UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

GESTION DE RECURSOS DE RED MEDIANTE LA INTERFASE DE PROGRAMACION DE APLICACIONES java.net

INFORME DE SUFICIENCIA

PARA OPTAR EL TÍTULO PROFESIONAL DE:

INGENIERO ELECTRÓNICO

PRESENTADO POR:

PEDRO HUGO VALENCIA MORALES

PROMOCIÓN

2003-1

LIMA – PERÚ

2008

GESTIÓN DE RECURSOS DE RED MEDIANTE LA INTERFASE DE PROGRAMACIÓN DE APLICACIONES java.net

Dedico este trabajo

A mi esposa Mary, quien es mi amor y la que me motiva a ser mejor cada día,

A mi hijo Hugo Alejandro, quien es mi alegría y fuerza en mi actividad diaria,

A mi madre Maria, quien con su apoyo logro que alcanzara esta meta,

Y a mis hermanos Mauro y Margarita que son mis ejemplos de superación.

SUMARIO

El presente informe tiene como objetivo mostrar las posibilidades que brinda el lenguaje de programación java en la administración de recursos de una red de datos, convirtiéndose así en una herramienta alternativa para el desarrollo de sistemas de gestión en un proceso de comunicación.

Para lograr este objetivo, hemos dividido el informe en cinco capítulos donde en cada uno de ellos presentamos inicialmente un fundamento sobre el recurso de red a ser administrado, a continuación se describe las interfaces de programación de aplicaciones (API) que tiene la plataforma java para implementar un programa que maneje dicho recurso y finalmente las salidas que se obtienen al ejecutarlos.

Al final del informe, observaremos que los resultados obtenidos con los ejemplos mostrados nos hacer preveer la existencia de una herramienta adicional para el desarrollo de aplicaciones de gestión de red, sobre todo por ser independiente de la plataforma operativa en la cual se ejecute, característica propia de la plataforma java.

INDICE

Págs.

PRÓLOGO

La tecnología Java a través de su lenguaje de programación Java en los últimos años se ha convertido en una de las principales plataformas para el desarrollo de aplicaciones de todo alcance, desde aplicaciones estándares con interfase grafica de usuario tipo ventanas hasta aplicaciones móviles con interacción a diferentes recursos empresariales.

Una de las áreas que ha ido evolucionando en este lenguaje es el de proporcionar una interface de programación para que mediante una aplicación se pueda gestionar directamente algunos recurso de red. Aunque en este ámbito, el lenguaje C++ sigue siendo el más adecuado para este tipo de aplicaciones, la plataforma estándar de Java en su versión actual 1.6 ha incorporado algunos servicios como el de identificación de interfaces y la posibilidad de programar ciertos parámetros de red.

Desde algunos años atrás vengo desarrollando aplicaciones con la tecnología Java orientado a soluciones comerciales y de gestión. Debido a mi formación universitaria en electrónica y técnica en programación es que me nace la inquietud de presentar los recursos actuales que cuenta esta plataforma para el área de gestión de recursos de red y la implementación de aplicaciones sencillas que muestran el uso de estas librerías.

En el desarrollo de los capítulos siguientes, presento una descripción concreta de la interfase Java para el manejo de ciertos servicios de red y programas ejemplos que ilustren el modo de uso.

CAPITULO I
PROGRAMACIÓN EN RED

1.1. Introducción

En el presente capitulo presentaremos algunos fundamentos sobre programación en red, sobre todo a los relacionados con las aplicaciones en Java que hacen uso de los recursos de una red. Para ello, presentaremos los siguientes temas:

1. Programación en Red con Java
2. Fundamentos de Redes

1.2. Programación en red con Java

1.2.1. Carga de Applets desde la Red

Mediante exploradores web con capacidad para presentar Applets, estos exploradores decodifican la etiqueta <APPLET> del documento html y según sus atributos descargan el applet, sin importar donde este ubicado, a la maquina local para su ejecución.

Esto es un ejemplo de acceso a alto nivel a Internet desde el entorno de desarrollo de Java. El explorador hizo el trabajo de conectarse a la red y obtener los datos necesarios, activando la ejecución de applets desde cualquier parte donde sé encuentre.

1.2.2. Carga de Imágenes desde URL (URL: Uniform Resource locutor)

Otra interacción de alto nivel con la red desde una aplicación java consiste en referenciar a una imagen o recursos de red a trabes de un localizador de recursos uniforme URL, el cual mediante una determinada función del lenguaje java realiza el trabajo de conectarse a la red y acceder al recurso.

1.3. Fundamentos de Redes

Computadoras con acceso a Internet se comunican a las otras PC's mediante los protocolos TCP o UDP, tal como se ilustra en el siguiente diagrama.

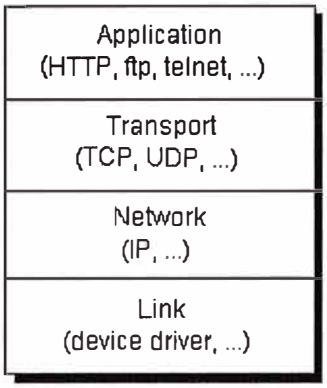


Fig. 1.1. Modelo TCP/IP

Cuando se escriben programas en Java que se comuniquen a través de la red, se esta programando en la capa de aplicación. Típicamente no se necesita conocer detalles sobre las capas TCP y UDP, en cambio se usan las clases del paquete java.net. Estas clases proveen comunicación de red independiente del sistema. Para decidir que clases de Java se puede utilizar se necesita conocer en que se diferencian TCP y UDP.

1.3.1. El Protocolo de Control de Transmisión (TCP: Transmision Control Protocol)

Cuando dos aplicaciones necesitan comunicarse entre si en forma confiable, se establece una conexión y se envían datos entre si sobre la conexión. Esto es análogo a realizar una llamada telefónica. TCP garantiza que los datos enviados desde un extremo de la conexión serán obtenidos al otro extremo y en el mismo orden en el cual se envía. De lo contrario se reporta un error.

TCP provee un canal punto a punto para aplicaciones que requieren comunicaciones confiables. Los protocolos HTTP, FTP y Telnet son ejemplos de aplicaciones que requieren un canal de comunicación confiable. El orden en el cual los datos son enviados y recibidos sobre la red son críticos para el éxito de estas aplicaciones. Cuando se usa HTTP para leer desde un URL, los datos deben ser recibidos en el cual fueron enviados, de lo contrario se recibirán archivos html recortados, archivos comprimidos corruptos o algún tipo de información invalida

Definición: TCP (Transmission Control Protocol) es un protocolo orientado a la conexión que provee un flujo de datos confiable entre dos computadoras.

1.3.2. El Protocolo de Datagramas de Usuario (UDP: User Datagram Protocol)

El protocolo UDP es usado para comunicaciones que no son garantizados entre dos aplicaciones en la red. UDP no es orientado a la conexión como TCP. Se envían paquetes independientes de datos llamados datagramas desde una aplicación a otra.

Este tipo de comunicaciones similar al de enviar una carta a través del servicio postal. El orden de la entrega no interesa y no es garantizado y cada mensaje es independiente del otro.

Definición: UDP (User Datagram Protocol) es un protocolo que envía paquetes independientes de datos llamados datagramas desde un computador a otro sin garantizar su llegada.

Nota: Muchos firewalls y routers han siso configurados para no aceptar paquetes UDP.

1.3.3. Puertos

En términos generales, una computadora tiene solo una conexión física a la red. Todos los datos destinados para una computadora particular llegan a través de dicha conexión. Sin embargo los datos están dirigidos a diferentes aplicaciones que se ejecutan en la misma computadora. ¿Cómo entonces la computadora sabe a que aplicación reenviar el dato?, la respuesta es a través de los puertos.

Los datos transmitidos sobre Internet son acompañados de información de direccionamiento que identifica a la computadora y el puerto al cual es destinado. La computadora es identificada por una dirección IP de 32 bits el cual se usa para entregar los datos a la computadora correcta. Los puertos están identificados por un número de 16 bits el cual TCP y UDP utilizan para entregar los datos a la aplicación correcta.

En una comunicación orientada a la conexión tal como realiza TCP, una aplicación servidora enlaza un socket a un número específico de puerto. Esto tiene el efecto de registrar el servidor con el sistema para la recepción de todos los datos destinados a ese puerto. Un cliente puede comunicarse con el servidor a través del puerto del servidor, como se ilustra a continuación:

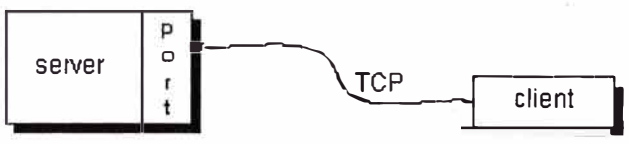


Fig. 1.2. Puerto del Servidor

Definición: Los protocolos TCP y UDP usan puertos para mapear entrada de datos a procesos particulares ejecutándose en un computador.

En comunicación basado en datagramas tales como UDP, los paquetes de datagrama contienen el numero de puertos de su destino y UDP en ruta el paquete a la aplicación apropiada tal como se ilustra en la figura siguiente>

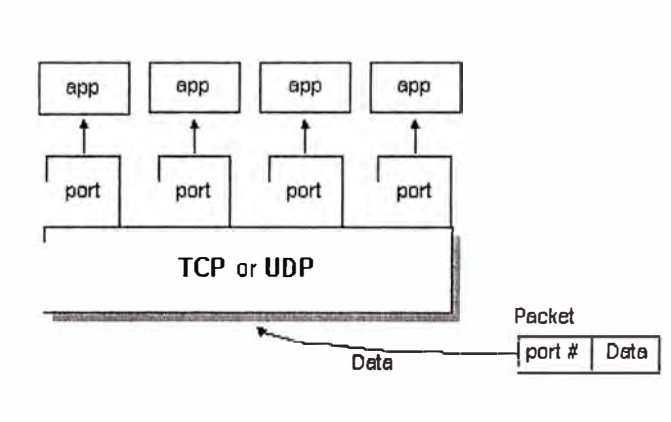


Fig. 1.3. Enrutamiento de Datagramas

El rango del número de puertos va desde 0 hasta 65,535 debido a que están representados por números de 16 bits. Los números de puerto comprendidos entre 0 y 1023 están restringidos ellos están reservados para ser usados por servicios conocidos tales como HTTP y FTP y otros servicios del sistema. Estos puertos son llamados puertos bien conocidos. Una aplicación no debe intentar enlazarse a uno de ellos.

1.3.4. Clases de Redes en el JDK
(JDK: Java Development Kit)

Los programas en java pueden usar TCP y UDP para comunicarse sobre Internet. Las clases URL, URLConnection, Socket y ServerSocket todos ellos usan TCP para comunicarse sobre la red. Las clases DatagramPacket, DatagramSocket y MulticastSocket usan UDP.

CAPITULO II
TRABAJANDO CON URL'S

2.1. Introducción.

URL es el acrónimo de Uniform Resource Locator. Es una referencia (dirección) a un recurso en Internet. El URL es suministrado al explorador web de tal manera que pueda ubicar los archivos en Internet en la misma forma que se provee una dirección en las cartas para que la oficina postal pueda ubicar la correspondencia.

Los programas java que interactúan con Internet también pueden usar URL's para hallar los recursos en Internet que desean acceder. Estos programas usan una clase llamada URL del paquete java.net para representar una dirección URL.

2.2. Que es una URL

(URL: Uniform Resource Locator)

Un URL tiene la forma de un texto que describe como hallar un recurso en Internet. Tiene dos componentes principales: el protocolo que se necesita para acceder al recurso y la ubicación del recurso.

El siguiente texto es un ejemplo de URL el cual direcciona el sitio Web de Java contenido en Sun Microsystems:

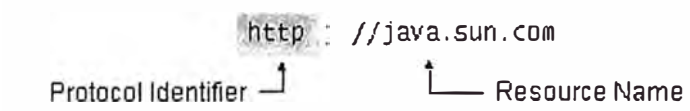


Fig. 2.1. URL

Como se observa en el diagrama, un URL tiene dos componentes principales:

- Identificador de protocolo
- Nombre del recurso

El identificador de protocolo indica el nombre del protocolo a ser usado para ubicar el recurso. El ejemplo utiliza HyperText Transfer Potocol (HTTP) típicamente usado para acceder a documentos hipertexto. Otros protocolos que podrían incluirse serian File Transfer Protocol (FTP), Gopher, File y News.

El nombre del recurso es la dirección completa para acceder al recurso. El formato del nombre del recurso depende del protocolo usado pero para varios protocolos, incluyendo HTTP, el nombre de recurso contiene uno o más de los componentes listados en la siguiente tabla:

Host Name	El nombre de la maquina donde reside el recurso
Filename	La ruta al archivo en la maquina
Port Number	El numero de puerto a conectarse (típicamente opcional)
Reference	Una referencia a un nombre de sección dentro de un recurso que usualmente identifica a una ubicación especifica dentro del archivo (típicamente opcional).

Para varios protocolos, el nombre de host y el de archivo son necesarios, mientras que el número de puerto y la referencia son opcionales.

2.3. Creación de una URL.

La forma más sencilla de crear un objeto URL es desde una cadena que represente en forma legible una dirección URL. Por ejemplo, el URL para el sitio Google tendrá la forma:

http://www.google.com/

En un programa Java, se pasa una cadena que contenga este texto para crear un objeto URL:

URL google = new URL("http://www.google.com/");

El objeto URL creado anteriormente representa un URL absoluto, el cual contiene toda la información necesaria para acceder al recurso indicado.

2.3.1. Creación de un URL relativo a otro

Un URL relativo contiene solo la información necesaria para alcanzar al recurso relativo a (o dentro del contexto) otro URL.

Las especificaciones relativas de URL son usadas comúnmente en archivos HTML. Por ejemplo, considere que se tiene un archivo HTML de nombre HomePage.html. Dentro de esta pagina hay enlaces a otras paginas, Presentacion.html y Ayuda.html las

cuales están en la misma maquina y carpeta que el archivo HomePage.html. Los enlaces a los archivos Presentacion.html y Ayuda.html desde HomePage.html pueden ser especificados como nombres de archivos, como en el siguiente ejemplo:

```
<a href="Presentacion.html">Presentacion</a>
<a href="Ayuda.html">Ayuda</a>
```

Estas direcciones URL son URL's relativos debido a que están especificados en forma relativa al archivo en el cual están direccionados, HomePage.html.

En un programa Java, se puede crear un objeto URL desde una especificación URL relativa. Por ejemplo, supongamos que se conocen dos URL's en el sitio HugoVM:

```
http://www.hugovm.com/pages/hugovm.net.html
http://www.hugovm.com/pages/hugovm.rsc.html
```

Se pueden crear objetos URL para estas páginas relativas al URL base común http://www,hugovm.com/pages/ tal como:

```
URL hugovm = new URL("http://www.hugovm.com/pages/");
URL hugovmnet = new URL(hugovm, "hugovm.net.html");
URL hugovmsrc = new URL(hugovm, "hugovmsrc.html");
```

Este código usa el constructor URL que permite crear un objeto URL desde otro objeto URL (la base) y una especificación URL relativa. La forma general del constructor es:

URL(URL *baseURL*, String *relativeURL*)

El primer argumento es un objeto URL que especifica la base del nuevo URL. El segundo argumento es un texto que especifica el resto del nombre del recurso relativo a la base. Si baseURL es nulo, el constructor trata a relativeURL como una especificación a un URL absoluto. Si relativeURL es una especificación a un URL absoluto el constructor ignora baseURL.

Este constructor también es útil para crear objetos URL para enlaces con nombre (también llamados referencias) dentro de un archivo. Por ejemplo, supongamos que el archivo hugovm.net.html tiene una referencia llamada FINAL al final del archivo. Se puede utilizar el constructor URL relativo para crear un objeto URL para esta referencia:

```
URL hugovmnet.Final = new URL(hugovmnet, "#FINAL");
```

2.3.2. Otros constructores URL.

La clase URL provee dos constructores adicionales para crear un objeto URL. Estos constructores son útiles cuando se trabaja con URL's tales como HTTP URL que tienen nombre del host, nombre del archivo, número de puerto y componentes referenciados en la parte del nombre del recurso del URL. También son útiles cunado no

se tiene un texto conteniendo la especificación URL completa, pero se conocen varios componentes del URL.

Por ejemplo, un primer constructor crea un objeto URL desde un protocolo, nombre de host y nombre de archivo tal como se muestra en el siguiente ejemplo:

new URL("http", "www.hugovm.com", "/pages/hugovm.net.html");

El cual es equivalente a:

new URL("http://www.hugovm.com/pages/hugovm.net.html");

Un segundo constructor agrega el número de puerto a la lista de argumentos usados en le constructor anterior:

new URL("http", "www.hugovm.com", 80, "/pages/hugovm.net.html");

El cual crea un objeto URL para el siguiente URL:

http://www.hugovm.com:80/pages/hugovm.net.html

Para obtener un texto conteniendo el URL completo usar el método toString() del objeto URL o el método toExternalForm().

2.3.3. Direcciones URL con caracteres especiales.

Algunas direcciones URL contienen caracteres especiales como por ejemplo un espacio:

http://hugovm.com/hola mundo/

Para hacer validos estos caracteres necesitan codificarse antes de pasarlo al constructor URL:

new URL("http://hugovm.com/hola%20mundo");

En el caso que se tengan varios caracteres especiales se puede usar el constructor de la clase java.net.URI que automáticamente codifica estos caracteres:

URI uri = new URI("http", "hugovm.com", "/hola mundo/","");

y luego convertir el URI a URL:

URL url = uri.toURL();

2.3.4. La excepción MalformedURLException

Los constructores anteriores lanzan una excepción MalformedURLExcepcion si los argumentos del constructor referencia a un protocolo nulo desconocido. Típicamente se puede capturar y manejar esta excepción insertándolo dentro del bloque try/catch:

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

2.4. Analizando un URL.

La clase URL proporciona varios métodos que permiten obtener de los objetos URL el protocolo, nombre de host, número de puerto y nombre de archivo de una URL. Se mencionan algunos de estos métodos:

Método	Descripción
getProtocol()	Devuelve el componente identificador de protocolo de la URL.
getHost()	Devuelve el componente nombre del host de la URL.
getPort()	Devuelve el componente número del puerto de la URL. Este método devuelve un entero que es el número de puerto. Si el puerto no está seleccionado, devuelve -1.
getFile()	Devuelve el componente nombre de fichero de la URL.
getRef()	Obtiene el componente referencia de la URL.

Recuerda que no todas las direcciones URL contienen estos componentes. La clase URL proporciona estos métodos porque las URL's de HTTP contienen estos componentes y quizás son las URL's más utilizadas. La clase URL está centrada de alguna forma sobre HTTP.

Se pueden utilizar estos métodos getXXX() para obtener información sobre la URL sin importar el constructor que se haya utilizado para crear el objeto URL.

La clase URL, junto con estos métodos accesores, libera de tener que analizar la URL de nuevo dando a cualquier cadena la especificación de una URL, y sólo creando un nuevo objeto URL y llamando a uno de sus métodos accesores para la información que se necesite. Este pequeño programa de ejemplo crea una URL partiendo de una especificación y luego utiliza los métodos accesores del objeto URL para analizar la URL.

```
import java.net.*;
import java.io.*;
class ParseURL {
   public static void main(String[] args) {
      URL aURL = null;
      try {
         aURL = new
        URL("http://java.sun.com:80/tutorial/intro.html#DOWNLOADING");
         System.out.println("protocol = " + aURL.getProtocol());
         System.out.println("host = " + aURL.getHost());
         System.out.println("filename = " + aURL.getFile());
         System.out.println("port = " + aURL.getPort());
         System.out.println("ref = " + aURL.getRef());
      } catch (MalformedURLException e) {
         System.out.println("MalformedURLException: " + e);
      }
   }
}
```

La salida del programa seria:

```
protocol = http
host = java.sun.com
filename = /tutorial/intro.html
port = 80
ref = DOWNLOADING
```

2.5. Lectura desde un URL.

Después de haber creado satisfactoriamente una URL, se puede llamar al método openStream() de la clase URL para obtener un canal desde el que poder leer el contenido de la URL. El método retorna un objeto java.io.InputStream por lo que se puede leer normalmente de la URL utilizando los métodos normales de InputStream. Los Canales de Entrada y Salida describen las clases de I/O proporcionadas por el entorno de desarrollo de Java y enseña cómo utilizarlas.

Leer desde una URL es tan sencillo como leer de un canal de entrada. El siguiente programa utiliza openStream() para obtener un stream de entrada a la URL

"http://www.yahoo.com/". Lee el contenido del canal de entrada y lo muestra en la pantalla.

```
import java.net.*;
import java.io.*;

class OpenStreamTest {
   public static void main(String[] args) {
      try {
         URL yahoo = new URL("http://www.yahoo.com/");
         DataInputStream dis = new DataInputStream(yahoo.openStream());
         String inputLine;

         while ((inputLine = dis.readLine()) != null) {
            System.out.println(inputLine);
         }
         dis.close();
      } catch (MalformedURLException me) {
         System.out.println("MalformedURLException: " + me);
      } catch (IOException ioe) {
         System.out.println("IOException: " + ioe);
      }
   }
}
```

Cuando se ejecuta el programa, se debería ver los comandos HTML y el contenido textual del fichero HTML localizado en "http://www.yahoo.com/" desplazándose por la ventana de comandos. O podrías ver el siguiente mensaje de error.

```
IOException: java.net.UnknownHostException: www.yahoo.com
```

El mensaje anterior indica que se podría tener seleccionado un proxy y por eso el programa no puede encontrar el servidor www.yahoo.com.

2.6. Conectándose a un URL.

Luego de crear satisfactoriamente una URL, se puede llamar al método openConnection() de la clase URL para conectar con ella. Cuando se haya conectado con una URL habrá inicializado un enlace de comunicación entre un programa Java y la URL a través de la red. Por ejemplo, se puede abrir una conexión con el motor de búsqueda de Yahoo con el código siguiente.

```
try {
   URL yahoo = new URL("http://www.yahoo.com/");
   yahoo.openConnection();
} catch (MalformedURLException e) {    // nueva URL() fallada
   ...
} catch (IOException e) {              // openConnection() fallada
   ...
}
```

Si es posible, el método openConnection() crea una nuevo objeto URLConnection (si no existe ninguno apropiado), lo inicializa, conecta con la URL y devuelve el objeto URLConnection. Si algo va mal -- por ejemplo, el servidor de Yahoo está apagado -- el método openConnection() lanza una IOException.

Ahora que se ha conectado satisfactoriamente con la URL se puede utilizar el objeto URLConnection para realizar algunas acciones como leer o escribir a través de la conexión.

2.7. Lectura y Escritura a través de un objeto URLConnection.

Si se ha utilizado satisfactoriamente openConnection() para inicializar comunicaciones con una URL, se tendrá una referencia a un objeto URLConnection. La clase URLConnection contiene muchos métodos que permiten comunicarse con la URL a través de la red. URLConnection es una clase centrada sobre HTTP -- muchos de sus métodos son útiles sólo cuando trabajan con URLs HTTP. Sin embargo, la mayoría de los protocolos URL permite leer y escribir desde una conexión.

2.7.1. Lectura desde un objeto URLConnection.

El siguiente programa realiza la misma función que el mostrado en Leer Directamente desde una URL. Sin embargo, mejor que abrir directamente un stream

desde la URL, este programa abre explícitamente una conexión con la URL, obtiene un stream de entrada sobre la conexión, y lee desde el stream de entrada.

```
import java.net.*;
import java.io.*;

class ConnectionTest {
   public static void main(String[] args) {
      try {
         URL yahoo = new URL("http://www.yahoo.com/");
         URLConnection yahooConnection = yahoo.openConnection();
         DataInputStream dis = new DataInputStream(yahooConnection.getInputStream());
         String inputLine;

         while ((inputLine = dis.readLine()) != null) {
            System.out.println(inputLine);
         }
         dis.close();
      } catch (MalformedURLException me) {
         System.out.println("MalformedURLException: " + me);
      } catch (IOException ioe) {
         System.out.println("IOException: " + ioe);
      }
   }
}
```

La salida de este programa debería ser idéntica a la salida del programa que abría directamente el stream desde la URL. Se puede utilizar cualquiera de estas dos formas para leer desde una URL. Sin embargo, algunas veces leer desde una URLConnection en vez de leer directamente desde una URL podría ser más útil ya que se puede utilizar el objeto URLConnection para otras tareas (como escribir sobre la conexión URL) al mismo tiempo.

De nuevo, si en vez de ver la salida del programa, se viera el siguiente mensaje error.

```
IOException: java.net.UnknownHostException: www.yahoo.com
```

Se podrías tener activado un proxy y el programa no podría encontrar el servidor de www.yahoo.com.

2.7.2. Escritura con un objeto URLConnection.

Muchas páginas HTML contienen forms -- campos de texto y otros objeto GUI que le permiten introducir datos en el servidor. Después de teclear la información requerida e iniciar la petición pulsando un botón, el navegador que se utiliza escribe los datos en la URL a través de la red. Después de que la otra parte de la conexión (normalmente un script cgi-bin) en el servidor de datos, los procesa, y le envía de vuelta una respuesta, normalmente en la forma de una nueva página HTML. Este escenario es el utilizado normalmente por los motores de búsqueda.

Muchos scripts cgi-bin utilizan el POST METHOD para leer los datos desde el cliente. Así, escribir sobre una URL frecuentemente es conocido como posting a URL. Los scripts del lado del servidor utilizan el método POST METHOD para leer desde su entrada estándar.

Algunos scripts cgi-bin del lado del servidor utilizan el método GET METHOD para leer sus datos. El método POST METHOD es más rápido haciendo que GET METHOD esté obsoleto porque es más versátil y no tiene limitaciones sobre los datos que pueden ser enviados a través de la conexión.

Los programas Java también pueden interactuar con los scripts cgi-bin del lado del servidor. Sólo deben poder escribir a una URL, así proporcionan los datos al servidor. Tu programa puede hacer esto siguiendo los siguientes pasos.

1. Crear una URL.
2. Abrir una conexión con la URL.
3. Obtener un stream de salida sobre la conexión. Este canal de entrada está conectado al stream de entrada estándar del script cgi-bin del servidor.
4. Escribir en el stream de salida.
5. Cerrar el stream de salida.

Hassan Schroeder, un miembro del equipo de Java, escribió un script cgi-bin, llamado backwards, y está disponible en la Web site de, java.sun.com. Puedes utilizar este script para probar el siguiente programa de ejemplo. Si por alguna razón no puedes obtenerlo de nuestra Web; puedes poner el script en cualquier lugar de la red, llamándolo backwards, y prueba el programa localmente.

El script de nuestra Web lee una cadena de la entrada estándar, invierte la cadena, y escribe el resultado en la salida estándar. El script requiere una entrada de la

siguiente forma: string=string_to_reverse, donde string_to_reverse es la cadena cuyos caracteres van a mostrarse en orden inverso.

Aquí tienes un programa de ejemplo que ejecuta el script backwards a través de la red utilizando un URLConnection.

```
import java.io.*;
import java.net.*;

public class ReverseTest {
   public static void main(String[] args) {
      try {
         if (args.length != 1) {
            System.err.println("Usage:  java ReverseTest string_to_reverse");
            System.exit(1);
         }
         String stringToReverse = URLEncoder.encode(args[0]);

         URL url = new URL("http://java.sun.com/cgi-bin/backwards");
         URLConnection connection = url.openConnection();

         PrintStream outStream = new PrintStream(connection.getOutputStream());
         outStream.println("string=" + stringToReverse);
         outStream.close();

         DataInputStream inStream = new DataInputStream(connection.getInputStream());
         String inputLine;
         while ((inputLine = inStream.readLine()) != null) {
            System.out.println(inputLine);
         }
         inStream.close();
      } catch (MalformedURLException me) {
         System.err.println("MalformedURLException: " + me);
      } catch (IOException ioe) {
         System.err.println("IOException: " + ioe);
      }
   }
}
```

Examinemos el programa y veamos como trabaja. Primero, el programa procesa los argumentos de la línea de comandos.

```
if (args.length != 1) {
   System.err.println("Usage:  java ReverseTest string_to_reverse");
   System.exit(1);
}
String stringToReverse = URLEncoder.encode(args[0]);
```

Estas líneas aseguran que el usuario proporciona uno y sólo un argumento de la línea de comandos del programa y lo codifica. El argumento de la línea de comandos es la cadena a invertir por el script cgi-bin backwards. El argumento de la línea de comandos podría tener espacios u otros caracteres no alfanuméricos.

Estos caracteres deben ser codificados porque podrían suceder varios procesos en la cadena en el lado del servidor. Esto se consigue mediante la clase URLEncoder.

Luego el programa crea el objeto URL -- la URL para el script backwards en java.sun.com.

```
URL url = new URL("http://java.sun.com/cgi-bin/backwards");
```

El programa crea una URLConnection y abre un stream de salida sobre esa conexión. El stream de salida está filtrado a través de un PrintStream.

```
URLConnection connection = url.openConnection();
PrintStream outStream = new PrintStream(connection.getOutputStream());
```

La segunda línea anterior llama al método getOutputStream() sobre la conexión. Si no URL no soporta salida, este método lanza una UnknownServiceException. Si la URL soporta salida, este método devuelve un stream de salida que está conectado al stream de entrada estándar de la URL en el lado del servidor -- la salida del cliente es la entrada del servidor.

Luego, el programa escribe la información requerida al stream de salida y cierra el stream.

```
outStream.println("string=" + stringToReverse);
outStream.close();
```

Esta línea escribe en el canal de salida utilizando el método println(). Como puedes ver, escribir datos a una URL es tan sencillo como escribir datos en un stream. Los datos escritos en el stream de salida en el lado del cliente son la entrada para el script backwards en el lado del servidor. El programa ReverseTest construye la entrada en la forma requerida por el script mediante la concatenación string= para codificar la cadena.

Frecuentemente, como en este ejemplo, cuando escribe en una URL está pasando información al script cgi-bin que lee la información que usted escribe, realiza alguna acción y luego envía la información de vuelta mediante la misma URL. Por lo que querrás leer desde la URL después de haber escrito en ella. El programa ReverseTest los hace de esta forma.

```
DataInputStream inStream = new
        DataInputStream(connection.getInputStream());
String inputLine;

while (null != (inputLine = inStream.readLine())) {
   System.out.println(inputLine);
}
inStream.close();
```

Cuando ejecutes el programa ReverseTest utilizando Invierteme como argumento, deberías ver esta salida.

```
Invierteme
 reversed is.
emetreivnI
```

CAPITULO III
SOCKETS

3.1. Fundamentos

Los *sockets* son un sistema de comunicación entre procesos de diferentes máquinas de una red. Más exactamente, un *socket* es un punto de comunicación por el cual un proceso puede emitir o recibir información.

Fueron popularizados por *Berkeley Software Distribution*, de la universidad norteamericana de Berkley. Los *sockets* han de ser capaces de utilizar el protocolo de streams TCP (Transfer Control Protocol) y el de datagramas UDP (User Datagram Protocol).

Utilizan una serie de primitivas para establecer el punto de comunicación, para conectarse a una máquina remota en un determinado puerto que esté disponible, para escuchar en él, para leer o escribir y publicar información en él, y finalmente para desconectarse.

Con todas las primitivas se puede crear un sistema de diálogo muy completo.

3.2. Funcionamiento genérico

Normalmente, un servidor se ejecuta sobre una computadora específica y tiene un *socket* que responde en un puerto específico. El servidor únicamente espera, escuchando a través del *socket* a que un cliente haga una petición.

En el lado del cliente: el cliente conoce el nombre de host de la máquina en la cual el servidor se encuentra ejecutando y el número de puerto en el cual el servidor está conectado. Para realizar una petición de conexión, el cliente intenta encontrar al servidor en la máquina servidora en el puerto especificado.

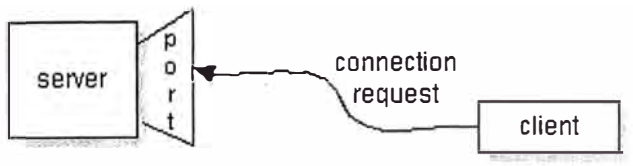


Fig. 3.1. Conexión del Cliente

Si todo va bien, el servidor acepta la conexión. Además de aceptar, el servidor obtiene un nuevo *socket* sobre un puerto diferente. Esto se debe a que necesita un nuevo *socket* (y , en consecuencia, un numero de puerto diferente) para que sigu atendiendo el *socket* original peticiones de conexión mientras atiende las necesidades del cliente que se conectó.

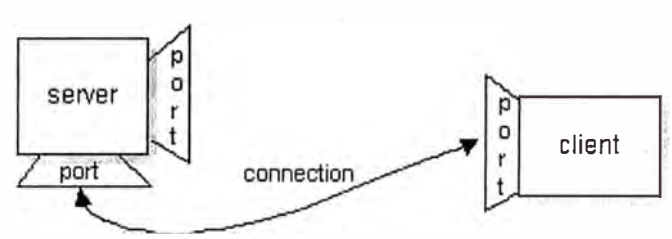


Fig. 3.2. Servidor acepta conexión

Por la parte del cliente, si la conexión es aceptada, se crea un *socket* de forma satisfactoria y puede usarlo para comunicarse con el servidor. Es importante darse cuenta que el *socket* en el cliente no está utilizando el número de puerto usado para realizar la petición al servidor. En lugar de éste, el cliente asigna un número de puerto local a la máquina en la cual está siendo ejecutado.

Ahora el cliente y el servidor pueden comunicarse escribiendo o leyendo en o desde sus respectivos *sockets*.

3.3. Sockets: Interfaz de comunicación entre procesos

El paquete **java.net** de la plataforma Java proporciona una clase **Socket**, la cual implementa una de las partes de la comunicación bidireccional entre un programa Java y otro programa en la red.

La clase **Socket** se sitúa en la parte más alta de una implementación dependiente de la plataforma, ocultando los detalles de cualquier sistema particular al programa Java. Usando la clase **java.net.Socket** en lugar de utilizar código nativo de la plataforma, los programas Java pueden comunicarse a través de la red de una forma totalmente independiente de la plataforma.

De forma adicional, java.net incluye la clase **ServerSocket**, la cual implementa un *socket* el cual los servidores pueden utilizar para escuchar y aceptar peticiones de conexión de clientes.

Nuestro objetivo será conocer cómo utilizar las clases **Socket** y **ServerSocket**. Por otra parte, si intentamos conectar a través de la Web, la clase **URL** y clases relacionadas (**URLConnection**, **URLEncoder**) son probablemente más apropiadas que

las clases de *sockets*. Pero de hecho , las clases **URL** no son más que una conexión a un nivel más alto a la Web y utilizan como parte de su implementación interna los *sockets*.

3.4. Modelo de comunicaciones con Java

El modelo de *sockets* más simple es:· El servidor establece un puerto y espera durante un cierto tiempo (*timeout* segundos), a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión *socket* con el método *accept()*. El cliente establece una conexión con la máquina host a través del puerto que se designe en *puerto#*

El cliente y el servidor se comunican con manejadores **InputStream** y **OutputStream**

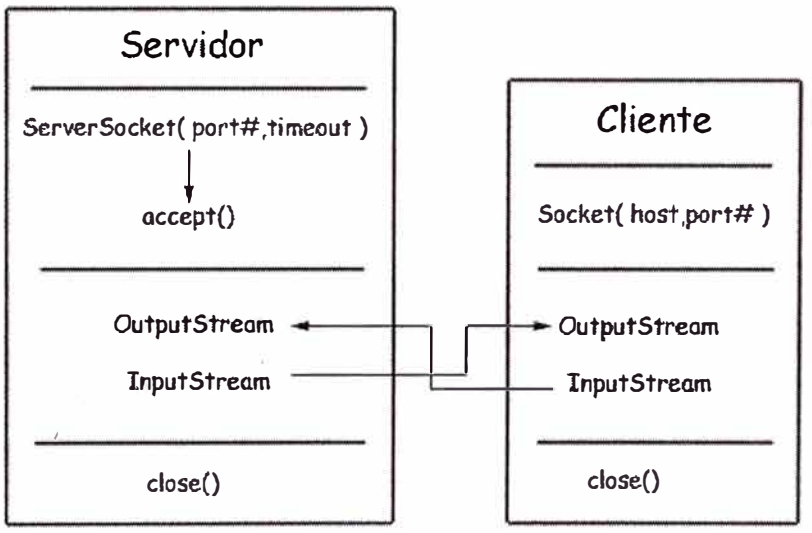


Fig. 3.3. Comunicación Cliente-Servidor

3.5. Apertura de Sockets

Si estamos programando un **CLIENTE**, el socket se abre de la forma:

```
Socket miCliente;
miCliente = new Socket( "maquina", numeroPuerto );
```

Donde *maquina* es el nombre de la máquina en donde estamos intentando abrir la conexión y *numeroPuerto* es el puerto (un número) del servidor que está corriendo sobre el cual nos queremos conectar. Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados para usuarios con muchos privilegios (superusuarios o root). Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp o http. Para las aplicaciones que se desarrollen,

asegurarse de seleccionar un puerto por encima del 1023.

En el ejemplo anterior no se usan excepciones; sin embargo, es una gran idea la captura de excepciones cuando se está trabajando con *sockets.* El mismo ejemplo quedaría como:

```
Socket miCliente;
try {
miCliente = new Socket( "maquina",numeroPuerto );
} catch( IOException e ) { System.out.println( e );
}
```

Si estamos programando un **SERVIDOR**, la forma de apertura del *socket* es la que muestra el siguiente ejemplo:

```
Socket miServicio;
try {
miServicio = new ServerSocket( numeroPuerto );
} catch( IOException e ) { System.out.println( e );
}
```

A la hora de la implementación de un servidor también necesitamos crear un objeto *socket* desde el **ServerSocket** para que esté atento a las conexiones que le puedan realizar clientes potenciales y poder aceptar esas conexiones:

```
Socket socketServicio = null;
try {
socketServicio = miServicio.accept();
} catch( IOException e ) { System.out.println( e );
}
```

3.6. Creación de Streams
(Streams: Flujo de transferencia de datos)

3.6.1. Creación de Streams de Entrada

En la parte **CLIENTE** de la aplicación, se puede utilizar la clase **DataInputStream** para crear un stream de entrada que esté listo a recibir todas las respuestas que el

servidor le envíe.

```
DataInputStream entrada;
try {
entrada = new DataInputStream( miCliente.getInputStream() );
} catch( IOException e ) { System.out.println( e );
}
```

La clase **DataInputStream** permite la lectura de líneas de texto y tipos de datos primitivos de Java de un modo altamente portable; dispone de métodos para leer todos esos tipos como: *read()*, *readChar()*, *readInt()*, *readDouble()* y *readLine()*. Deberemos utilizar la función que creamos necesaria dependiendo del tipo de dato que esperemos recibir del servidor.

En el lado del **SERVIDOR**, también usaremos **DataInputStream**, pero en este caso para recibir las entradas que se produzcan de los clientes que se hayan conectado:

```
DataInputStream entrada;
try {
entrada =
new DataInputStream( socketServicio.getInputStream() );
} catch( IOException e ) { System.out.println( e );
}
```

3.6.2. Creación de Streams de Salida

En el lado del **CLIENTE**, podemos crear un stream de salida para enviar información al socket del servidor utilizando las clases **PrintStream** o **DataOutputStream**:

```
PrintStream salida;
try {
salida = new PrintStream( miCliente.getOutputStream() );
} catch( IOException e ) { System.out.println( e );
}
```

La clase **PrintStream** tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos *write* y *println()* tienen una especial importancia en

este aspecto. No obstante, para el envío de información al servidor también podemos utilizar

DataOutputStream:

```
DataOutputStream salida;
try {
salida = new DataOutputStream( miCliente.getOutputStream() );
} catch( IOException e ) { System.out.println( e );
}
```

La clase **DataOutputStream** permite escribir cualquiera de los tipos primitivos de Java, muchos de sus métodos escriben un tipo de dato primitivo en el stream de salida. De todos esos métodos, el más útil quizás sea *writeBytes()*.

En el lado del **SERVIDOR**, podemos utilizar la clase **PrintStream** para enviar información al cliente:

```
PrintStream salida;
try {
salida = new PrintStream( socketServicio.getOutputStream() );
} catch( IOException e ) { System.out.println( e );
}
```

Pero también podemos utilizar la clase **DataOutputStream** como en el caso de envío de información desde el cliente.

3.7. Cierre de Sockets

Siempre deberemos cerrar los canales de entrada y salida que se hayan abierto durante la ejecución de la aplicación. En la parte del cliente:

```
try {
salida.close(); entrada.close(); miCliente.close();
} catch( IOException e ) { System.out.println( e );
}
```

Y en la parte del servidor:

```
try {
salida.close(); entrada.close(); socketServicio.close(); miServicio.close();
} catch( IOException e ) { System.out.println( e );
}
```

Es importante destacar que el orden de cierre es relevante. Es decir, se deben cerrar primero los streams relacionados con un *socket* antes que el propio *socket*, ya que de esta forma evitamos posibles errores de escrituras o lecturas sobre descriptores ya cerrados.

3.8. Clases útiles en comunicaciones

3.8.1. Socket

Es el objeto básico en toda comunicación a través de Internet, bajo el protocolo TCP. Esta clase proporciona métodos para la entrada/salida a través de streams que hacen la lectura y escritura a través de sockets muy sencilla.

3.8.2. ServerSocket

Es un objeto utilizado en las aplicaciones servidor para escuchar las peticiones que realicen los clientes conectados a ese servidor. Este objeto no realiza el servicio, sino que crea un objeto Socket en función del cliente para realizar toda la comunicación a través de él.

3.8.3. DatagramSocket

La clase de sockets datagrama puede ser utilizada para implementar datagramas no fiables(sockets UDP), no ordenados. Aunque la comunicación por estos sockets es muy rápida porque no hay que perder tiempo estableciendo la conexión entre cliente y servidor.

3.8.4. DatagramPacket

Clase que representa un paquete datagrama conteniendo información de paquete, longitud de paquete, direcciones Internet y números de puerto.

3.8.5. MulticastSocket

Clase utilizada para crear una versión multicast de las clase socket datagrama. Múltiples clientes/servidores pueden transmitir a un grupo multicast (un grupo de direcciones IP compartiendo el mismo número de puerto).

3.8.6. NetworkServer

Una clase creada para implementar métodos y variables utilizadas en la creación de un servidor TCP/IP.

3.8.7. NetworkClient

Una clase creada para implementar métodos y variables utilizadas en la creación de un cliente TCP/IP.

3.8.8. SocketImpl

Es un Interface que nos permite crearnos nuestro propio modelo de comunicación. Tendremos que implementar sus métodos cuando la usemos. Si vamos a desarrollar una aplicación con requerimientos especiales de comunicaciones, como pueden se la implementación de un **cortafuegos** (TCP es un protocolo no seguro), o acceder a equipos especiales (como un lector de código de barras o un GPS diferencial), necesitaremos nuestra propia clase **Socket**.

3.9. Ejemplo de uso

Para comprender el funcionamiento de los *sockets* no hay nada mejor que estudiar un ejemplo. El que a continuación se presenta establece un pequeño diálogo entre un programa servidor y sus clientes, que intercambiarán cadenas de información.

3.9.1. Programa Ćliente

El programa cliente se conecta a un servidor indicando el nombre de la máquina y el número puerto (tipo de servicio que solicita) en el que el servidor está instalado. Una vez conectado, lee una cadena del servidor y la escribe en la pantalla:

```
import java.io.*; import java.net.*; class Cliente {
static final String HOST = "localhost";
static final int PUERTO=5000;
public Cliente( ) {
try{
Socket skCliente = new Socket( HOST , Puerto );
InputStream aux = skCliente.getInputStream(); DataInputStream flujo = new DataInputStream
( aux ); System.out.println( flujo.readUTF() );
skCliente.close();
} catch( Exception e ) {
```

```
System.out.println( e.getMessage() );
}
}
public static void main( String[] arg ) {
new Cliente();
}
}
```

En primer lugar se crea el *socket* denominado *skCliente*, al que se le especifican el nombre de *host* (*HOST*) y el número de puerto (*PORT*) en este ejemplo constantes.

Luego se asocia el flujo de datos de dicho *socket* (obtenido mediante *getInputStream)*), que es asociado a un flujo (*flujo*) *DataInputStream* de lectura secuencial. De dicho flujo capturamos una cadena (*readUTF()*), y la imprimimos por pantalla (*System.out*).

El *socket* se cierra, una vez finalizadas las operaciones, mediante el método *close()*.

Debe observarse que se realiza una gestión de excepción para capturar los posibles fallos tanto de los flujos de datos como del *socket*.

3.9.2. Programa Servidor

El programa servidor se instala en un puerto determinado, a la espera de conexiones, a las que tratará mediante un segundo *socket*.

Cada vez que se presenta un cliente, le saluda con una frase "Hola cliente N".

Este servidor sólo atenderá hasta tres clientes, y después finalizará su ejecución, pero es habitual utilizar bucles infinitos (*while(true)*) en los servidores, para que atiendan llamadas continuamente.

Tras atender cuatro clientes, el servidor deja de ofrecer su servicio:

```
import java.io.* ; import java.net.* ; class Servidor {
static final int PUERTO=5000;
public Servidor( ) {
try {
ServerSocket skServidor = new ServerSocket(PUERTO); System.out.println("Escucho el puerto " +
PUERTO );
for ( int numCli = 0; numCli < 3; numCli++; ) {
Socket skCliente = skServidor.accept(); // Crea objeto
```

```
System.out.println("Sirvo al cliente " + numCli); OutputStream aux = skCliente.getOutputStream();
DataOutputStream flujo= new DataOutputStream( aux ); flujo.writeUTF( "Hola cliente " + numCli );
skCliente.close();
}
System.out.println("Demasiados clientes por hoy");
} catch( Exception e ) {
System.out.println( e.getMessage() );
}
}
public static void main( String[] arg ) {
new Servidor();
}
}
```

Utiliza un objeto de la clase *ServerSocket* (*skServidor*), que sirve para esperar las conexiones en un puerto determinado (*PUERTO*), y un objeto de la clase *Socket* (*skCliente)* que sirve para gestionar una conexión con cada cliente.

Mediante un bucle *for* y la variable *numCli* se restringe el número de clientes a tres, con lo que cada vez que en el puerto de este servidor aparezca un cliente, se atiende y se incrementa el contador.

Para atender a los clientes se utiliza la primitiva *accept()* de la clase *ServerSocket*, que es una rutina que crea un nuevo *Socket* (*skCliente*) para atender a un cliente que se ha conectado a ese servidor.

Se asocia al *socket* creado (*skCliente*) un flujo (*flujo)* de salida *DataOutputStream* de escritura secuencial, en el que se escribe el mensaje a enviar al cliente.

El tratamiento de las excepciones es muy reducido en nuestro ejemplo, tan solo se captura e imprime el mensaje que incluye la excepción mediante *getMessage()*.

3.9.3. Ejecución

Aunque la ejecución de los *sockets* está diseñada para trabajar con ordenadores en red, en sistemas operativos multitarea (por ejemplo Windows y UNIX) se puede probar el correcto funcionamiento de un programa de *sockets* en una misma máquina.

Para ellos se ha de colocar el servidor en una ventana, obteniendo lo siguiente:

```
>java Servidor
Escucho el puerto 5000
```

En otra ventana se lanza varias veces el programa cliente, obteniendo:

```
>java Cliente
Hola cliente 1
>java cliente
Hola cliente 2
>java cliente
Hola cliente 3
>java cliente
connection refused: no further information
```

Mientras tanto en la ventana del servidor se ha impreso: Sirvo al cliente 1

```
Sirvo al cliente 2
Sirvo al cliente 3
Demasiados cllentes por hoy
```

Cuando se lanza el cuarto de cliente, el servidor ya ha cortado la conexión, con lo que se lanza una excepción.

Obsérvese que tanto el cliente como el servidor pueden leer o escribir del *socket*. Los mecanismos de comunicación pueden ser refinados cambiando la implementación de los *sockets*, mediante la utilización de las clases abstractas que el paquete **java.net** provee.

Aplicaciones

```
// Set up a Server that will receive a connection from a client, send
// a string to the client, and close the connection.
import java.io.EOFException;
import java.io.IOException;
```

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class Server extends JFrame
{
   private JTextField enterField; // inputs message from user
   private JTextArea displayArea; // display information to user
   private ObjectOutputStream output; // output stream to client
   private ObjectInputStream input; // input stream from client
   private ServerSocket server; // server socket
   private Socket connection; // connection to client
   private int counter = 1; // counter of number of connections

   // set up GUI
   public Server()
   {
      super( "Server" );

      enterField = new JTextField(); // create enterField
      enterField.setEditable( false );
      enterField.addActionListener(
         new ActionListener()
         {
            // send message to client
            public void actionPerformed( ActionEvent event )
            {
```

```
            sendData( event.getActionCommand() );
            enterField.setText( "" );
         } // end method actionPerformed
      } // end anonymous inner class
   ); // end call to addActionListener

   add( enterField, BorderLayout.NORTH );

   displayArea = new JTextArea(); // create displayArea
   add( new JScrollPane( displayArea ), BorderLayout.CENTER );

   setSize( 300, 150 ); // set size of window
   setVisible( true ); // show window
} // end Server constructor

// set up and run server
public void runServer()
{
   try // set up server to receive connections; process connections
   {
      server = new ServerSocket( 12345, 100 ); // create ServerSocket

      while ( true )
      {
         try
         {
            waitForConnection(); // wait for a connection
            getStreams(); // get input & output streams
            processConnection(); // process connection
         } // end try
         catch ( EOFException eofException )
         {
            displayMessage( "\nServer terminated connection" );
         } // end catch
         finally
         {
```

```
            closeConnection(); //  close connection
            counter++;
          } // end finally
        } // end while
      } // end try
      catch ( IOException ioException )
      {
        ioException.printStackTrace();
      } // end catch
    } // end method runServer

    // wait for connection to arrive, then display connection info
    private void waitForConnection() throws IOException
    {
      displayMessage( "Waiting for connection\n" );
      connection = server.accept(); // allow server to accept connection
      displayMessage( "Connection " + counter + " received from: " +
        connection.getInetAddress().getHostName() );
    } // end method waitForConnection

    // get streams to send and receive data
    private void getStreams() throws IOException
    {
      // set up output stream for objects
      output = new ObjectOutputStream( connection.getOutputStream() );
      output.flush(); // flush output buffer to send header information

      // set up input stream for objects
      input = new ObjectInputStream( connection.getInputStream() );

      displayMessage( "\nGot I/O streams\n" );
    } // end method getStreams

    // process connection with client
    private void processConnection() throws IOException
    {
```

```
      String message = "Connection successful";
      sendData( message ); // send connection successful message

      // enable enterField so server user can send messages
      setTextFieldEditable( true );

      do // process messages sent from client
      {
         try // read message and display it
         {
            message = ( String ) input.readObject(); // read new message
            displayMessage( "\n" + message ); // display message
         } // end try
         catch ( ClassNotFoundException classNotFoundException )
         {
            displayMessage( "\nUnknown object type received" );
         } // end catch

      } while ( !message.equals( "CLIENT>>> TERMINATE" ) );
   } // end method processConnection

   // close streams and socket
   private void closeConnection()
   {
      displayMessage( "\nTerminating connection\n" );
      setTextFieldEditable( false ); // disable enterField

      try
      {
         output.close(); // close output stream
         input.close(); // close input stream
         connection.close(); // close socket
      } // end try
      catch ( IOException ioException )
      {
         ioException.printStackTrace();
```

```
    } // end catch
} // end method closeConnection

// send message to client
private void sendData( String message )
{
    try // send object to client
    {
        output.writeObject( "SERVER>>> " + message );
        output.flush(); // flush output to client
        displayMessage( "\nSERVER>>> " + message );
    } // end try
    catch ( IOException ioException )
    {
        displayArea.append( "\nError writing object" );
    } // end catch
} // end method sendData
// manipulates displayArea in the event-dispatch thread
private void displayMessage( final String messageToDisplay )
{
    SwingUtilities.invokeLater(
        new Runnable()
        {
            public void run() // updates displayArea
            {
                displayArea.append( messageToDisplay ); // append message
            } // end method run
        } // end anonymous inner class
    ); // end call to SwingUtilities.invokeLater
} // end method displayMessage

// manipulates enterField in the event-dispatch thread
private void setTextFieldEditable( final boolean editable )
{
    SwingUtilities.invokeLater(
        new Runnable()
```

```
        {
          public void run() // sets enterField's editability
          {
            enterField.setEditable( editable );
          } // end method run
        }  // end inner class
      ); // end call to SwingUtilities.invokeLater
   } // end method setTextFieldEditable
} // end class Server

// Client that reads and displays information sent from a Server.
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class Client extends JFrame
{
   private JTextField enterField; // enters information from user
   private JTextArea displayArea; // display information to user
   private ObjectOutputStream output; // output stream to server
   private ObjectInputStream input; // input stream from server
   private String message = ""; // message from server
   private String chatServer; // host server for this application
   private Socket client; // socket to communicate with server
```

```
// initialize chatServer and set up GUI
public Client( String host )
{
   super( "Client" );

   chatServer = host; // set server to which this client connects

   enterField = new JTextField(); // create enterField
   enterField.setEditable( false );
   enterField.addActionListener(
      new ActionListener()
      {
         // send message to server
         public void actionPerformed( ActionEvent event )
         {
            sendData( event.getActionCommand() );
            enterField.setText( "" );
         } // end method actionPerformed
      } // end anonymous inner class
   ); // end call to addActionListener

   add( enterField, BorderLayout.NORTH );

   displayArea = new JTextArea(); // create displayArea
   add( new JScrollPane( displayArea ), BorderLayout.CENTER );

   setSize( 300, 150 ); // set size of window
   setVisible( true ); // show window
} // end Client constructor

// connect to server and process messages from server
public void runClient()
{
   try // connect to server, get streams, process connection
   {
      connectToServer(); // create a Socket to make connection
```

```
      getStreams(); // get the input and output streams
      processConnection(); // process connection
    } // end try
    catch ( EOFException eofException )
    {
      displayMessage( "\nClient terminated connection" );
    } // end catch
    catch ( IOException ioException )
    {
      ioException.printStackTrace();
    } // end catch
    finally
    {
      closeConnection(); // close connection
    } // end finally
  } // end method runClient

  // connect to server
  private void connectToServer() throws IOException
  {
    displayMessage( "Attempting connection\n" );

    // create Socket to make connection to server
    client = new Socket( InetAddress.getByName( chatServer ), 12345 );

    // display connection information
    displayMessage( "Connected to: " +
      client.getInetAddress().getHostName() );
  } // end method connectToServer

  // get streams to send and receive data
  private void getStreams() throws IOException
  {
    // set up output stream for objects
    output = new ObjectOutputStream( client.getOutputStream() );
    output.flush(); // flush output buffer to send header information
```

```
   // set up input stream for objects
   input = new ObjectInputStream( client.getInputStream() );

   displayMessage( "\nGot I/O streams\n" );
} // end method getStreams

// process connection with server
private void processConnection() throws IOException
{
   // enable enterField so client user can send messages
   setTextFieldEditable( true );

   do // process messages sent from server
   {
      try // read message and display it
      {
         message = ( String ) input.readObject(); // read new message
         displayMessage( "\n" + message ); // display message
      } // end try
      catch ( ClassNotFoundException classNotFoundException )
      {
         displayMessage( "\nUnknown object type received" );
      } // end catch

   } while ( !message.equals( "SERVER>>> TERMINATE" ) );
} // end method processConnection

// close streams and socket
private void closeConnection()
{
   displayMessage( "\nClosing connection" );
   setTextFieldEditable( false ); // disable enterField
```

```
   try
   {
      output.close(); // close output stream
      input.close(); // close input stream
      client.close(); // close socket
   } // end try
   catch ( IOException ioException )
   {
      ioException.printStackTrace();
   } // end catch
} // end method closeConnection

// send message to server
private void sendData( String message )
{
   try // send object to server
   {
      output.writeObject( "CLIENT>>> " + message );
      output.flush(); // flush data to output
      displayMessage( "\nCLIENT>>> " + message );
   } // end try
   catch ( IOException ioException )
   {
      displayArea.append( "\nError writing object" );
   } // end catch
} // end method sendData

// manipulates displayArea in the event-dispatch thread
private void displayMessage( final String messageToDisplay )
{
   SwingUtilities.invokeLater(
      new Runnable()
      {
         public void run() // updates displayArea
         {
            displayArea.append( messageToDisplay );
```

```
            } // end method run
         }  // end anonymous inner class
      ); // end call to SwingUtilities.invokeLater
   } // end method displayMessage

   // manipulates enterField in the event-dispatch thread
   private void setTextFieldEditable( final boolean editable )
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // sets enterField's editability
            {
               enterField.setEditable( editable );
            } // end method run
         } // end anonymous inner class
      ); // end call to SwingUtilities.invokeLater
   } // end method setTextFieldEditable
} // end class Client
```

CAPÍTULO IV
DATAGRAMAS

4.1. Datagramas en Java

La clase DatagramPacket, junto con la clase DatagramSocket, son las que se utilizan para la implementación del protocolo UDP (*User Datagram Protocol*).

En este protocolo, a diferencia de lo que ocurría en el protocolo TCP, en el cual si un paquete se dañaba durante la transmisión se reenviaba ese paquete, para asegurar una comunicación segura entre cliente y servidor; con UDP no hay garantía alguna de que los paquetes lleguen en el orden correcto a su destino y, ni tan siquiera hay seguridad de que lleguen todos los paquetes que se hayan enviado. Sin embargo, los paquetes que consiguen llegar, lo hacen mucho más rápidamente que con TCP y, en algunos casos, la velocidad de transmisión es mucho más importante que el que lleguen todos los paquetes; por ejemplo, si lo que se están transmitiendo son señales de sensores en tiempo real para la presentación en pantalla, la velocidad es más importante que la integridad, ya que si un paquete no llega o no puede recomponerse, en el instante siguiente llegará otro.

La programación para uso del protocolo UDP se diferencia de la programación para utilizar el protocolo TCP en que no existe el concepto de ServerSocket para los datagramas y que es el programador el que debe construirse sus propios paquetes a enviar por UDP.

Para enviar datos a través de UDP, hay que construir un objeto de tipo DatagramPacket y enviarlo a través de un objeto DatagramSocket, y al revés para recibirlos, es decir, a través de un objeto DatagramSocket se recoge el objeto DatagramPacket. Toda la información respecto a la dirección, puerto y datos, está contenida en el paquete.

Para enviar un paquete, primero se construye ese paquete con la información que se desea transmitir, luego se almacena en un objeto DatagramSocket y, finalmente se invoca el método *send()* sobre ese objeto. Para recibir un paquete, primero se construye un paquete vacío, luego se le presenta a un objeto DatagramSocket para que almacene allí el resultado de la ejecución del método *receive()* sobre ese objeto.

Hay que tener en cuenta que el hilo de ejecución encargado de todo esto estará bloqueado en el método *receive()* hasta que un paquete físico de datos se reciba a través de la red; este paquete físico será el que se utilice para rellenar el paquete vacío que se había creado.

También hay que tener cuidado cuando se pone a escuchar a un objeto DatagramSocket en un puerto determinado, ya que va a recibir los datagramas enviados por cualquier cliente. Es decir, que si los mensajes enviados por los clientes están formados por múltiples paquetes; en la recepción pueden llegar paquetes entremezclados de varios clientes y es responsabilidad de la aplicación el ordenarlos.

4.1.1. La clase *DatagramPacket*

Para la clase DatagramPacket se dispone de dos constructores, uno utilizado cuando se quiere enviar paquetes y el otro se usa cuando se quieren recibir paquetes. Ambos requieren que se les proporcione un array de bytes y la longitud que tiene. En el caso de la recepción de datos, no es necesario nada más, los datos que se reciban se depositarán en el array; aunque, en el caso de que se reciban más datos físicos de los que soporta el array, el exceso de información se perderá y se lanzará una excepción de tipo IllegalArgumentException, que a pesar de que no sea necesaria su captura, siempre es bueno recogerla.

Cuando se construye el paquete a enviar, es necesario colocar los datos en el array antes de llamar al método *send()*; además de eso, hay que incluir la longitud de ese array y, también se debe proporcionar un objeto de tipo InetAddress indicando la dirección de destino del paquete y el número del puerto de ese destino en el cual estará escuchando el receptor del mensaje. Es decir, que la dirección de destino y el puerto de escucha deben ir en el paquete, al contrario de lo que pasaba en el caso de TCP que se indicaban en el momento de construir el objeto Socket.

El tamaño físico máximo de un datagrama es 65535 bytes, y teniendo en cuenta que hay que incluir datos de cabecera, esa longitud nunca está disponible para datos de usuario, sino que siempre es algo menor.

La clase DatagramPacket proporciona varios métodos para poder extraer los datos que llegan en el paquete recibido. La información que se obtiene con cada método coincide con el propio nombre del método, aunque hay algunos casos en que es necesario saber interpretar la información que proporciona ese mismo método.

El método *getAddress()* devuelve un objeto de tipo InetAddress que contiene la dirección del host remoto. El saber cuál es el ordenador de origen del envío depende de la forma en que se haya obtenido el datagrama. Si ese datagrama ha sido recibido a

través de Internet, la dirección representará al ordenador que ha enviado el datagrama (el origen del datagrama); pero si el datagrama se ha construido localmente, la dirección representará al ordenador al cual se intenta enviar el datagrama (el destino del datagrama).

De igual modo, el método *getPort()* devuelve el puerto desde el cual ha sido enviado el datagrama, o el puerto a través del cual se enviará, dependiendo de la forma en que se haya obtenido el datagrama.

El método *getData()* devuelve un array de bytes que contiene la parte de datos del datagrama, ya eliminada la cabecera con la información de encaminamiento de ese datagrama. La forma de interpretar ese array depende del tipo de datos que contenga. Los ejemplos que se ven en este Tutorial utilizan exclusivamente datos de tipo String, pero esto no es un requerimiento, y se pueden utilizar datagramas para intercambiar cualquier tipo de datos, siempre que se puedan colocar en un array de bytes en un ordenador y extraerlos de ese array en la parte contraria. Es decir, que la responsabilidad del sistema se limita al desplazamiento del array de bytes de un ordenador a otro, y es responsabilidad del programador el asignar significado a esos bytes.

El método *getLength()* devuelve el número de bytes que contiene la parte de datos del datagrama, y el método *getOffset()* devuelve la posición en la cual empieza el array de bytes dentro del datagrama completo.

4.1.2. La clase *DatagramSocket*

Un objeto de la clase DatagramSocket puede utilizarse tanto para enviar como para recibir un datagrama.

La clase tiene tres constructores. Uno de ellos se conecta al primer puerto libre de la máquina local; el otro permite especificar el puerto a través del cual operará el socket; y el tercero permite especificar un puerto y una dirección para identificar a una máquina concreta.

Independientemente del constructor que se utilice, el puerto desde el cual se envíe el datagrama, siempre se incluirá en la cabecera del paquete. Normalmente, la parte del servidor utilizará el constructor que permite indicar el puerto concreto a usar, ya que sino, la parte cliente no tendría forma de conocer el puerto por el cual le van a llegar los datagramas.

La parte cliente puede utilizar cualquier constructor, pero por flexibilidad, lo mejor es utilizar el constructor que deja que el sistema seleccione uno de los puertos disponibles. El servidor debería entonces comprobar cuál es el puerto que se está utilizando para el envío de datagramas y enviar la respuesta por ese puerto.

A diferencia de esta posibilidad de especificar el puerto, o no hacerlo, no hay ninguna otra diferencia entre los sockets datagrama utilizados por cliente y servidor. Si el lector se encuentra un poco perdido, no desespere, porque en los ejemplos todo esto que es muy difícil explicar con palabras, se ve mucho más claramente al intuir el funcionamiento físico de la comunicación entre cliente y servidor.

Para enviar un datagrama hay que invocar al método *send()* sobre un socket datagrama existente, pasándole el objeto paquete como parámetro. Cuando el paquete es enviado, la dirección y número de puerto del ordenador origen se coloca automáticamente en la porción de cabecera del paquete, de forma que esa información pueda ser recuperada en el ordenador destino del paquete.

Para recibir datagramas, hay que instanciar un objeto de tipo DatagramSocket, conectarse a un puerto determinado e invocar al *método receive()* sobre ese socket. Este método bloquea el hilo de ejecución hasta que se recibe un datagrama, por lo que si es necesario hacer alguna cosa durante la espera, hay que invocar al método *receive()* en su propio hilo de ejecución.

Si se trata de un servidor, hay que conectarse con un puerto específico. Si se trata de un cliente que está esperando respuestas de un servidor, hay que escuchar en el mismo puerto que fue utilizado para enviar el datagrama inicial. Si se envía un datagrama a un puerto anónimo, se puede mantener el socket abierto que fue utilizado en el envío del primer datagrama y utilizar ese mismo socket para esperar la respuesta. También se puede invocar al método *getLocalPort()* sobre el socket antes de cerrarlo, de forma que se pueda saber y guardar el número del puerto que se ha empleado; de este modo se puede cerrar el socket original y abrir otro socket en el mismo puerto en el momento en que se necesite.

Si el datagrama inicial es enviado a un puerto determinado, se puede utilizar el mismo socket para recibir la respuesta, o cerrar el socket original y abrir uno nuevo sobre el mismo puerto. Para responder a un datagrama, hay que obtener la dirección del origen y el número de puerto a través del cual fue enviado el datagrama, de la cabecera del paquete y luego, colocar esta información en el nuevo paquete que se construya con la información a enviar como respuesta. Una vez pasada esta información a la parte de datos del paquete, se invoca al método *send()* sobre el objeto DatagramSocket existente, pasándole el objeto paquete como parámetro. La dirección y número de puerto de la máquina que está enviando será incorporada automáticamente a la cabecera del paquete en el momento de ser enviado.

Es importante tener en cuenta que los números de puerto TCP y UDP no están relacionados. Se puede utilizar el mismo número de puerto en dos procesos si uno se

comunica a través de protocolo TCP y el otro lo hace a través de protocolo UDP. Es muy común que los servidores utilicen el mismo puerto para proporcionar servicios similares a través de los dos protocolos en algunos servicios estándar, como puede ser el *eco*.

4.2. Aplicación: El Cliente ECO

Esta aplicación realiza dos pruebas del servicio *Echo* contra el mismo servidor, enviando una línea de texto al puerto estándar de este servicio, el puerto 7. En la primera prueba utiliza el protocolo TCP/IP y en la segunda emplea un datagrama UDP.

```
import java.net.*;
import java.io.*;
import java.util.*;

class Eco {
  public static void main( String[] args ) {
    String servidor = "fiee.uni.edu.pe";        // servidor
    int puerto = 7;                             // puerto eco
    String cadTcp = "Prueba de Eco TCP";
    String cadUdp = "Prueba de Eco UDP";

    // Primero realizamos el test de Eco con TCP/IP
    try {
      // Abrimos un socket conectado al servidor y al
      // puerto estándar de echo
      Socket socket = new Socket( servidor,puerto );

      // Conseguimos el canal de entrada
      BufferedReader entrada = new BufferedReader(
        new InputStreamReader( socket.getInputStream() ) );

      // Conseguimos el canal de salida, con liberación automática
      PrintWriter salida = new PrintWriter(
        new OutputStreamWriter( socket.getOutputStream() ),true );

      // Envía la línea de texto del mensaje al servidor
      salida.println( cadTcp );
```

```
// Y recoge la respuesta del servidor, presentándola en pantalla
System.out.println( entrada.readLine() );

// se cierra el socket TCP
socket.close();

// Ahora se realiza la prueba con datagramas sobre el mimso
// puerto del mismo servidor
// Convertimos el mensaje en un array de bytes
byte[] mensajeUdp = cadUdp.getBytes();
// Obtenemos la dirección IP del servdor
InetAddress dirIp = InetAddress.getByName( servidor );
// Creamos el paquete que se va a enviar al pueto
DatagramPacket paquete =
  new DatagramPacket( mensajeUdp,mensajeUdp.length,dirIp,puerto );
// Abrimos un socket datagrama para enviarle el mensaje
DatagramSocket socketDgrama = new DatagramSocket();
// Y lo enviamos
socketDgrama.send( paquete );

// Sobreescrimos el mensaje en el paquete para confirmar que el
// eco es réalmente lo que llega
byte[] arrayDatos = paquete.getData();
for( int cnt=0; cnt < paquete.getLength(); cnt++ )
  arrayDatos[cnt] = (byte)'x';
// Escribimos esta versión del mensaje
System.out.println( new String( paquete.getData() ) );
// Ahora recibimos el eco en ese mismo paquete, de forma que
// sobreescriba las "x" que se habían colocado
socketDgrama.receive( paquete );
// Presentamos en pantalla el eco
System.out.println( new String( paquete.getData() ) );

// Se cierra el socket
socketDgrama.close();
} catch( UnknownHostException e ) {
```

```
            e.printStackTrace();
            System.out.println(
              "Debes estar conectado para que esto funcione bien." );
          } catch( SocketException e ) {
            e.printStackTrace();
          } catch( IOException e ) {
            e.printStackTrace();
          }
        }
      }
```

Se instancian dos objetos de tipo String, para utilizar uno diferente en cada protocolo. Luego está la parte correspondiente a la prueba de eco a través de TCP, sobre la que no se insiste más. Una vez cerrado el socket TCP, comienza la prueba de eco a través de UDP.

En primer lugar, se convierte el mensaje que se ha de enviar por UDP a un array de bytes. Hay que instanciar un objeto de tipo InetAddress que contenga la dirección del servidor con el que se va a realizar la conexión y el envío del datagrama con el array de bytes recién creado.

Se crea un objeto de tipo DatagramPacket que contenga el array de bytes de la cadena a enviar, junto con la dirección del servidor y el puerto al que hay que conectarse. Se crea ahora un objeto de tipo DatagramSocket que será utilizado para enviar el paquete al servidor. Sin embargo, hay que recordar que el protocolo UDP no garantiza que el paquete llegue íntegro al servidor, o que tan siquiera llegue.

Y, ya solamente resta invocar al método *send()* sobre el objeto DatagramSocket, pasándole el objeto DatagramPacket como parámetro, tal como muestra la siguiente línea de código. Esto hace que la dirección local y el número de puerto se incorporen en el paquete y éste sea enviado a la dirección y número de puerto que se ha encapsulado en el objeto DatagramPacket en el momento de su creación.

Los mismos objetos DatagramSocket y DatagramPacket serán los utilizados para recibir el paquete de respuesta del servidor (siempre que la suerte acompañe). Se usa un bucle para sobreescribir los datos en el paquete con una letra para poder comprobar que una vez recibido el paquete de respuesta del servidor de eco, los datos son nuevos y no simplemente el residuo del mensaje que se había colocado originalmente en el paquete.

Luego se invoca el método *receive()* sobre el objeto DatagramSocket, pasándole el objeto DatagramPacket como parámetro. Esto hace que el hilo de ejecución se

bloquee en el mismo puerto por el que se ha enviado el paquete, hasta que llegue una respuesta. En el momento en que un paquete físico llegue desde el servidor, se extraen los datos y se colocan en el objeto DatagramPacket que se le ha proporcionado como parámetro. Una vez hecho esto, el hilo de ejecución se desbloquea y el flujo de control de la aplicación sigue por la sentencia que muestra el contenido del paquete.

Y ya, finalmente, se cierra el socket y el programa termina.

El primer fragmento en que hay que detenerse es justo en la declaración del método *main()*, en donde se declaran e inicializan algunas variables importantes, cuyo nombre es autoexplicativo, como se puede comprobar.

```
public static void main( String[] args ) {
String servidor = "fiee.uni.edu.pe";        // servidor
int puerto = 7;                             // puerto eco
String cadTcp = "Prueba de Eco TCP";
String cadUdp = "Prueba de Eco UDP";
```

Luego está todo el código referente al protocolo TCP, que en este caso no resulta interesante, y ya se alcanza el punto del programa en el que se convierte el mensaje en un array de bytes, y se instancia un objeto que identifique al servidor. Se utiliza el método *getBytes()* de la clase String para convertir el mensaje a un array de bytes, como se puede ver.

```
// Convertimos el mensaje en un array de bytes
byte[] mensajeUdp = cadUdp.getBytes();
// Obtenemos la dirección IP del servdor
InetAddress dirIp = InetAddress.getByName( servidor );
```

La siguiente línea de código también es importante, ya que es la que instancia un objeto de tipo DatagramPacket que va a llevar toda la información del mensaje y del servidor, como es el array de bytes creado antes, su longitud, y la dirección y puerto del servidor.

```
// Creamos el paquete que se va a enviar al pueto
DatagramPacket paquete =
  new DatagramPacket( mensajeUdp,mensajeUdp.length,dirIp,puerto );
```

A continuación, se instancia un objeto DatagramSocket anónimo. Esto de *anónimo* puede resultar confuso. El objeto es *anónimo* porque el número de puerto no está especificado. En algún sitio anteriormente se ha indicado que los objetos anónimos son aquellos que no tienen una variable de referencia asignada; que no es el caso que se produce aquí. Este objeto se usa para enviar el datagrama al servidor invocando al método *send()* sobre el objeto DatagramSocket.

```
// Abrimos un socket datagrama para enviarle el mensaje
DatagramSocket socketDgrama = new DatagramSocket();
// Y lo enviamos
socketDgrama.send( paquete );
```

Las siguientes líneas de código son las que sobreescriben los datos del mensaje con una letra, para el propósito que se ha indicado antes de comprobar que el mensaje que se recibe no es en realidad el resto del que se ha mandado. Este fragmento de código también muestra al lector cómo puede acceder a la parte de datos de un objeto DatagramPacket, en caso de que lo necesite para cualquier otro propósito.

```
// Sobreescrimos el mensaje en el paquete para confirmar que el
// eco es realmente lo que llega
byte[] arrayDatos = paquete.getData();
for( int cnt=0; cnt < paquete.getLength(); cnt++ )
  arrayDatos[cnt] = (byte)'x';
```

A partir de este punto, se asume que habrá una respuesta del servidor, así que se invoca el método *receive()* sobre el mismo objeto DatagramSocket que se ha utilizado para enviar el mensaje original al servidor. Esto bloquea el *thread*, o hilo de ejecución, quedando a la espera de respuesta. Aunque no se ha indicado en ningún sitio, es posible utilizar el método *setTimeout()* para indicar la cantidad de tiempo que el método *receive()* estará a la espera y bloqueando, lo cual permite colocar una protección al programa para que no se quede colgado esperando una respuesta que no llegue jamás.

```
// Ahora recibimos el eco en ese mismo paquete, de forma que
// sobreescriba las "x" que se habían colocado
socketDgrama.receive( paquete );
```

Y el resto del código del ejemplo es la presentación en pantalla del mensaje recibido del servidor, que debe coincidir con el enviado, y el código de manejo de excepciones.

4.3. Servidores de Datagramas

En el ejemplo se implementan tres servidores. Uno de ellos es un *servidor Eco UDP* implementado a través de un hilo de ejecución que monitoriza un DatagramSocket sobre el puerto 7. Este servidor devuelve el array de bytes que llegue en cada datagrama que reciba, enviando esos datos de regreso al cliente que haya originado el mensaje.

El segundo servidor es un servidor *Eco TCP* implementado a través de un hilo de ejecución que monitoriza un ServerSocket sobre el puerto 7. Este servidor también devuelve los datos que recibe al cliente que haya realizado la conexión.

El tercer servidor es un servidor *HTTP* muy simple implementado a través de un hilo de ejecución TCP que monitoriza el puerto 80. Este servidor solamente responde al comando GET que se envíe desde un navegador, y enviar un fichero como un stream de bytes. La inclusión de estos tres tipos de servidores es para mostrar al lector la forma en que se pueden utilizar los hilos de ejecución para dar servicio a múltiples puertos, haciendo además una mezcla de protocolos TCP y UDP.

La parte del servidor HTTP se puede comprobar a través de un navegador indicando localhost como nombre del servidor. El código completo del ejemplo es el que se muestra a continuación.

```
import java.net.*;
import java.io.*;
import java.util.*;

public class Ejemplo02 {
  public static void main( String[] args ) {
    // Se instancia el controlador de seguridad propio nuestro
    System.setSecurityManager( new MiSecurityManager() );
    // Se instancia un objeto servidor para escuchar el puerto 80
    ServidorHttp servidorHttp = new ServidorHttp();
    // Se instancia un objeto servidor para escuchar el puerto 7
    ServidorEco servidorEcho = new ServidorEco();
    // Se instancia un objeto servidor UDP para escuchar el puerto 7
    ServidorEcoUdp servidorEchoUdp = new ServidorEcoUdp();
```

```
  }
 }

// Esta clase se utiliza para instanciar un controlador de seguridad que
// solamente permita descargar los ficheros que se encuentren en el
// directorio actual o en cualquier subdirectorio de ese directorio.
// Evidentemente, no instalar este servidor en una red, sin antes haber
// hecho cambios para que el control sea más exhaustivo, porque aquí
// se ha dejado muy abierto, ya que solamente se pretende utilizar para
// pruebas
class MiSecurityManager extends SecurityManager {
 // Este método sobreescrito es el que controla que el servidor solamente
 // permita descargar los ficheros del directorio actual y sus ramas,
 // es decir, que no permite navegar hacia arriba en el árbol, ni
 // permite la descarga de una dirección absoluta, pero tiene
 // muchos agujeros de seguridad, así que no debe utilizarse a no ser
 // que se complete adecuadamente, y además se implemente la respuesta
 // que se va a dar en los demás métodos del controlador de Seguridad
 public void checkRead( String str ) {
  if( new File(str).isAbsolute() || (str.indexOf("..") != -1 ) )
   throw new SecurityException( "\n"+str+": Acceso denegado.");
  }

 // Ahora se sobreescriben los demás métodos, que es lo que hace que el
 // tema de seguridad quede totalmente abierto.
 public void checkAccept( String s,int i ){}
 public void checkAccess( Thread t ){}
 public void checkAccess( ThreadGroup g ){}
 public void checkAwtEventQueueAccess(){}
 public void checkConnect( String s,int i ){}
 public void checkConnect( String s,int i,Object o ){}
 public void checkCreateClassLoader(){}
 public void checkDelete( String s ){}
 public void checkExec( String s ){}
 public void checkExit( int i ){}
 public void checkLink( String s ){}
```

```
  public void checkListen( int i ){}
  public void checkMemberAccess( Class c,int i ){}
  public void checkMulticast( InetAddress a ){}
  public void checkMulticast( InetAddress a,byte b ){}
  public void checkPackageAccess( String s ){}
  public void checkPackageDefinition( String s ){}
  public void checkPrintJobAccess(){}
  public void checkPropertiesAccess(){}
  public void checkPropertyAccess( String s ){}
  public void checkRead( FileDescriptor f ){}
  // public void checkRead( String s ){} // Este es el único sobreescrito
  public void checkRead( String s,Object o ){}
  public void checkSecurityAccess( String s ){}
  public void checkSetFactory(){}
  public void checkSystemClipboardAccess(){}
  public boolean checkTopLevelWindow( Object o ) {
    return true;
    }
  public void checkWrite( FileDescriptor f ){}
  public void checkWrite( String s ){}
  }

// Esta es la clase que se utiliza para instanciar un hilo de ejecución
// para el servidor Udp que se encarga de escuchar el puerto 7, que es
// el definido como estándar para el protocolo de "echo"
class ServidorEcoUdp extends Thread {
  // Constructor
  ServidorEcoUdp() {
    // Arrancamos el hilo e invocamos al método run() para que empiece
    // a correr
    start();
    }

  public void run() {
    try {
      // Se instancia un objeto sobre el puerto 7
```

```
      DatagramSocket socketDgrama = new DatagramSocket( 7 );
      System.out.println( "Servidor Udp escuchando el 7" );
      // Entramos en bucle infinito ecuchando el puerto. Cuando se
      // produce una llamada, se lanza un hilo de ejecución de
      // ConexionEchoUdp para atenderla
      while( true )
        // Limitamos la cadena de eco a 1024 bytes
        DatagramPacket paquete = new DatagramPacket( new byte[1024],1024 );
        // Nos quedamos parados en el receive(), y devolvemos un socket
        // cuando se recibe la llamada. Este socket es el que se pasa
        // como parámetro al nuevo hilo de ejecución que se crrea
        socketDgrama.receive( paquete );
        new ConexionEcoUdp( paquete );
      }
    } catch( IOException e ) {
      e.printStackTrace();
      }
    }
  }

// Esta clase se utiliza la lanzar un hilo de ejecución que atienda
// la llamada récibida a través del puerto 7, el puerto de eco
class ConexionEchoUdp extends Thread {
  DatagramPacket paquete;

  // Constructor
  ConexionEchoUdp( DatagramPacket paquete ) {
    System.out.println( "Recibida una llamada en el puerto 7" );
    this.paquete = paquete;
    // Trabajamos por debajo de la prioridad de los otros puertos
    setPriority( NORM_PRIORITY-1 );
    // Se arranca el hilo y se pone a correr
    start();
  }
```

```
public void run() {
  System.out.println( "Lanzado el hilo UDP de atencion del puerto 7" );

  // Se crea el paquete de eco basándonos en los datos del paquete
  // que se ha recibido como parámetro
  DatagramPacket paqueteEnvio = new DatagramPacket(
    paquete.getData(),paquete.getLength(),
    paquete.getAddress(),paquete.getPort() );
  // Declaramos el socket datagrama
  DatagramSocket socketDgrama = null;

  try {
    // Abrimos un socket datagrama
    socketDgrama = new DatagramSocket();
    // Se utiliza el nuevo socket datagrama para enviar el mensaje
    // y cerrar el socket
    socketDgrama.send( paqueteEnvio );
    socketDgrama.close();
    System.out.println("Socket UDP cerrado." );
    e.printStackTrace();
  }catch( UnknownHostException e ) {
    datagramSocket.close();
    System.out.println("Socket UDP cerrado." );
    e.printStackTrace();
  }catch( SocketException e ) {
    datagramSocket.close();
    System.out.println("Socket UDP cerrado." );
    e.printStackTrace();
  }catch( IOException e ) {
    datagramSocket.close();
    System.out.println("Socket UDP cerrado." );
    e.printStackTrace();
  }
 }
}
```

```
// Esta es la clase que se utiliza para instanciar un hilo de ejecución
// para el servidor que se encarga de escuchar el puerto 7, que es el
// definido como estándar para el protocolo de "echo"
```

Cada uno de los servidores se implementan mediante hilos de ejecución, así que los tres servidores actúan concurrente y asíncronamente en diferente hilos de ejecución. Además, siempre que un objeto servidor necesite proporcionar un servicio a un cliente, lanzará otro hilo de ejecución a una prioridad más baja, para dar servicio a ese cliente, siguiendo a la escucha de otros posibles clientes que requieran su atención.

Dejando a un lado la parte ya vista en el ejemplo base, el primer trozo de código en que hay que detenerse es la implementación del servidor UDP.

```
// Se instancia un objeto servidor UDP para escuchar el puerto 7
ServidorEcoUdp servidorEchoUdp = new ServidorEcoUdp();
```

Y, lo siguiente ya es el propio constructor de esa clase que se utiliza para instanciar el objeto que va a proporcionar los servicios UDP de Eco a través del puerto 7. Este constructor, como se muestra, se limita a invocar a su propio método *start()* para levantarse y empezar a correr. El método *start()* invoca al método *run()*.

```
// Constructor
ServidorEcoUdp() {
  // Arrancamos el hilo e invocamos al método run() para que empiece
  // a correr
  start();
  }
```

La siguiente línea de código muestra la instanciación del objeto DatagramSocket sobre el puerto 7, que ya se encuentra dentro de la acción del método *run()*, es decir, que es el comienzo el hilo de ejecución.

```
// Se instancia un objeto sobre el puerto 7
DatagramSocket socketDgrama = new DatagramSocket( 7 );
```

Y ya, solamente queda el corazón del hilo de ejecución, que está formado por el bucle infinito que instancia en primer lugar un objeto DatagramPacket vacío, para invocar

al método *receive()* sobre el DatagramSocket, pasándole el objeto DatagramPacket como parámetro.

Observe el lector que se ha limitado a 1024 bytes los datos de entrada, por lo que no va a poder recibir mensajes de longitud mayor que esa. Si es necesaria una longitud mayor, es suficiente con adecuar el valor que se le pasa al constructor al que se necesite.

```
// Entramos en bucle infinito ecuchando el puerto. Cuando se
// produce una llamada, se lanza un hilo de ejecución de
// ConexionEchoUdp para atenderla
while( true )
  // Limitamos la cadena de eco a 1024 bytes
  DatagramPacket paquete = new DatagramPacket( new byte[1024],1024 );
  // Nos quedamos parados en el receive(), y devolvemos un socket
  // cuando se recibe la llamada. Este socket es el que se pasa
  // como parámetro al nuevo hilo de ejecución que se crea
  socketDgrama.receive( paquete );
  new ConexionEcoUdp( paquete );
}
```

El método *receive()* bloquea el hilo de ejecución y se queda a la espera de que llegue un paquete datagrama. Cuando esto suceda, se rellenará el objeto DatagramPacket vacío, y se instanciará un nuevo hilo de ejecución de tipo ConexionEcoUdp para atender la petición del cliente, pasándole también como parámetro el objeto DatagramPacket, pero en este caso ya relleno con los datos recibidos en el paquete enviado por el cliente.

Una vez satisfecho el requerimiento del cliente, el hilo de ejecución vuelve al inicio del bucle, instanciando un nuevo objeto DatagramPacket vacío, y bloqueando el hilo de ejecución a la espera de la llegada del siguiente paquete datagrama.

En código que sigue en el programa, corresponde a la clase ConexionEchoUdp, de la cual se instancia un objeto para atender al cliente. Toda la información disponible de este cliente se encuentra en el objeto DatagramPacket que el constructor de este hilo de ejecución recibe como parámetro.

Ese objeto es guardado por el constructor para usarlo más tarde, a continuación fija la prioridad por debajo del nivel de los hilos de ejecución que están monitorizando los puertos, de forma que la actividad del hilo correspondiente a ConexionEchoUdp no interfiera con otros hilos de ejecución que puedan lanzarse para atender a las peticiones

recibidas por esos puertos. Y, por fin, ya se invoca el método *start()*, que a su vez invoca al método *run()* y el hilo de ejecución se pone en marcha. Todo esto es lo que se hace en el código que se muestra.

```
ConexionEchoUdp( DatagramPacket paquete ) {
 System.out.println( "Recibida una llamada en el puerto 7" );
 this.paquete = paquete;
 // Trabajamos por debajo de la prioridad de los otros puertos
 setPriority( NORM_PRIORITY-1 );
 // Se arranca el hilo y se pone a correr
 start();
 }
```

La misión encargada a este hilo de ejecución se limita al envío de una copia de los datos que le llegan en el paquete recibido, de vuelta al cliente que se lo ha enviado. La dirección y puerto de este cliente están incluidos en el paquete, donde el DatagramSocket del cliente los colocó antes de enviar ese paquete.

El objeto DatagramPacket es casi directamente enviable de vuelta al cliente, pero para mostrar el uso de algunos de los métodos de la clase DatagramPacket, se construye un nuevo objeto para enviar de regreso al cliente, lo cual suele suceder más a menudo en servidores que realicen tareas más complejas. El código siguiente es el que permite extraer la información necesaria para generar un nuevo objeto. Como sugerencia al lector, podría intentar incluir la fecha en la parte de datos del objeto, para que la información devuelta al cliente sea la misma que él envió, más la fecha en que se recibió en el servidor; pero esto queda como ejercicio para el lector.

```
// Se crea el paquete de eco basándonos en los datos del paquete
// que se ha recibido como parámetro
DatagramPacket paqueteEnvio = new DatagramPacket(
 paquete.getData(),paquete.getLength(),
 paquete.getAddress(),paquete.getPort() );
```

El último código en que merece la pena detenerse en este ejemplo es ya la instanciación de un nuevo objeto DatagramSocket que se va a utilizar para enviar el nuevo objeto DatagramPacket creado, invocando al método *send()* del socket, de regreso

al cliente. El resto del código ya es el cierre del socket y el tratamiento de las excepciones.

```
// Abrimos un socket datagrama
socketDgrama = new DatagramSocket();
// Se utiliza el nuevo socket datagrama para enviar el mensaje
// y cerrar el socket
socketDgrama.send( paqueteEnvio );
socketDgrama.close();
```

CAPITULO V
ACCESO A PARÁMETROS DE RED

5.1. Introducción

Muchas aplicaciones trabajan con múltiples conexiones de red activas, tales como Ethernet, 802.11 b/g (inalámbrico) y bluetooth. Algunas de estas aplicaciones necesitan acceder a este tipo de información para poder realizar sus actividades de red sobre una conexión especifica.

La clase *java.net.NetworkInterface* provee acceso a este tipo de información

Este capitulo mostrara los usos mas comunes de esta clase y proveerá ejemplos que listaran todas las interfaces de red en un computador como también sus direcciones IP y estados.

5.2. Interfase de Red

Una interfase de red es un punto de interconexión entre un computador y una red privada o publica. Una interfase de red suele ser generalmente una tarjeta de red (NIC) pero no necesariamente tiene que ser un componente físico. Puede ser implementado mediante software. Por ejemplo, la interfase de loopback (127.0.0.1 para IPv4 y ::1 para IPv6) no es un dispositivo físico pero es un componente software que simula una interfase de red. Esta interface es comúnmente usada para pruebas de configuración.

La clase *java.net.NetworkInterface* representa ambos tipos de interfase. Esta clase es útil para sistemas con múltiples NICs. Mediante esta clase se puede especificar cual NIC se debe usar para una actividad de red particular.

Por ejemplo, asumamos que tenemos un computador con dos NICs configurados y se necesita enviar datos a un servidor. Se crearía un socket de la siguiente manera:

```
Socket soc = new java.net.Socket();
soc.connect(new InetSocketAddress(address, port));
```

Para enviar los datos, el sistema determina que interfase será usado. Sin embargo, si se tiene una preferencia o se necesita especificar cual NIC se debe usar,

se puede consultar al sistema por la interfase apropiada y buscar una dirección sobre la interfase que se desea usar. Cuando se crea el socket y se enlaza a la dirección, el sistema usará la interfase asociada. Por ejemplo:

```
NetworkInterface nif = NetworkInterface.getByName("bge0");
Enumeration nifAddresses = nif.getInetAddresses();

Socket soc = new java.net.Socket();
soc.bind(nifAddresses.nextElement());
soc.connect(new InetSocketAddress(address, port));
```

También se puede identificar la interfase local en el cual un grupo multicast esta enlazado. Por ejemplo:

```
NetworkInterface nif = NetworkInterface.getByName("bge0");

MulticastSocket() ms = new MulticastSocket();
ms.joinGroup(new InetSocketAddress(hostname, port) , nif);
```

La clase NetworkInterface se puede usar con las APIs de java de muchas otras formas.

5.3. Leyendo Interfaces de Red

La clase NetworkInterface no tiene un constructor público. Por lo tanto no puede crearse una instancia de esta clase con el operador new. En cambio, los siguiente métodos estáticos están disponibles para poder obtener los detalles de la interfase del sistema: getByInetAddress(), getByName() y getNetworkInterfaces(). Los dos primeros métodos son usados cuando se conoce las direcciones IP o nombre de la interfase particular. El tercer método retorna la lista completa de las interfaces en la maquina.

Las interfaces de red pueden ser organizadas jerárquicamente. La clase NI incluye dos métodos, getParent() y getSubInterfaces() que son pertinentes a la jerarquía de interfaces de red. El método getParent() retorna el padre de NI de una interfase. Si la interfase de red es una subinterfase, getParent() retorna valué no nulo. El método getSubInterfaces() retorna todas las subinterfaces de una interfase de red.

El siguiente programa ejemplo lista los nombre de todas las interfaces de red y subinterfaces (si es que existen) en una máquina.

```
import java.io.*;
import java.net.*;
import java.util.*;
import static java.lang.System.out;

public class ListNIFs
{
   public static void main(String args[]) throws SocketException {
      Enumeration<NetworkInterface> nets =
NetworkInterface.getNetworkInterfaces();

      for (NetworkInterface netIf : Collections.list(nets)) {
         out.printf("Display name: %s\n", netIf.getDisplayName());
         out.printf("Name: %s\n", netIf.getName());
         displaySubInterfaces(netIf);
         out.printf("\n");
      }
   }

   static void displaySubInterfaces(NetworkInterface netIf) throws SocketException {
      Enumeration<NetworkInterface> subIfs = netIf.getSubInterfaces();

      for (NetworkInterface subIf : Collections.list(subIfs)) {
         out.printf("\tSub Interface Display name: %s\n", subIf.getDisplayName());
         out.printf("\tSub Interface Name: %s\n", subIf.getName());
      }
   }
}
```

Lo siguiente es un ejemplo de la salida del programa ejemplo:

```
Display name: bge0
Name: bge0
   Sub Interface Display name: bge0:3
   Sub Interface Name: bge0:3
   Sub Interface Display name: bge0:2
```

```
Sub Interface Name: bge0:2
Sub Interface Display name: bge0:1
Sub Interface Name: bge0:1

Display name: lo0
Name: lo0
```

5.4. Listado de las Direcciones de la Interfase de Red

Una de la más importante información que se puede obtener de una interfase de red es la lista de direcciones IP asignadas a ella. Esta información se puede obtener a través de una instancia de NI mediante el uso de uno de dos métodos. El primer método getInetAddresses() retorna un Enumeration de InetAddress. El otro método, getIterfacesAddresses() retorna una lista de instancia de java.net,InterfaceAddress. Este método es usado cuando se necesita más información acerca de una dirección de interfase aparte de la dirección IP. Por ejemplo, se puede necesitar información adicional acerca de la mascara de subred y dirección de broadcast cuando una dirección esta en Ipv4, y una longitud de prefijo de red en el caso de una dirección Ipv6.

El siguiente programa ejemplo lista todas las interfaces de red y sus direcciones en una máquina.

```
import java.io.*;
import java.net.*;
import java.util.*;
import static java.lang.System.out;

public class ListNets
{
   public static void main(String args[]) throws SocketException {
      Enumeration<NetworkInterface> nets = NetworkInterface.getNetworkInterfaces();
      for (NetworkInterface netint : Collections.list(nets))
         displayInterfaceInformation(netint);
   }

   static void displayInterfaceInformation(NetworkInterface netint) throws SocketException {
      out.printf("Display name: %s\n", netint.getDisplayName());
      out.printf("Name: %s\n", netint.getName());
```

```
            Enumeration<InetAddress> inetAddresses = netint.getInetAddresses();
            for (InetAddress inetAddress : Collections.list(inetAddresses)) {
                out.printf("InetAddress: %s\n", inetAddress);
            }
            out.printf("\n");
        }
    }
```

Lo siguiente es un ejemplo de la salida del programa ejemplo:

```
Display name: bge0
Name: bge0
InetAddress: /fe80:0:0:0:203:baff:fef2:e99d%2
InetAddress: /121.153.225.59

Display name: lo0
Name: lo0
InetAddress: /0:0:0:0:0:0:0:1%1
InetAddress: /127.0.0.1
```

5.5. Parámetros de una Interface de Red

Se puede acceder a los parámetros de red de una interfase de red a través del nombre y dirección IP asignado a el.

Se puede determinar si una interfase de red esta “up” (esto es ejecutándose) con el método isUP(). Los siguientes métodos indican el tipo de la interfase de red:

- isLoopback() indica si en una interfase loopback
- isPointToPoint() indica si es interface punto a punto.
- isVirtual() indica si es una interface virtual.

El método supportsMulticast() indica si la interface de red soporta multicasting. El método getHardwareAddress() retorna la dirección física de una interfase de red, usualmente llamado dirección MAC, cuando esta disponible. El método getMTU() retorna la unidad máxima de transmisión (MTU) el cual es el tamaño del paquete mas largo.

El siguiente ejemplo lista las interfaces de redes de una pc y los parámetros de red descritos:

```
import java.io.*;
import java.net.*;
import java.util.*;
import static java.lang.System.out;

public class ListNetsEx
{
   public static void main(String args[]) throws SocketException {
      Enumeration<NetworkInterface> nets = NetworkInterface.getNetworkInterfaces();
      for (NetworkInterface netint : Collections.list(nets))
         displayInterfaceInformation(netint);
   }

   static void displayInterfaceInformation(NetworkInterface netint) throws SocketException {
      out.printf("Display name: %s\n", netint.getDisplayName());
      out.printf("Name: %s\n", netint.getName());
      Enumeration<InetAddress> inetAddresses = netint.getInetAddresses();

      for (InetAddress inetAddress : Collections.list(inetAddresses)) {
         out.printf("InetAddress: %s\n", inetAddress);
      }

      out.printf("Up? %s\n", netint.isUp());
      out.printf("Loopback? %s\n", netint.isLoopback());
      out.printf("PointToPoint? %s\n", netint.isPointToPoint());
      out.printf("Supports multicast? %s\n", netint.supportsMulticast());
      out.printf("Virtual? %s\n", netint.isVirtual());
      out.printf("Hardware address: %s\n",
            Arrays.toString(netint.getHardwareAddress()));
      out.printf("MTU: %s\n", netint.getMTU());

      out.printf("\n");

   }
}
```

La salida para el ejemplo anterior:

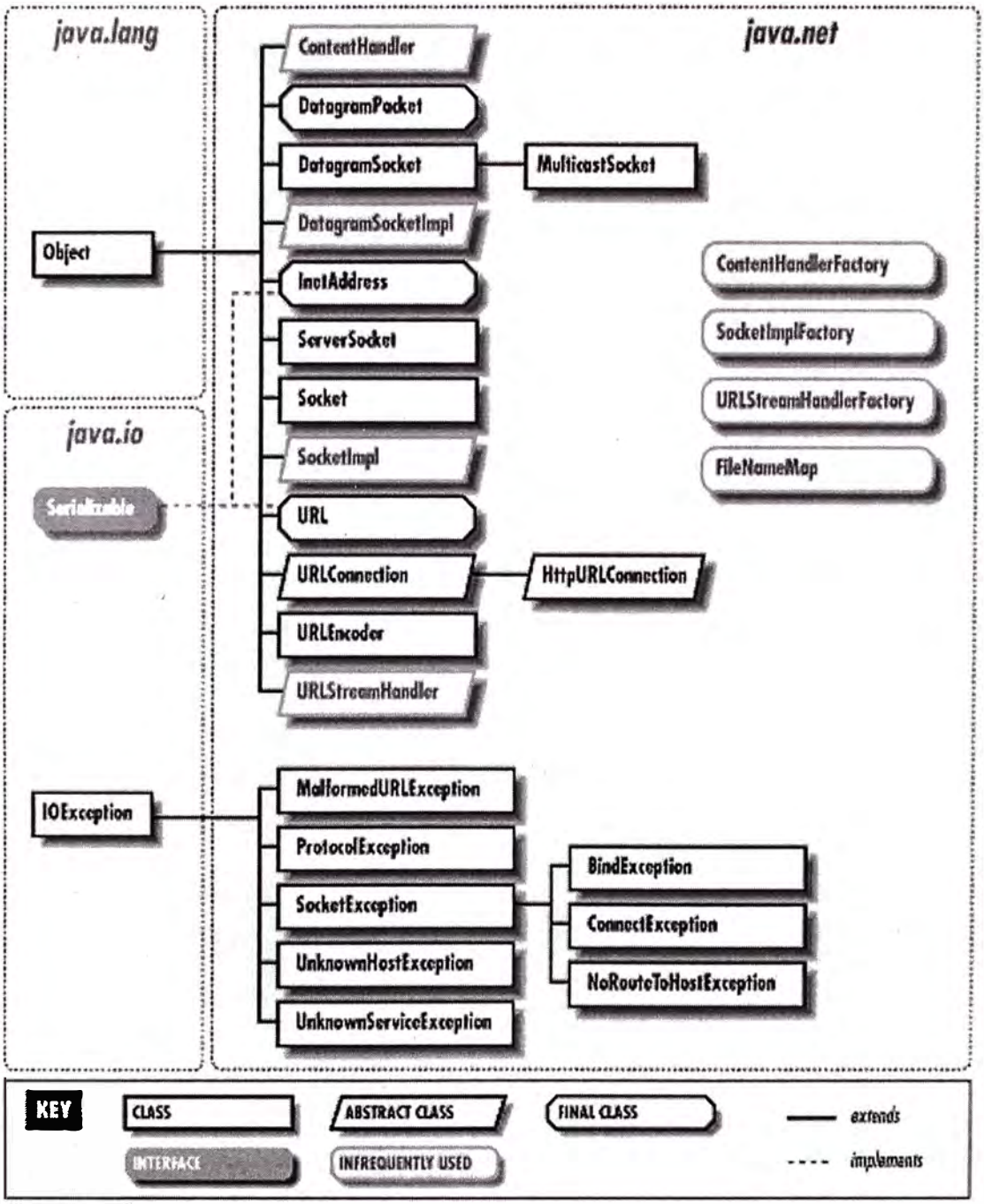
```
Display name: bge0
Name: bge0
InetAddress: /fe80:0:0:0:203:baff:fef2:e99d%2
InetAddress: /129.156.225.59
Up? true
Loopback? false
PointToPoint? false
Supports multicast? false
Virtual? false
Hardware address: [0, 3, 4, 5, 6, 7]
MTU: 1500

Display name: lo0
Name: lo0
InetAddress: /0:0:0:0:0:0:0:1%1
InetAddress: /127.0.0.1
Up? true
Loopback? true
PointToPoint? false
Supports multicast? false
Virtual? false
Hardware address: null
MTU: 8232
```

CONCLUSIONES

1. El lenguaje Java por su arquitectura neutral se convierte en una alternativa para la construcción de aplicaciones orientado al manejo de recursos de red independiente de la plataforma operativa sobre el cual se ejecuta la aplicación.
2. Así mismo, por su orientación a objetos se pueden construir librerías de clases personalizadas orientadas al manejo de recursos de red las cuales pueden ser reutilizadas por otras aplicaciones.
3. Como la tecnología Java tiene una plataforma de desarrollo orientado a dispositivos móviles, se podrían implementar interfaces de programación que gestionen servicios de red propios del medio inalámbrico.
4. En la actualidad, con los recursos proveídos por la tecnología Java se pueden seguir desarrollando aplicaciones que manejen los servicios descritos en el presente informe como URL, TCP y UDP.
5. Para una gestión de red de mayor alcance de servicios, se recomienda seguir utilizando el lenguaje C++ por su madurez en los recursos que provee, sin embargo se espera que el lenguaje Java con el transcurrir del tiempo amplié sus interfaces de programación de red permitiendo usar todas las virtudes de esta tecnología.

ANEXO A

JERARQUIA DE CLASES DE java.net



ANEXO B

LA API java.net

Interfaces	
ContentHandlerFactory	Define al proveedor del manejador de contenidos.
CookiePolicy	Las implementaciones deciden que cookies serán aceptados o rechazados.
CookieStore	El objeto representa un almacenamiento para cookies.
DatagramSocketImplFactory	Define un proveedor par las implementaciones de sockets de tipo datagrama.
FileNameMap	Define un mecanismo para mapear un nombre de archivo y un texto de tipo MIME.
SocketImplFactory	Define un proveedor para la implementación de sockets.
SocketOptions	Contiene métodos para acceder/asignar opciones de sockets.
URLStreamHandlerFactory	Define un proveedor para el manejador de protocolos de flujos URL.

Clases	
Authenticator	Representa a un objeto que conoce como obtener la autenticación de una conexión de red.
CacheRequest	Representa canales para el almacenamiento de recursos en el cache de respuesta.
CacheResponse	Representa canales para la recuperación de recursos desde el cache de respuesta.
ContentHandler	Clase abstracta que es la superclase de todas aquellas que leen un objeto desde una conexión URL.
CookieHandler	Representa un objeto que provee un mecanismo de retrollamada a la implementación de políticas de manejo de estado http dentro del manejador de protocolo.
CookieManager	Implementación de CookieHandler, el cual separa el almacenamiento de cookies de la política de aceptación o rechazo de cookies.
DatagramPacket	Representa un paquete de tipo datagrama.
DatagramSocket	Representa un socket para el envió y recepción de datagramas.
DatagramSocketImpl	Clase base para la implementación de datagramas abstracto y socket multicast.
HttpCookie	Representa un cookie HTTP el cual transporta información de estado entre el servidor y el agente usuario.
HttpURLConnection	Conexión URL con soporte especifico para especificaciones HTTP.
IDN	Provee métodos para convertir nombres de dominios internacionales (IDNs) entre una representación unicode normal y una representación encriptada compatible con ASCII (ACE).
Inet4Address	Representa una dirección del protocolo Internet versión 4 (IPv4).
Inet6Address	Representa una dirección del protocolo Internet versión 6 (IPv6).
InetAddress	Representa una dirección del protocolo internet (IP).
InetSocketAddress	Implementa una dirección de socket IP (dirección IP + número de puerto). También podrá ser nombre de host + numero de puerto, en cuyo caso se tendrá que resolver el nombre de host.
InterfaceAddress	Representa una dirección de interfase de red.
JarURLConnection	Una conexión URL a un archivo java (JAR).

MulticastSocket	Utilidad para el envió y recepción de paquetes multicast IP.
NetPermission	Para definir permisos de red.
NetworkInterface	Representa una interfase de red compuesto de un nombre y una lista de direcciones IP asignados a esta interfase.
PasswordAuthentication	Almacenador de datos que es usado por el autenticador.
Proxy	Representa una configuración de proxy, definiendo un tipo (http, socks) y una dirección de socket.
ProxySelector	Selecciona el servidor proxy a usar cuando se conecte al recurso de la red referenciado por un URL..
ResponseCache	Representa implementaciones de cache de una conexión URL.
SecureCacheResponse	Representa al cache de respuesta originalmente recuperado a través de medios seguros
ServerSocket	Implementa socket del servidor.
Socket	Implementa socket del cliente.
SocketAddress	Representa una dirección de socket sin protocolo asociado.
SocketImpl	Superclase abstracta común a todas las clases que implementan sockets.
SocketPermission	Representa acceso a la red a través de sockets.
URI	Representa una referencia a un identificador de recurso uniforme (URI).
URL	Representa a un localizador de recurso uniforme, una referencia a un recurso en la World Wide Web.
URLClassLoader	Usado para cargar clases y recursos desde una ruta de búsqueda de un URL que referencia a archivos y carpetas JAR.
URLConnection	Superclase abstracta de todas las clases que representan un enlace de comunicación entre la aplicación y una URL.
URLDecoder	Clase utilitaria para encriptamiento de formulario HTML.
URLEncoder	Clase utilitaria para encriptamiento de formulario HTML.
URLStreamHandler	Superclase abstracta común para todos los manejadores de protocolos de flujos.

Enumerados	
Authenticator.RequestorType	Tipo de la entidad que solicita una autenticación.
Proxy.Type	Representa el tipo de Proxy.

Excepciones	
BindException	Error producido mientras se intentaba enlazar un socket a una dirección y puerto local.
ConnectException	Error producido mientras se intentaba conectar un socket a una dirección y puerto remoto.
HttpRetryException	Indica que una petición HTTP necesita ser reintentado pero no se puede hacer automáticamente.
MalformedURLException	Indica que se ha construido un mal URL.
NoRouteToHostException	Error producido mientras se intentaba conectar un socket a una dirección y puerto remoto.
PortUnreachableException	Indica que se ha recibido un mensaje ICMP de puerto inalcanzable sobre un datagrama conectado.
ProtocolException	Indica que hay un error en el protocolo relacionado tal como un error TCP.
SocketException	Indica que hay un error en el protocolo relacionado tal como un error TCP.
SocketTimeoutException	Indica que ha finalizado el tiempo de espera en la lectura o aceptación de socket.
UnknownHostException	Indica que una dirección IP de un host no puede ser determinado.
UnknownServiceException	Indica que un servicio desconocido ha ocurrido.
URISyntaxException	Indica que un texto no puede ser convertido como una referencia URI.

BIBLIOGRAFIA

1. Sun Microsystems, "Custom Networking", The Java Tutorial, 2007

2. Sun Microsystems, "Package java.net", Java Platform Standard Edition 6, 2007

3. Jan Graba, "An Introduction to Network Programming with Java", Springer, 2007

4. Elliotte Rusty Harold, "Java Network Programming, 3rd Edition", O'Reilly, 2004

5. David Reilly – Michael Reilly, "Java Network Programming and Distributed Computing", Addison Wesley, 2002