8 COMUNICACIONES

En esta clase no nos vamos a extender demasiado en profundidades sobre la comunicación y funcionamiento de redes, aunque sí proporcionaremos una breve introducción para sentar, o recordar, los fundamentos de la comunicación en red, tomando como base Unix. Presentaremos un ejemplo básico de cliente/servidor sobre sockets TCP/IP, proporcionando un punto de partida para el desarrollo de otras aplicaciones cliente/servidor basadas en sockets, que posteriormente implementaremos.

8.1 Comunicaciones en Unix

El sistema de Entrada/Salida de Unix sigue el paradigma que normalmente se designa como Abrir-Leer-Escribir-Cerrar. Antes de que un proceso de usuario pueda realizar operaciones de entrada/salida, debe hacer una llamada a Abrir (open) para indicar, y obtener permisos para su uso, el fichero o dispositivo que quiere utilizar. Una vez que el objeto está abierto, el proceso de usuario realiza una o varias llamadas a Leer (read) y Escribir (write), para conseguir leer y escribir datos. Leer coge datos desde el objeto y los transfiere al proceso de usuario, mientras que Escribir transfiere datos desde el proceso de usuario al objeto. Una vez que todos estos intercambios de información estén concluidos, el proceso de usuario llamará a Cerrar (close) para informar al sistema operativo que ha finalizado la utilización del objeto que antes había abierto.

Cuando se incorporan las características a Unix de comunicación entre procesos (IPC) y el manejo de redes, la idea fue implementar la interface con IPC similar a la que se estaba utilizando para la entrada/salida de ficheros, es decir, siguiendo el paradigma del párrafo anterior. En Unix, un proceso tiene un conjunto de descriptores de entrada/salida desde donde Leer y por donde Escribir. Estos descriptores pueden estar referidos a ficheros, dispositivos, o canales de comunicaciones (sockets). El ciclo de vida de un descriptor, aplicado a un canal de comunicación (socket), está determinado por tres fases (siguiendo el paradigma):

- Creación, apertura del socket
- Lectura y Escritura, recepción y envío de datos por el socket
- Destrucción, cierre del socket

La interface IPC en Unix-BSD está implementada sobre los protocolos de red TCP y UDP. Los destinatarios de los mensajes se especifican como direcciones de socket; cada dirección de socket es un identificador de comunicación que consiste en una dirección Internet y un número de puerto.

Las operaciones IPC se basan en pares de sockets. Se intercambian información transmitiendo datos a través de mensajes que circulan entre un socket en un proceso y otro socket en otro proceso. Cuando los mensajes son enviados, se encolan en el socket hasta que el protocolo de red los haya transmitido. Cuando llegan, los mensajes son encolados en el socket de recepción hasta que el proceso que tiene que recibirlos haga las llamadas necesarias para recoger esos datos.

8.2 Sockets

Los sockets son puntos finales de enlaces de comunicaciones entre procesos. Los procesos los tratan como descriptores de ficheros, de forma que se pueden intercambiar datos con otros procesos transmitiendo y recibiendo a través de sockets.

El tipo de sockets describe la forma en la que se transfiere información a través de ese socket.

8.2.1 Sockets Stream (TCP, Transport Control Protocol)

Son un servicio orientado a conexión donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados.

El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.

8.2.2 Sockets Datagrama (UDP, User Datagram Protocol)

Son un servicio de transporte sin conexión. Son más eficientes que TCP, pero no está garantizada la fiabilidad. Los datos se envían y reciben en paquetes, cuya entrega no está garantizada. Los paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.

El protocolo de comunicaciones con datagramas es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación.

8.2.3 Sockets Raw

Son sockets que dan acceso directo a la capa de software de red subyacente o a protocolos de más bajo nivel. Se utilizan sobre todo para la depuración del código de los protocolos.

Diferencias entre Sockets Stream y Datagrama
Ahora se nos presenta un problema, ¿qué protocolo, o tipo de sockets, debemos usar - UDP o TCP? La decisión depende de la aplicación cliente/servidor que estemos escribiendo. Vamos a ver algunas diferencias entre los protocolos para ayudar en la decisión.

En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, tenemos que establecer esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión, que no existe en UDP.

En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene límite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.

UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el socket de recepción. Al contrario, TCP

es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.

Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TCP en vez de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado el tiempo adicional que supone realizar la verificación de los datos, los datagramas son un mecanismo realmente útil.

En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y transmisión de ficheros (ftp); que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión; esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

8.3 Uso de Sockets

Podemos pensar que un Servidor Internet es un conjunto de sockets que proporciona capacidades adicionales del sistema, los llamados servicios.

8.3.1 Puertos y Servicios

Cada servicio está asociado a un puerto. Un puerto es una dirección numérica a través de la cual se procesa el servicio. Sobre un sistema Unix, los servicios que proporciona ese sistema se indican en el fichero /etc/services, y algunos ejemplos son:

```
Daytime 13/udp
ftp 21/tcp
telnet 23/tcp telnet
smtp 25/tcp mail
http 80/tcp
```

La primera columna indica el nombre del servicio. La segunda columna indica el puerto y el protocolo que está asociado al servicio. La tercera columna es un alias del servicio; por ejemplo, el servicio smtp, también conocido como mail, es la implementación del servicio de correo electrónico.

Las comunicaciones de información relacionada con Web tienen lugar a través del puerto 80 mediante protocolo TCP. Para emular esto en Java, usaremos la clase Socket. La fecha (daytime). Sin embargo, el servicio que coge la fecha y la hora del sistema, está ligado al puerto 13 utilizando el protocolo UDP. Un servidor que lo emule en Java usaría un objeto DatagramSocket.

8.4 La clase URL

La clase URL contiene contructores y métodos para la manipulación de URL (Universal Resource Locator): un objeto o servicio en Internet. El protocolo TCP necesita dos tipos de información: la dirección IP y el número de puerto. Vamos a ver como podemos recibir pues la página Web principal de nuestro buscador favorito al teclear:

```
http://www.yahoo.com
```

En primer lugar, Yahoo tiene registrado su nombre, permitiendo que se use yahoo.com como su dirección IP, o lo que es lo mismo, cuando indicamos yahoo.com es como si hubiesemos indicado 205.216.146.71, su dirección IP real.

La verdad es que la cosa es un poco más complicada que eso. Hay un servicio, el DNS (Domain Name Service), que traslada www.yahoo.com a 205.216.146.71, lo que nos permite teclear www.yahoo.com, en lugar de tener que recordar su dirección IP.

Si queremos obtener la dirección IP real de la red en que estamos corriendo, podemos realizar llamadas a los métodos getLocalHost() y getAddress(). Primero, getLocalHost() nos devuelve un objeto iNetAddress, que si usamos con getAddress() generará un array con los cuatro bytes de la dirección IP, por ejemplo:

```
InetAddress direccion = InetAddress.getLocalHost();
byte direccionIp[] = direccion.getAddress();
```

Si la dirección de la máquina en que estamos corriendo es 150.150.112.145, entonces:

```
direccionIp[0] = 150
direccionIp[1] = 150
direccionIp[2] = 112
direccionIp[3] = 145
```

Una cosa interesante en este punto es que una red puede mapear muchas direcciones IP. Esto puede ser necesario para un Servidor Web, como Yahoo, que tiene que soportar grandes cantidades de tráfico y necesita más de una dirección IP para poder atender a todo ese tráfico. El nombre interno para la dirección 205.216.146.71, por ejemplo, es www7.yahoo.com. El DNS puede trasladar una lista de direcciones IP asignadas a Yahoo en www.yahoo.com. Esto es una cualidad útil, pero por ahora abre un agujero en cuestión de seguridad.

Ya conocemos la dirección IP, nos falta el número del puerto. Si no se indica nada, se utilizará el que se haya definido por defecto en el fichero de configuración de los servicios del sistema. En Unix se indican en el fichero /etc/services, en Windows-NT en el fichero services y en otros sistemas puede ser diferente.

El puerto habitual de los servicios Web es el 80, así que si no indicamos nada, entraremos en el servidor de Yahoo por el puerto 80. Si tecleamos la URL siguiente en un navegador:

```
http://www.yahoo.com:80
```

también recibiremos la página principal de Yahoo. No hay nada que nos impida cambiar el puerto en el que residirá el servidor Web; sin embargo, el uso del puerto 80 es casi estándar, porque elimina pulsaciones en el teclado y, además, las direcciones URL son lo suficientemente difíciles de recordar como para añadirle encima el número del puerto.
Si necesitamos otro protocolo, como:

```
ftp://ftp.microsoft.com
```

el puerto se derivará de ese protocolo. Así el puerto FTP de Microsoft es el 21, según su fichero services. La primera parte, antes de los dos puntos, de la URL, indica el protocolo que se quiere utilizar en la conexión con el servidor. El protocolo http (HyperText

Transmission Protocol), es el utilizado para manipular documentos Web. Y si no se especifica ningún documento, muchos servidores están configurados para devolver un documento de nombre index.html.

Con todo esto, Java permite los siguientes cuatro constructores para la clase URL:

```
public URL( String spec ) throws MalformedURLException;
public URL( String protocol,String host,int port,String file ) throws
MalformedURLException;
public URL( String protocol,String host,String file ) throws
MalformedURLException;
public URL( URL context,String spec ) throws MalformedURLException;
```

Así que podríamos especificar todos los componenetes del URL como en:

```
URL( "http","www.yahoo.com","80","index.html" );
```

o dejar que los sistemas utilicen todos los valores por defecto que tienen definidos, como en:

```
URL( "http://www.yahoo.com" );
```

y en los dos casos obtendríamos la visualización de la página principal de Yahoo en nuestro navegador.

8.5 Dominios de Comunicaciones

El mecanismo de sockets está diseñado para ser todo lo genérico posible. El socket por sí mismo no contiene información suficiente para describir la comunicación entre procesos. Los sockets operan dentro de dominios de comunicación, entre ellos se define si los dos procesos que se comunican se encuentran en el mismo sistema o en sistemas diferentes y cómo pueden ser direccionados.

8.5.1 Dominio Unix

Bajo Unix, hay dos dominios, uno para comunicaciones internas al sistema y otro para comunicaciones entre sistemas.

Las comunicaciones intrasistema (entre dos procesos en el mismo sistema) ocurren (en una máquina Unix) en el dominio Unix. Se permiten tanto los sockets stream como los datagrama. Los sockets de dominio Unix bajo Solaris 2.x se implementan sobre TLI (Transport Level Interface).
En el dominio Unix no se permiten sockets de tipo Raw.

8.5.2 Dominio Internet

Las comunicaciones intersistemas proporcionan acceso a TCP, ejecutando sobre IP (Internet Protocol). De la misma forma que el dominio Unix, el dominio Internet permite tanto sockets stream como datagrama, pero además permite sockets de tipo Raw.
Los sockets stream permiten a los procesos comunicarse a través de TCP. Una vez establecidas las conexiones, los datos se pueden leer y escribir a/desde los sockets como un flujo (stream) de bytes. Algunas aplicaciones de servicios TCP son:

- File Tranfer Protocol, FTP
- Simple Mail Transfer Protocol, SMTP
- TELNET, servicio de conexión de terminal remoto

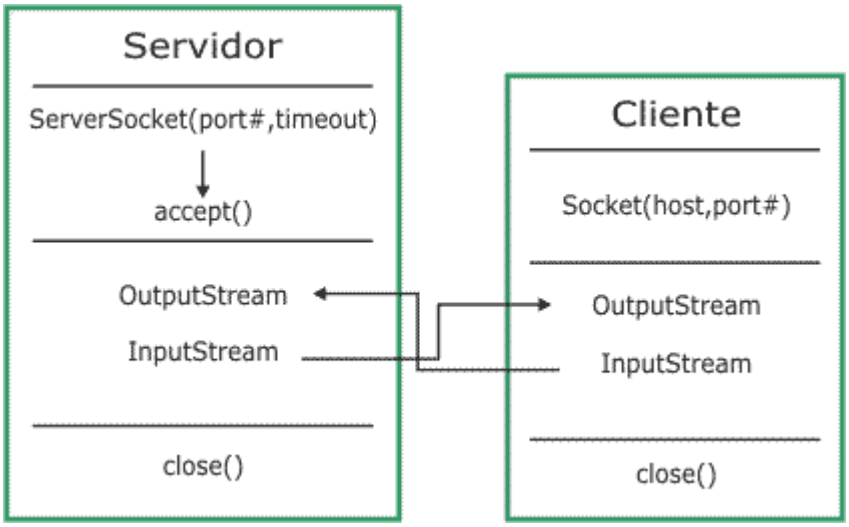
Los sockets datagrama permiten a los procesos utilizar el protocolo UDP para comunicarse a y desde esos sockets por medio de bloques. UDP es un protocolo no fiable y la entrega de los paquetes no está garantizada. Servicios UDP son:

- Simple Network Management Protocol, SNMP
- Trivial File Transfer Protocol, TFTP (versión de FTP sin conexión)
- Versatile Message Transaction Protocol, VMTP (servicio fiable de entrega punto a punto de datagramas independiente de TCP)

Los sockets raw proporcionan acceso al Internet Control Message Protocol, ICMP, y se utiliza para comunicarse entre varias entidades IP.

8.6 Modelo de Comunicaciones con Java

En Java, crear una conexión socket TCP/IP se realiza directamente con el paquete java.net. A continuación mostramos un diagrama de lo que ocurre en el lado del cliente y del servidor:



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

Hay una cuestión al respecto de los sockets, que viene impuesta por la implementación del sistema de seguridad de Java. Actualmente, los applets sólo pueden establecer conexiones con el nodo desde el cual se transfirió su código. Esto está implementado en el JDK y en el intérprete de Java de Netscape. Esto reduce en gran manera la flexibilidad de las fuentes de datos disponibles para los applets. El problema si se permite que un applet se conecte a cualquier máquina de la red, es que entonces se podrían utilizar los

applets para inundar la red desde un ordenador con un cliente Netscape del que no se sospecha y sin ninguna posibilidad de rastreo.

8.7 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 ); }
```

8.8 Creación de Streams

8.8.1 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 ); }
```

8.8.2 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.

8.9 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 );
}
```

8.10 Pequeño Cliente SMTP

Vamos a desarrollar un pequeño cliente SMTP (simple mail transfer protocol), de forma que podamos encapsular todos los datos en la aplicación. El código es libre de modificación para las necesidades que sean; por ejemplo, una modificación interesante sería que aceptase argumentos desde la línea de comandos y también capturase el texto del mensaje desde la entrada estándar del sistema. Con estas modificaciones tendríamos casi la misma aplicación de correo que utiliza Unix. Veamos el código de nuestro cliente, smtpCliente.java:

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

class smtpCliente {
    public static void main( String args[] ) {
       Socket s = null;
       DataInputStream sIn = null;
       DataOutputStream sOut = null;

       // Abrimos una conexión con un servidor de smtp denominado breogan en el puerto
25
       // que es el correspondiente al protocolo smtp, e intentamos abrir los streams de
       //entrada y salida
       try {
          s = new Socket( "breogan",25 );
          sIn = new DataInputStream( s.getInputStream() );
          sOut = new DataOutputStream( s.getOutputStream() );
       }
       catch( UnknownHostException e ) { System.out.println( "No conozco el host" ); }
       catch( IOException e ) { System.out.println( e ); }

       // Si todo está inicializado correctamente, vamos a escribir
       // algunos datos en el canal de salida que se ha establecido
```

```
      // con el puerto del protocolo smtp del servidor
      if( s != null && sIn != null && sOut != null )
        {
        try {
          // Tenemos que respetar la especificación SMTP dada en
          // RFC1822/3, de forma que lo que va en mayúsculas
          // antes de los dos puntos tiene un significado especial
          // en el protocolo
           sOut.writeBytes( "MAIL From: c130710a@ hotmail.com \n" );
          sOut.writeBytes( "RCPT To: direccion@destino\n" );
          sOut.writeBytes( "DATA\n" );
          sOut.writeBytes( "From: c130710a@hotmail.com\n" );
          sOut.writeBytes( "Subject: Pruebas\n" ); // Ahora el cuerpo del mensaje
          sOut.writeBytes( "Hola, desde el Curso de Java\n" );
          sOut.writeBytes( "\n.\n" );

          // Nos quedamos a la espera de recibir el "Ok" del
          // servidor para saber que ha recibido el mensaje
          // correctamente, momento en el cual cortamos
          String respuesta;
          while( ( respuesta = sIn.readLine() ) != null ) {
            System.out.println( "Servidor: "+respuesta );
            if( respuesta.indexOf( "Ok" ) != -1 ) break;
          }

          // Cerramos todo lo que hemos abierto
          sOut.close();
          sIn.close();
          s.close();
        } catch( UnknownHostException e ) {
          System.out.println( "Intentando conectar: "+e );
        } catch( IOException e ) {
          System.out.println( e );
          }
        }
      }
    }
```

8.11 Servidor de Eco

En el siguiente ejemplo, vamos a desarrollar un servidor similar al que se ejecuta sobre el puerto 7 de las máquinas Unix, el servidor echo. Básicamente, este servidor recibe texto desde un cliente y reenvía ese mismo texto al cliente. Desde luego, este es el servidor más simple de los simples que se pueden escribir. El ejemplo que presentamos, ecoServidor.java, maneja solamente un cliente. Una modificación interesante sería adecuarlo para que aceptase múltiples clientes simultáneos mediante el uso de threads.

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

class ecoServidor {
   public static void main( String args[] ) {
```

```
      ServerSocket s = null;
      DataInputStream sIn;
      PrintStream sOut;
      Socket cliente = null;
      String texto;

      // Abrimos una conexión en un servidor en el puerto 9999
      // No podemos elegir un puerto por debajo del 1023 si no somos
      // usuarios con los máximos privilegios (root)
      try { s = new ServerSocket( 9999 ); }
      catch( IOException e ) { }

      // Creamos el objeto desde el cual atenderemos y aceptaremos
      // las conexiones de los clientes y abrimos los canales de
      // comunicación de entrada y salida
      try {
        cliente = s.accept();
        sIn = new DataInputStream( cliente.getInputStream() );
        sOut = new PrintStream( cliente.getOutputStream() );

        // Cuando recibamos datos, se los devolvemos al cliente
        // que los haya enviado
        while( true ) {
          texto = sIn.readLine();
          sOut.println( texto );
        }
      }
      catch( IOException e ) { System.out.println( e ); }
    }
}
```

8.12 Cliente/Servidor TCP/IP

8.12.1 Pequeño Servidor TCP/IP

Veamos el código que presentamos en el siguiente ejemplo, donde desarrollamos un mínimo servidor TCP/IP, para el cual desarrollaremos después su contrapartida cliente TCP/IP. La aplicación servidor TCP/IP depende de una clase de comunicaciones proporcionada por Java: ServerSocket. Esta clase realiza la mayor parte del trabajo de crear un servidor.

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

class minimoServidor {
    public static void main( String args[] ) {
      ServerSocket s = (ServerSocket)null;
      Socket s1;
      String cadena = "Tutorial de Java!";
```

```
      int longCad;
      OutputStream s1out;

      // Establece el servidor en el socket 4321 (espera 300 segundos)
      try { s = new ServerSocket( 4321,300 ); }
      catch( IOException e ) { System.out.println( e ); }

      // Ejecuta un bucle infinito de listen/accept
      while( true ) {
        try {
          // Espera para aceptar una conexión
          s1 = s.accept();
          // Obtiene un controlador de fichero de salida asociado
          // con el socket
          s1out = s1.getOutputStream();

          // Enviamos nuestro texto
          longCad = sendString.length();
          for( int i=0; i < longCad; i++ )
             s1out.write( (int)sendString.charAt( i ) );

          // Cierra la conexión, pero no el socket del servidor
          s1.close();
        }
        catch( IOException e ) { System.out.println( e ); }
      }
    }
}
```

8.12.2 Pequeño Cliente TCP/IP

El lado cliente de una aplicación TCP/IP descansa en la clase Socket. De nuevo, mucho del trabajo necesario para establecer la conexión lo ha realizado la clase Socket. Vamos a presentar ahora el código de nuestro cliente más simple que encaja con el servidor presentado antes. El trabajo que realiza este cliente es que todo lo que recibe del servidor lo imprime por la salida estándar del sistema.

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

class minimoCliente {
    public static void main( String args[] ) throws IOException {
      int c;
      Socket s;
      InputStream sIn;

      // Abrimos una conexión con el localhost en el puerto 4321
      try {
        s = new Socket( "128.0.0.1",4321 );
      } catch( IOException e ) {
```

```
        System.out.println( e );
        }

      // Obtenemos un controlador de fichero de entrada del socket y
      // leemos esa entrada
      sIn = s.getInputStream();
      while( ( c = sIn.read() ) != -1 )
        System.out.print( (char)c );

      // Cuando se alcance el fin de fichero, cerramos la conexión y
      // abandonamos
      s.close();
      }
  }
```

8.13 Problemas más frecuentes

Los tres problemas que pueden presentarse cuando intentemos comprobar el funcionamiento correcto de la red interna que acabamos de montar son:

Cuando hacemos "ping" obtenemos "Bad IP address breogan"

Intentar teclear "ping 102.102.102.102". Si ahora sí se obtiene réplica, la causa del problema es que el los pasos 12 de la Configuración y 3 de la Creación de la entrada en la tabla de hosts, no se ha introducido correctamente el nombre de la máquina. Comprobar esos pasos y que todo coincide.

El programa cliente o el servidor fallan al intentar el "connect"

La causa podría estar en que se produzca un fallo por fichero no encontrado en el directorio Windows/System de las librerías WINSOCK.DLL o WSOCK32.DLL. Muchos programas que se utilizan en Internet reemplazan estos ficheros cuando se instalan. Asegurarse de que están estos ficheros y que son los originales que vienen con la distribución de Windows '95.

El programa servidor dice que no puede "bind" a un socket

Esto sucede porque tiene el DNS activado y no puede encontrar ese DNS o servidor de direcciones, porque estamos solos en la red. Asegurarse de que en el paso 8 de la Configuración la opción de DNS está deshabilitada.

8.14 Clases útiles en Comunicaciones

Vamos a exponer otras clases que resultan útiles cuando estamos desarrollando programas de comunicaciones, aparte de las que ya se han visto. El problema es que la mayoría de estas clases se prestan a discusión, porque se encuentran bajo el directorio sun. Esto quiere decir que son implementaciones Solaris y, por tanto, específicas del Unix Solaris. Además su API no está garantizada, pudiendo cambiar. Pero, a pesar de todo, resultan muy interesantes y vamos a comentar un grupo de ellas solamente que se encuentran en el **paquete sun.net**

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.

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.

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.

DatagramPacket

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

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).

NetworkServer

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

NetworkClient

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

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.

Vamos a ver un ejemplo de utilización, presentando un sencillo ejemplo,

servidorUDP.java, de implementación de sockets UDP utilizando la clase DatagramSocket.

```
import java.net.*;
import java.io.*;
import sun.net.*;
// Implementación del servidor de datagramas UDP. Envía una cadena tras petición
class servidorUDP {
    public static void main( String args[] ) {
       DatagramSocket s = (DatagramSocket)null;
       DatagramPacket enviap,recibep;
       byte ibuffer[] = new byte[100];
       String sendString = "Hola Tutorial de Java!\n";
       InetAddress IP = (InetAddress)null;
       int longitud = sendString.length();
       int puertoEnvio = 4321;
       int puertoRecep = 4322;
       int puertoRemoto;
       // Intentamos conseguir la dirección IP del host
       try {
          IP = InetAddress.getByName( "localhost" );
       } catch( UnknownHostException e ) {
          System.out.println( "No encuentro al host localhost" );
```

```
      System.exit( -1 );
      }
    // Establecemos el servidor para escuchar en el socket 4322
    try {
      s = new DatagramSocket( puertoRecep );
    } catch( SocketException e ) {
      System.out.println( "Error - "+e.toString() );
      }
    // Creamos un paquete de solicitud en el cliente
    // y nos quedamos esperando a sus peticiones
    recibep = new DatagramPacket( ibuffer,longitud );
    try {
      s.receive( recibep );
    } catch( IOException e ) {
      System.out.println( "Error - "+e.toString() );
      }
    // Creamos un paquete para enviar al cliente y lo enviamos
    sendString.getBytes( 0,longitud,ibuffer,0 );
    enviap = new DatagramPacket( ibuffer,longitud,IP,puertoEnvio );
    try {
      s.send( enviap );
    } catch( IOException e ) {
      System.out.println( "Error - "+e.toString() );
      System.exit( -1 );
      }
    // Cerramos el socket
    s.close();
    }
  }
```

Y también vamos a implementar el cliente, clienteUDP.java, del socket UDP correspondiente al servidor que acabamos de presentar:

```
import java.net.*;
import java.io.*;
import sun.net.*;
// Implementación del cliente de datagramas UDP. Devuelve la salida de los servidores
class clienteUDP {
  public static void main( String args[] ) {
    int longitud = 100;
    DatagramSocket s = (DatagramSocket)null;
    DatagramPacket enviap,recibep;
    byte ibuffer[] = new byte[100];
    InetAddress IP = (InetAddress)null;
    int puertoEnvio = 4321;
    int puertoRecep = 4322;
    // Abre una conexión y establece el cliente para recibir
    // una petición en el socket 4321
    try {
      s = new DatagramSocket( puertoRecep );
    } catch( SocketException e ) {
      System.out.println( "Error - "+e.toString() );
      }
```

```
    // Crea una petición para enviar bytes. Intenta conseguir la dirección IP del host
    try {
      IP = InetAddress.getByName( "depserver" );
    } catch( UnknownHostException e ) {
      System.out.println( "No encuentro el host depserver" );
      System.exit( -1 );
      }
    // Envía una petición para que responda el servidor
    try {
      enviap = new DatagramPacket( ibuffer,ibuffer.length,
        IP,4322 );
      s.send( enviap );
    } catch( IOException e ) {
      System.out.println( "Error - "+e.toString() );
      }
    // Consigue un controlador de fichero de entrada del socket y lee
    // dicha entrada. Creamos un paquete descriptor para recibir el
    // paquete UDP
    recibep = new DatagramPacket( ibuffer,longitud );
    // Espera a recibir un paquete
    try {
      s.receive( recibep );
    } catch( IOException e ) {
      System.out.println( "Error - "+e.toString() );
      System.exit( -1 );
      }
    // Imprimimos los resultados de lo que conseguimos
    System.out.println( "Recibido: "+recibep.getLength()+" bytes" );
    String datos = new String( recibep.getData(),0 );
    System.out.println( "Datos: "+datos );
    System.out.println( "Recibido por puerto: "+recibep.getPort() );
    // Cerramos la conexión y abandonamos
    s.close();
    }
  }
```

La salida que se producirá cuando ejecutemos primero el servidor y luego el cliente será la misma que reproducimos a continuación:

```
%java clienteUDP
Recibido: 17 bytes
Datos: Hola Tutorial de Java!
Recibido por puerto: 4322
```