentencias tienen que estar rodeadas de bloque try-catch, para capturar los errores de conexión.

Una vez tenga la conexión, puede obtener flujos (Streams) de entrada y salida utilizando

```
out = new PrintWriter(echoSocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
    echoSocket.getInputStream()));
```

Una vez tenga los streams, puede enviar solicitudes y recibir las respuestas. No olvide cerrar los sockets y los flujos. La figura 2 muestra el código del cliente.

4.2. Como utilizar los sockets desde el servidor

La siguiente parte consiste en implementar el servidor. El servidor escucha en un puerto y responde a las solicitudes de cada cliente.

La figura 3 tiene el código del servidor. Este servidor responde el mismo mensaje que recibe.

4.3. Ejercicios

4.3.1.

Escriba un servidor que reciba un número y responda el cuadrado de este número.

```

1 import java.io.*;
2 import java.net.*;
4
4 public class EchoClient {
5     public static void main(String[] args) throws IOException {
6
7         Socket echoSocket = null;
8         PrintWriter out = null;
9         BufferedReader in = null;
10
11     try {
12         echoSocket = new Socket("127.0.0.1", 35000);
13         out = new PrintWriter(echoSocket.getOutputStream(), true);
14         in = new BufferedReader(new InputStreamReader(
15             echoSocket.getInputStream()));
16     } catch (UnknownHostException e) {
17         System.err.println("Don't know about host!.");
18         System.exit(1);
19     } catch (IOException e) {
20         System.err.println("Couldn't get I/O for "
21                         + "the connection to: localhost.");
22         System.exit(1);
23     }
24
25     BufferedReader stdIn = new BufferedReader(
26                     new InputStreamReader(System.in));
27     String userInput;
28
29     while ((userInput = stdIn.readLine()) != null) {
30         out.println(userInput);
31         System.out.println("echo: " + in.readLine());
32     }
33
34     out.close();
35     in.close();
36     stdIn.close();
37     echoSocket.close();
38 }

```

Figura 2: clase cliente que envía datos y recibe respuestas

```

1 import java.net.*;
2 import java.io.*;
3
4 public class EchoServer {
5     public static void main(String[] args) throws IOException {
6
7         ServerSocket serverSocket = null;
8         try {
9             serverSocket = new ServerSocket(35000);
10        } catch (IOException e) {
11            System.err.println("Could not listen on port: 35000.");
12            System.exit(1);
13        }
14
15        Socket clientSocket = null;
16        try {
17            clientSocket = serverSocket.accept();
18        } catch (IOException e) {
19            System.err.println("Accept failed.");
20            System.exit(1);
21        }
22
23        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
24        BufferedReader in = new BufferedReader(
25            new InputStreamReader(
26                clientSocket.getInputStream()));
27        String inputLine, outputLine;
28
29        while ((inputLine = in.readLine()) != null) {
30            System.out.println("Mensaje:" + inputLine);
31            outputLine = "Respuesta" + inputLine ;
32            out.println(outputLine);
33            if (outputLine.equals("Respuestas: Bye."))
34                break;
35        }
36        out.close();
37        in.close();
38        clientSocket.close();
39        serverSocket.close();
40    }
41 }
```

Figura 3: clase servidor que regresa el mismo mensaje que lee

4.3.2.

Escriba un servidor que pueda recibir un número y responda con un operación sobre este número. Este servidor puede recibir un mensaje que empiece por “fun:”, si recibe este mensaje cambia la operación a las especificada. El servidor debe responder las funciones seno, coseno y tangente. Por defecto debe empezar calculando el coseno. Por ejemplo, si el primer número que recibe es 0, debe responder 1, si después recibe $\pi/2$ debe responder 0, si luego recibe “fun:sin” debe cambiar la operación actual a seno, es decir a a partir de ese momento debe calcular senos. Si enseguida recibe 0 debe responder 0.

4.4. Servidor web

El código 4 presenta un servidor web que atiende una solicitud. Implemente el servidor e intente conectarse desde el browser.

4.5. Ejercicios

4.5.1.

Escriba un servidor web que soporte múltiples solicitudes seguidas (no concurrentes). El servidor debe retornar todos los archivos solicitados, incluyendo páginas html e imágenes.

5. Datagramas

Los programas escritos en las secciones anteriores presentan ejemplos de aplicaciones que se conectan punto a punto con otras aplicaciones. Estos ejemplos usaban por debajo el protocolo TCP.

Esta sección muestra programas que se comunican sin importar si los mensajes enviados fueron o no recibidos, o en qué orden llegan. Esto, se implementa usando el protocolo UDP. La abstracción fundamental para hacer este tipo de programas es el datagrama y el `java.net.DatagramSocket`.

5.1. Datagramas

Un datagrama es un mensaje independiente autocontenido que es enviado a través de la red, y cuya llegada, tiempo de llegada y contenido no son garantizados.

Estos datagramas son útiles para implementar servicios cuyos mensajes no tienen un contenido del cual dependen procesos fundamentales. Por ejemplo usted quiere que la comunicación entre un avión y la torre de control sea inmediata y garantizada, sin embargo, si tiene una página que muestra el estado del tiempo en la playa, no le importa si el último mensaje es de hace 1 hora y de pronto no es tan exacto.

```

1 import java.net.*;
3 import java.io.*;

5 public class HttpServer {
7
    public static void main(String[] args) throws IOException {
9
        ServerSocket serverSocket = null;
11       try {
11           serverSocket = new ServerSocket(35000);
13       } catch (IOException e) {
13           System.err.println("Could not listen on port: 35000.");
13           System.exit(1);
15       }
17
        Socket clientSocket = null;
18       try {
18           System.out.println("Listo para recibir ...");
19           clientSocket = serverSocket.accept();
21       } catch (IOException e) {
21           System.err.println("Accept failed.");
23           System.exit(1);
24       }
25
        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
27        BufferedReader in = new BufferedReader(
27            new InputStreamReader(
28                clientSocket.getInputStream()));
29        String inputLine, outputLine;
30
31        while ((inputLine = in.readLine()) != null) {
32            System.out.println("Received: " + inputLine);
33            if (!in.ready()) {
34                break;
35            }
36        }
37        outputLine = "<!DOCTYPE html>" +
38            + "<html>" +
39            + "<head>" +
40            + "<meta charset=\"UTF-8\">" +
41            + "<title>Title of the document</title>\n" +
42            + "</head>" +
43            + "<body>" +
44            + "My Web Site" +
45            + "</body>" +
46            + "</html>" + inputLine;
47        out.println(outputLine);
48
49        out.close();
50        in.close();
51        clientSocket.close();
52        serverSocket.close();
53    }
54}

```

Figura 4: clase que implementa un servidor web de un request

En esta sección vamos a construir un servidor que reporta la hora cuando recibe un mensaje que le solicita este servicio. Igualmente construiremos un cliente que pide el servicio.

La figura 5 implementa un servidor de datagramas. El servidor primero crea un objeto de tipo Datagramsocket y lo asocia al puerto 45000. Después, en el método startServer crea un buffer de 256 bytes que es usado para crear un DatagramPacket con este tamaño. Una vez se tiene el paquete creado se le dice que espere por un paquete,

```
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
```

el servidor espera a recibir un mensaje, y una vez lo recibe, lee la información de dirección ip y puerto de origen. Con esta información crea un paquete de respuesta mensaje de respuesta y responde al cliente.

La figura 6 implementa un cliente de datagramas. Este cliente crea un socket de datagramas pegado a un puerto, luego crea un paquete de salida y envía el datagrama al cliente solicitado. Luego espera por la respuesta del servidor. Observe que si no tiene respuesta el cliente se queda esperando para siempre. Si necesita cancelar la espera, puede hacer un pool de hilos, colocar la actividad en un hilo del pool, y asignarle un tiempo máximo de espera al pool de hilos.

5.2. Ejercicios

5.2.1.

Utilizando Datagramas escriba un programa que se conecte a un servidor que responde la hora actual en el servidor. El programa debe actualizar la hora cada 5 segundos según los datos del servidor. Si una hora no es recibida debe mantener la hora que tenía. Para la prueba se apagará el servidor y después de unos segundos se reactivará. El cliente debe seguir funcionando y actualizarse cuando el servidor este nuevamente funcionando.

6. Invocación remota de métodos: RMI

El sistema RMI (Remote method invocation) permite a un programa corriendo en un máquina virtual de Java llamar los métodos de objetos que están corriendo en otra máquina virtual de Java. Es decir, RMI permite la comunicación, utilizando un modelo orientado a objetos, entre dos aplicaciones Java.

6.1. Modelo general de comunicación

El modelo RMI busca implementar un modelo de objetos distribuidos con una semántica clara, simple y cercana a la semántica de objetos propuesta en el lenguaje de programación Java. Por esto utiliza las abstracciones de objeto y método como eje fundamental del modelo. Así, en un aplicación distribuida

```

2 import java.io.IOException;
3 import java.net.DatagramPacket;
4 import java.net.DatagramSocket;
5 import java.net.InetAddress;
6
7 import java.net.SocketException;
8 import java.util.Date;
9 import java.util.logging.Level;
10 import java.util.logging.Logger;
11
12 public class DatagramTimeServer {
13
14     DatagramSocket socket;
15
16     public DatagramTimeServer() {
17         try {
18             socket = new DatagramSocket(4445);
19         } catch (SocketException ex) {
20             Logger.getLogger(DatagramTimeServer.class.getName()).log(Level.SEVERE, null, ex);
21         }
22     }
23
24     public void startServer() {
25         byte[] buf = new byte[256];
26         try {
27
28             DatagramPacket packet = new DatagramPacket(buf, buf.length);
29             socket.receive(packet);
30
31             String dString = new Date().toString();
32             buf = dString.getBytes();
33             InetAddress address = packet.getAddress();
34             int port = packet.getPort();
35             packet = new DatagramPacket(buf, buf.length, address, port);
36             socket.send(packet);
37
38         } catch (IOException ex) {
39             Logger.getLogger(DatagramTimeServer.class.getName()).log(Level.SEVERE, null, ex);
40         }
41         socket.close();
42     }
43
44     public static void main(String[] args){
45         DatagramTimeServer ds = new DatagramTimeServer();
46         ds.startServer();
47     }
48 }
```

Figura 5: clase que implementa un servidor de datagramas

```

1 import java.io.IOException;
2 import java.net.DatagramPacket;
3 import java.net.DatagramSocket;
4 import java.net.InetAddress;
5 import java.net.SocketException;
6 import java.net.UnknownHostException;
7 import java.util.logging.Level;
8 import java.util.logging.Logger;
9
10 public class DatagramTimeClient {
11
12     public static void main(String[] args) {
13         byte[] sendBuf = new byte[256];
14         try {
15             DatagramSocket socket = new DatagramSocket();
16             byte[] buf = new byte[256];
17             InetAddress address = InetAddress.getByName("127.0.0.1");
18
19             DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 4445);
20             socket.send(packet);
21
22             packet = new DatagramPacket(buf, buf.length);
23             socket.receive(packet);
24             String received = new String(packet.getData(), 0, packet.getLength());
25             System.out.println("Date: " + received);
26             } catch (SocketException ex) {
27                 Logger.getLogger(DatagramTimeClient.class.getName()).log(Level.SEVERE, null, ex);
28             } catch (UnknownHostException ex) {
29                 Logger.getLogger(DatagramTimeClient.class.getName()).log(Level.SEVERE, null, ex);
30             } catch (IOException ex) {
31                 Logger.getLogger(DatagramTimeClient.class.getName()).log(Level.SEVERE, null, ex);
32             }
33         }
34     }
35 }
```

Figura 6: clase que implementa un cliente de datagramas

típica el servidor crea uno o varios objetos que se hacen disponibles para atender llamados remotos. Por su parte, el cliente localiza estos objetos, obtiene referencias remotas a ellos y llama sus métodos. Para lograr esto la aplicación necesita soportar los siguientes mecanismos:

- Mecanismos de localización de objetos remotos: para hacer un llamado remoto un cliente necesita saber la referencia remota de un objeto. Hay muchos mecanismos para obtener esta referencia, por ejemplo podría recibir un e-mail, un mensaje de texto, o incluso recibirla por teléfono. El mecanismo no es importante, lo importante es tener la referencia. Sin embargo, RMI provee un servicio de nombres (rmiregistry) que permite que el servidor publique sus objetos asociándolos con un nombre, y permita también que el cliente obtenga la referencia a un objeto remoto por medio de dicho nombre. Otra forma de recibir referencias remotas es que sean pasadas como parámetros o valores de retorno cuando se invoca un método.

IMPORTANTE: Observe que el mecanismo de nombrado permite desacoplar las implementaciones del cliente y el servidor. Es decir, el cliente y el servidor ya no tienen que conocerse, el cliente solo está interesado que alguien le suministre el servicio asociado a un nombre específico.

- Mecanismo de comunicación: Este mecanismo es el que me permite hacer la comunicación remota. En la siguiente sección se presenta algún detalle técnico de este mecanismo.
- Mecanismo de carga de definiciones de clases que son pasadas como referencias o como valores de retorno: Este mecanismo de cargue dinámico de bytecode es de vital importancia en el modelo RMI. Lo que permite es que si el cliente o el servidor no tiene la definición de una clase específica, la pueden solicitar remotamente para que estas sea transferida. Por supuesto para que esto pueda realizarse las definiciones de clases tienen que estar disponibles en algún lugar conocido y accesible tanto para el cliente como para el servidor

La figura 7 muestra un escenario de comunicación estándar, donde el cliente usa el rmiregistry para localizar un objeto remoto, luego llama un método en dicho objeto, el servidor descarga definiciones de clase si alguno de los parámetros es de un tipo que el servidor no conoce, el servidor retorna un valor, y finalmente si el cliente no conoce el tipo de retorno tiene la opción de descargar la definición de clase.

6.2. Stubs y Skeletons

RMI utiliza un mecanismo basado en stubs y skeletons para implementar la comunicación entre objetos remotos. El mecanismo tiene un funcionamiento básico en el cual el cliente invoca un método en el stub (que es un objeto local), y es este el encargado de hacer la invocación del método en el objeto remoto,

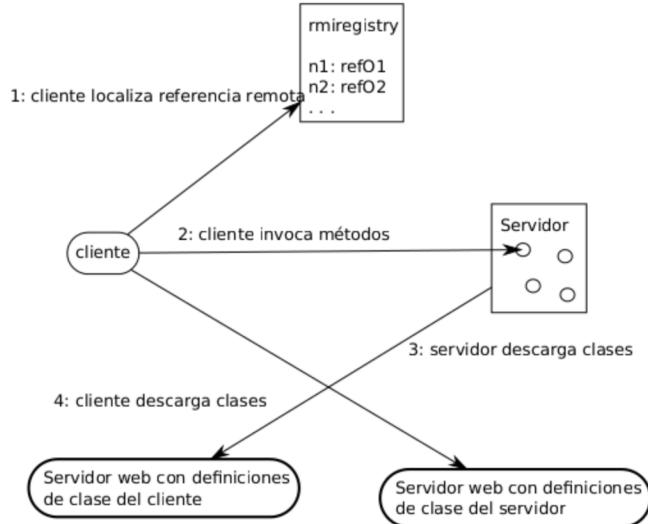


Figura 7: Modelo de comunicación RMI

i.e., oculta la complejidad de la comunicación remota. Este stub también es el encargado de serializar (preparar para transmitirlos) los parámetros que se envían al objeto remoto. Igualmente el stub se encarga de recibir la respuesta del llamado remoto y deserializarla para que pueda ser manejada por los objetos locales. La función del skeleton es muy similar a la del stub pero del lado del servidor. El skeleton espera por el llamado remoto, recibe los parámetros, realiza el llamado al método necesario y retorna el valor que regresa el método.

Observe que, aunque el stub como el skeleton se encargan de las complejidades de la comunicación, ambos objetos son solo proxies que son utilizados por el cliente y el servidor para llamar métodos reales sobre objetos locales.

Aunque el conocer este funcionamiento es muy útil, usted verá que la plataforma RMI oculta al programador estos detalles de bajo nivel, y la programación es totalmente transparente a este modelo.

6.3. Ejemplo

Vamos a implementar un servidor echo que retorna el mismo mensaje que recibe, pero con la etiqueta adicional “desde el servidor:”. Veamos primero la implementación del servidor.

6.3.1. Implementación del servidor

Primero tengo que declarar un interfaz remota, esta interfaz describe los servicios que prestará el objeto remoto. solo los métodos que se declaran en estas interfaces son los que pueden ser llamados remotamente. Es decir, esta interface

```

1 package rmiexample;
2
3 import java.rmi.Remote;
4 import java.rmi.RemoteException;
5
6 public interface EchoServer extends Remote {
7     public String echo(String cadena) throws RemoteException;
8 }
9

```

Figura 8: Interface que extiende la interface *Remote*

define el contrato de servicios remotos que presta un objeto. La interface extiende la interface *Remote* que es una interface de marcación, es decir que no define métodos, y que solo indica que será una interface de métodos que se pueden llamar remotamente. La única condición de los métodos definidos en una interface de tipo *Remote* es que deben lanzar la excepción *RemoteException*. La figura 8 muestra el código de la interface *EchoServer* que extiende la interface *Remote*. La interface define un método *echo* que recibe un *String* como parámetro y retorna un objeto de tipo *String* como respuesta.

Ahora que ya definimos los métodos que se pueden llamar remotamente debemos definir una clase que implemente estos métodos. La figura 9 muestra el código de la clase *EchoServerImpl* que implementa la interface *EchoServer* definida anteriormente. Esta clase implementa el método *echo* y adicionalmente define un constructor que realiza la publicación del objeto remoto en el servicio de referenciación por nombres (*rmiregistry*) correspondiente.

6.3.2. Implementación del cliente

El último paso en la implementación es escribir el cliente que se conectará utilizando RMI. La figura 10 muestra el código de la clase que implementa el cliente RMI. Esta clase primero carga un administrador de seguridad, luego se conecta al rmiregistry para solicitar la ubicación de un servicio utilizando el nombre, y finalmente invoca el método sobre el objeto remoto. Estudie el código y revise la documentación de las clases y métodos utilizados.

6.3.3. ¿Cómo ejecutar el software?

La ejecución de los programas RMI es un poco complicada porque hay que tener en cuenta las consideraciones de seguridad y de arquitectura de la aplicación. Para ejecutarla considere estos dos aspectos primero:

- **Class Path.** El class path es el conjunto de directorios donde se encuentran las clases que necesita su programa. Al invocar la máquina virtual de Java se puede pasar un parámetro indicándole dónde buscar las clases. Este parámetro se le indica a la máquina virtual usando “-cp” y adicionando en seguida los directorios del class path, donde la máquina virtual buscará

```

1 package rmiexample;
2
3 import java.rmi.RemoteException;
4 import java.rmi.registry.LocateRegistry;
5 import java.rmi.registry.Registry;
6 import java.rmi.server.UnicastRemoteObject;
7
8 public class EchoServerImpl implements EchoServer{
9
10    public EchoServerImpl(String ipRMIClient, int puertoRMIClient, String nombreDePublicacion){
11        if (System.getSecurityManager() == null) {
12            System.setSecurityManager(new SecurityManager());
13        }
14        try {
15            EchoServer echoServer =
16                (EchoServer) UnicastRemoteObject.exportObject(this,0);
17            Registry registry = LocateRegistry.getRegistry(ipRMIClient, puertoRMIClient);
18            registry.rebind(nombreDePublicacion, echoServer);
19            System.out.println("Echo server ready...");
20        } catch (Exception e) {
21            System.err.println("Echo server exception:");
22            e.printStackTrace();
23        }
24    }
25
26    public String echo(String cadena) throws RemoteException {
27        return "desde el servidor: " + cadena;
28    }
29
30    public static void main(String[] args){
31        EchoServerImpl ec = new EchoServerImpl("127.0.0.1", 23000, "echoServer");
32    }
33}

```

Figura 9: Clase que implementa la interface *EchoServer*

```

1 package rmiexample;
2
3 import java.rmi.registry.LocateRegistry;
4 import java.rmi.registry.Registry;
5
6 /**
7  * 
8  * @author danielben
9  */
10 public class EchoClient {
11
12     public void ejecutaServicio(String ipRmiregistry, int puertoRmiRegistry,
13                             String nombreServicio) {
14         if (System.getSecurityManager() == null) {
15             System.setSecurityManager(new SecurityManager());
16         }
17         try {
18             Registry registry = LocateRegistry.getRegistry(ipRmiregistry, puertoRmiRegistry);
19             EchoServer echoServer = (EchoServer) registry.lookup(nombreServicio);
20             System.out.println(echoServer.echo("Hola como estas?"));
21         } catch (Exception e) {
22             System.err.println("Hay un problema:");
23             e.printStackTrace();
24         }
25     }
26
27     public static void main(String[] args){
28         EchoClient ec = new EchoClient();
29         ec.ejecutaServicio("127.0.0.1", 23000, "echoServer");
30     }
31 }
```

Figura 10: Clase que implementa el cliente que se conecta utilizando RMI

las classes de su programa. Adicionalmente, al ejecutar varios de los componentes de este taller debe tener en cuenta ejecutarlos desde la raíz del class path, en particular el registry (servicio de directorio que relaciona nombres con referencias de objetos).

- **Seguridad.** También necesita archivos policy de seguridad que determinan que acceso tienen los programas que se conectan. Este archivo se puede crear en una carpeta separada de las clases. Para la ejecución de este ejemplo utilizaremos un archivo de seguridad con le nombre *policy*, el contenido de este archivo debe ser:

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

Este archivo le da permisos a la máquina virtual para conectarse y aceptar conexiones en todos los puertos mayores a 1024. Una vez tenga los archivos de seguridad ya podrá ejecutar la aplicación.

Lo primero es iniciar el servidor de nombres donde se registrarán los objetos que prestan servicios remotos, este servicio se iniciará en el puerto 23000 y los debe ejecutar desde la raíz del classpath (es decir desde el directorio donde el registry puede encontrar las definiciones de clase):

```
rmiregistry 23000
```

Ahora para ejecutar el servidor debe ejecutar desde la consola el siguiente comando:

```
java -cp .  
    -Djava.rmi.server.codebase=file:/<pathToClasses>/  
    -Djava.security.policy=file:/<pathToPolicy>/policy  
    rmiexample.EchoServerImpl
```

Este comando invoca la máquina virtual de java con tres parámetros específicos. El primer parámetro define un classpath (cp) donde el programa busca las definiciones de clase. También define el codebase que es donde el sistema de RMI busca las definiciones de clases que necesita enviar por la red. Finalmente, antes de invocar la clase a ejecutar, define la ubicación del archivo de seguridad.

De manera similar para ejecutar el cliente debe ejecutar desde la consola el siguiente comando:

```
java -cp .  
    -Djava.rmi.server.codebase=file:/<pathToClasses>/  
    -Djava.security.policy=file:/<pathToPolicy>/policy  
    rmiexample.EchoClient
```

6.4. Ejercicios

6.4.1.

CHAT: Utilizando RMI, escriba un aplicativo que pueda conectarse a otro aplicativo del mismo tipo en un servidor remoto para comenzar un chat. El aplicativo debe solicitar una dirección IP y un puerto antes de conectarse con el cliente que se desea. Igualmente, debe solicitar un puerto antes de iniciar para que publique el objeto que recibe los llamados remotos en dicho puerto.