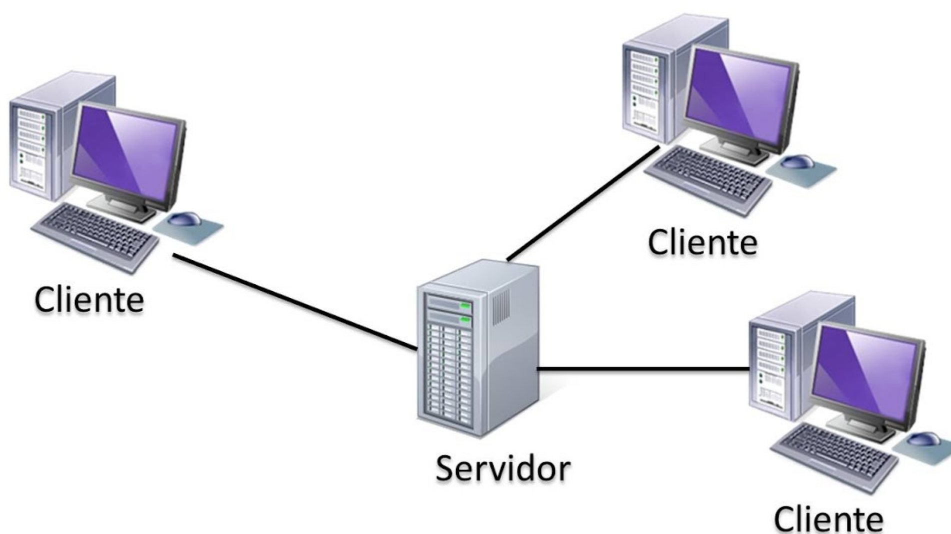
Clase Teórica 7 / 12

Índice de contenido

7.1 Arquitectura Cliente-Servidor....2
7.2 Sockets....4
clientes....5
lectura y escritura por el socket....6
servidores....6
servidor de múltiples clientes....7
clase InetAddress....8
conexiones URL....9
conexiones URLConnection....10
7.4 Cómo establecer un cliente simple utilizando sockets de flujo....11
7.3 Cómo establecer un servidor simple utilizando sockets de flujo....12
7.5 Interacción entre cliente/servidor mediante conexiones de socket de flujo....13
7.6 Interacción entre cliente/servidor sin conexión mediante datagramas....22
Bibliografía....28
Licencia....28

www.profmatiasgarcia.com.ar

7.1 Arquitectura Cliente-Servidor



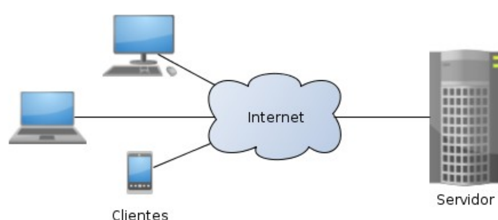
La arquitectura cliente-servidor es un modelo de diseño de software en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un cliente realiza peticiones a otro programa, el servidor, quien le da respuesta. Esta idea también se puede aplicar a programas que se ejecutan sobre una sola computadora, aunque es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de computadoras.

Algunos ejemplos de aplicaciones que usen el modelo cliente-servidor son el Correo electrónico, un Servidor de impresión y la World Wide Web.

En esta arquitectura la capacidad de proceso está repartida entre los clientes y los servidores, aunque son más importantes las ventajas de tipo organizativo debidas a la centralización de la gestión de la información y la separación de responsabilidades, lo que facilita y clarifica el diseño del sistema.

La separación entre cliente y servidor es una separación de tipo lógico, donde el servidor no se ejecuta necesariamente sobre una sola máquina ni es necesariamente un sólo programa. Los tipos específicos de servidores incluyen los servidores web, los servidores de archivo, los servidores del correo, los servidores de bases de datos, etc. Mientras que sus propósitos varían de unos servicios a otros, la arquitectura básica seguirá siendo la misma.

La red cliente-servidor es una red de comunicaciones en la cual los clientes están conectados a un servidor, en el que se centralizan los diversos recursos y aplicaciones con que se cuenta; y que los pone a disposición de los clientes cada vez que estos son solicitados. Esto significa que todas las gestiones que se realizan se concentran en el servidor, de manera que en él se disponen los requerimientos provenientes de los clientes que tienen prioridad, los archivos que son de uso público y los que son de uso restringido, los archivos que son de sólo lectura y los que, por el contrario, pueden ser modificados, etc.



En la relación cliente-servidor, el cliente solicita que se realice cierta acción, y el servidor realiza la acción y responde

www.profmatiasgarcia.com.ar

al cliente. Una implementación común del modelo de petición-respuesta se da entre los navegadores y servidores Web. Cuando un usuario selecciona un sitio Web para navegar mediante un navegador (la aplicación cliente), se envía una petición al servidor Web apropiado (la aplicación servidor). Por lo general, el servidor responde al cliente enviando una página Web en HTML apropiada.

Ventajas:

- Centralización del control de los recursos, datos y accesos.
- Facilidad de mantenimiento y actualización del lado del servidor. Por ejemplo, una actualización se aplica a un único servidor, pero los beneficios los obtienen múltiples clientes generalmente sin necesidad de que éstos actualicen nada.
- Escalabilidad: se puede aumentar la capacidad de clientes y servidores por separado. Cualquier elemento puede ser aumentado (o mejorado) en cualquier momento, o se pueden añadir nuevos nodos a la red (clientes y/o servidores).
- Toda la información es almacenada en el lado del servidor, que suele tener mayor seguridad que los clientes.
- Hay muchas herramientas cliente-servidor probadas, seguras y amigables para usar.

Desventajas:

- Si el número de clientes simultáneos es elevado, el servidor puede saturarse. Cuando una gran cantidad de clientes envían peticiones simultáneas al mismo servidor, puede ser que cause muchos problemas para éste (a mayor número de clientes, más problemas para el servidor).
- Frente a fallas del lado del servidor, el servicio queda paralizado para los clientes. Cuando un servidor está caído, las peticiones de los clientes no pueden ser satisfechas.
- El software y el hardware de un servidor son generalmente muy determinantes. Un hardware regular de una computadora personal puede no poder servir a cierta cantidad de clientes. Normalmente se necesita software y hardware específico, sobre todo en el lado del servidor, para satisfacer el trabajo. Por supuesto, esto aumentará el costo de todo el sistema.
- El cliente no dispone de los recursos que puedan existir en el servidor. Por ejemplo, si la aplicación es una Web, no podemos escribir en el disco duro del cliente o imprimir directamente sobre las impresoras sin sacar antes la ventana previa de impresión de los navegadores.

La comunicación entre un cliente y un servidor se puede realizar de dos formas distintas mediante conexiones (servicio orientado a conexión) o con datagramas (servicio sin conexión).

- Protocolo orientado a conexión: se basa en una conexión establecida en la que queda fijado el destino de la conexión desde el momento de abrirla. Esta conexión permanecerá hasta que se cierre, como ocurre con una llamada telefónica. Este protocolo en TCP/IP se llama TCP (Transmission Control Protocol).
- Protocolo sin conexión (datagramas): por el contrario envía mensajes individuales con el destino grabado en cada uno de ellos, como en una comunicación a través de correo postal, donde no son

www.profmatiasgarcia.com.ar

necesarias las fases de establecimiento y liberación de conexión alguna. Este protocolo en TCP/IP se llama UDP (User Datagram Protocol).

En este modelo de aplicaciones, el servidor suele tener un papel pasivo, respondiendo únicamente a las peticiones de los clientes, mientras que estos últimos son los que interactúan con los usuarios (y disponen de interface de usuario).

7.2 SOCKETS

Son la base de la programación en red. Socket designa un concepto abstracto por el cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada. Se trata de el conjunto de una dirección de servidor y un número de puerto. Esto posibilita la comunicación entre un cliente y un servidor a través del puerto del socket. Para ello el servidor tiene que estar escuchando por ese puerto.

Para ello habrá al menos dos aplicaciones en ejecución: una en el servidor que es la que abre el socket a la escucha, y otra en el cliente que hace las peticiones en el socket.

Normalmente la aplicación servidor ejecuta varias instancias de sí misma para permitir la comunicación con varios clientes a la vez.

En JAVA, las herramientas de red fundamentales se declaran mediante clases e interfaces del paquete `java.net`, mediante el cual JAVA ofrece comunicaciones basadas en flujos que permiten a las aplicaciones ver las redes como flujos de datos. Estas clases e interfaces también ofrecen comunicaciones basadas en paquetes, para transmitir paquetes individuales de información; esto se utiliza comúnmente para transmitir audio y video a través de Internet.

Las comunicaciones basadas en sockets permiten a las aplicaciones ver las redes como si fueran E/S de archivo; un programa puede leer de un socket o escribir en un socket de una manera tan simple como leer o escribir en un archivo. El socket es simplemente una construcción de software que representa un extremo de una conexión.

Con los sockets de flujo, un proceso establece una conexión con otro proceso. Mientras la conexión está en pie, los datos fluyen entre los procesos en flujos continuos. Se dice que los sockets de flujo proporcionan un servicio orientado a la conexión. El protocolo utilizado para la transmisión es el popular TCP (Protocolo de control de transmisión).

El paquete `java.net` proporciona tres clases: **`Socket`**, **`ServerSocket`** y **`DatagramSocket`**.

- **`Socket`** Implementa un extremo de la conexión (TCP) a través de la cual se realiza la comunicación.
- **`ServerSocket`** Implementa el extremo Servidor de la conexión (TCP) en la cual se esperan las conexiones de clientes.
- **`DatagramSocket`** Implementa tanto el extremo servidor como el cliente de UDP.

www.profmatiasgarcia.com.ar

El sistema operativo creará un punto de conexión en el protocolo de nivel 4 (nivel de transporte) correspondiente, en el caso de internet (arquitectura TCP/IP) las conexiones serán soportadas por el protocolo TCP (si el tipo es **`Socket`** fiable y los datagramas (**`DatagramSocket`**) por el protocolo UDP no fiable.

Concepto de fiabilidad:

1. No se perderán paquetes.
2. No llegarán paquetes duplicados.
3. El orden de recepción será el mismo que el de emisión.

Si se emplea un protocolo fiable, el programa de aplicación quedará liberado de la responsabilidad de gestionar reordenamientos, pérdidas y duplicados. A cambio, el software de protocolos interno necesitará un mayor tiempo de proceso.

CLIENTES

Las aplicaciones clientes son las que se comunican con servidores mediante un socket.

Se abre un puerto de comunicación en la PC del cliente hacia un servidor cuya dirección ha de ser conocida.

La clase que permite esta comunicación es la clase `java.net.Socket`.

Constructor	Uso
`Socket(String servidor, int puerto) throws IOException, UnknownHostException`	Crea un nuevo socket hacia el servidor utilizando el puerto indicado
`Socket(InetAdress servidor, int puerto) throws IOException`	Como el anterior, sólo que el servidor se establece con un objeto InetAddress
`Socket(InetAdress servidor, int puerto, InetAdress dirLocal, int puertoLocal) throws IOException`	Crea un socket hacia el servidor y puerto indicados, pero la lectura la realiza la dirección local y puerto local establecidos.
`Socket(String servidor, int puerto, InetAdress dirLocal, int puertoLocal) throws IOException, UnknownHostException`	Crea un socket hacia el servidor y puerto indicados, pero la lectura la realiza la dirección local y puerto local establecidos.

```
try {
    Socket s = new Socket("time-a.profmatiasgarcia.com.ar", 13);
} catch (UnknownHostException une) {
    System.out.println("No se encuentra el servidor");
} catch (IOException une) {
    System.out.println("Error en la comunicación");
}
```

www.profmatiasgarcia.com.ar

LECTURA Y ESCRITURA POR EL SOCKET

El hecho de establecer comunicación mediante un socket con un servidor, posibilita el envío y la recepción de datos. Esto se realiza con las clases de entrada y salida. La clase **Socket** proporciona estos métodos:

Metodo	Uso
`InputStream getInputStream() throws IOException`	Obtiene la corriente de entrada de datos para el socket
`OutputStream getOutputStream() throws IOException`	Obtiene la corriente de salida de datos para el socket

Se puede observar como se obtienen objetos de entrada y salida orientados al byte. Si la comunicación entre servidor y cliente se realiza mediante cadenas de texto (algo muy habitual) entonces se suele convertir en objetos de tipo **BufferedReader** para la entrada (como se hacía con el teclado) y objetos de tipo **PrintWriter** (salida de datos en forma de texto):

```
try {
    Socket s2 = new Socket("servidor.falso.com.ar", 7633);
    BufferedReader in = new BufferedReader(new
InputStreamReader(s2.getInputStream()));
    PrintWriter out = new PrintWriter(s2.getOutputStream(), true); // el parámetro
true sirve para volcar la salida al dispositivo de salida (autoflush)
    boolean salir = false;
    do {
          String s = in.readLine();
          if (s != null)
                System.out.println(s);
          else
                salir = true;
    } while (!salir);
} catch (UnknownHostException une) {
    System.out.println("No se encuentra el servidor");
} catch (IOException une) {
    System.out.println("Error en la comunicación");
}
```

SERVIDORES

El programa servidor se ocupa de recibir el fuljo de datos que procede del socket del cliente (además tiene que procurar servir a varios clientes a la vez).

Para que un programa abra un socket de servidor, se usa la clase **ServerSocket** cuyo constructor permite indicar el puerto que se abre:

```
ServerSocket socketparaServer = new ServerSocket(7582);
```

Después se tiene que crear un socket para atender a los clientes. Para ello hay un método llamado accept que espera que el servidor atienda a los clientes. Este método obtiene un objeto **Socket** para comunicarse con el cliente. Ejemplo:

```
try {
      ServerSocket s = new ServerSocket(8189);
      Socket recepcion = s.accept();
```

www.profmatiasgarcia.com.ar

```
        // El servidor espera hasta que llegue un cliente
        BufferedReader in = new BufferedReader(new
InputStreamReader(recepcion.getInputStream()));
        PrintWriter out = new PrintWriter(recepcion.getOutputStream(), true);
        out.println("Hola! Introduzca ADIOS para salir");
        boolean done = false;
        while (!done) {
                String linea = in.readLine();
                if (linea == null)
                        done = true;
                else {
                        out.println("Echo: " + linea);
                        if (linea.trim().equals("ADIOS"))
                                done = true;
                }
        }
        recepcion.close();
} catch (IOException e) {
}
```

Este es un servidor que acepta texto de entrada y lo repite hasta que el usuario escribe ADIOS. Al final la conexión del cliente se cierra con el método close de la clase Socket.

SERVIDOR DE MÚLTIPLES CLIENTES

Para poder escuchar a más de un cliente a la vez se utilizan threads. Lanzado un thread cada vez que llega un cliente y hay que manipularlo. En el thread el método **run** contendrá las instrucciones de comunicación con el cliente. En este tipo de servidores, el servidor se queda a la espera de clientes. cada vez que llega un cliente, le asigna un Thread para él. Por lo que se crea una clase derivada de la clase **Thread** que es la encargada de atender a los clientes.

Es decir, el servidor sería por ejemplo:

```
public class Servidor_Threads {
    public static void main(String args[]) {
            try {
                    // Se crea el servidor de sockets
                    ServerSocket servidor = new ServerSocket(8347);
                    while (true) {// bucle infinito
                            // El servidor espera al cliente siguiente y le
                            // asigna un nuevo socket
                            Socket socket = servidor.accept();
                            // se crea el Thread cliente y se le pasa el socket
                            Cliente_Threads c = new Cliente_Threads(socket);
                            c.start();// se lanza el Thread
                    }
            } catch (IOException e) {
            }
    }
}
```

www.profmatiasgarcia.com.ar

Por su parte el cliente tendría este código:

```
public class Cliente_Threads extends Thread {
      private Socket socket;

      public Cliente_Threads(Socket s) {
            socket = s;
      }

      public void run() {
            try {
                  // Obtención de las corrientes de datos
                  BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                  PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                  out.println("Bienvenido");
                  boolean salir = false;// controla la salida
                  while (!salir) {
                        String resp = in.readLine();// lectura
                        // proceso de datos de lectura
                        out.println("....");// datos de salida
                        if (resp == "BYE")
                              salir = true;// condición de salida
                  }
                  out.println("ADIOOOOOOS");
                  socket.close();
            } catch (Exception e) {
            }
      }
}
```

CLASE INETADDRESS

Obtiene un objeto que representa una dirección de Internet. Para ello se puede emplear este código:

```
InetAddress dirección = InetAddress.getByName("www.profmatiasgarcia.com.ar");
System.out.println(dirección.getHostAddress()); //imprime: 173.237.190.67
```

Otra forma de obtener la dirección es usar el método getAddress que devuelve un array de bytes. Cada byte representa un número de la dirección. A veces un servidor tiene varios nombres. Para ello se usa:

```
InetAddress[] nombres = InetAddress.getAllByName("www.google.com");
System.out.println(nombres.length);
for (int i = 0; i < nombres.length; i++) {
      System.out.println(nombres[i].getHostAddress());
}
//imprime:
//2
//172.217.28.164
//2800:3f0:4002:804:0:0:0:2004
```

www.profmatiasgarcia.com.ar

Método	Uso
`static InetAddress getByName(String servidor)`	Obtiene la dirección IP en forma de array de bytes
`static InetAddress getAllByName (String servidor)`	Obtiene la dirección IP en forma de array de bytes
`static InetAddress getByAddress (byte[] direcciónIP)`	Obtiene el objeto InetAddress asociado a esa dirección IP
`static InetAddress getLocalHostName()`	Obtiene el objeto InetAddress que corresponde al servidor actual
`String getHostAddress()`	Obtiene la dirección IP en forma de array de bytes
`byte[] getAddress()`	Obtiene la dirección IP en forma de array de bytes
`String getHostName()`	Obtiene el nombre del servidor
`String getCanonicalHostName()`	Obtiene el nombre canónigo completo (suele ser la dirección real del host

CONEXIONES URL

Realizar conexiones mediante sockets tiene el inconveniente de que hay que conocer las conexiones a un nivel de funcionamiento bastante bajo. Por eso se utilizan también conexiones a nivel más alto mediante objetos URL

Un objeto URL representa una dirección alcanzable de una red TCP/IP. Su construcción se suele realizar de esta forma:

```
URL url = new URL("http://www.profmatiasgarcia.com.ar");
```

El método *openStream* obtiene un objeto **InputStream** a través del cual se puede obtener el contenido.

Ejemplo:

```
try {
    URL url = new URL("http://www.profmatiasgarcia.com.ar");
    BufferedReader in = new BufferedReader(new
InputStreamReader(url.openStream()));
    String linea;
    while ((linea = in.readLine()) != null) {
          System.out.println(linea);
    }
} catch (MalformedURLException mue) {
    System.out.println("URL no válida");
} catch (IOException ioe) {
    System.out.println("Error en la comunicación");
}
```

El ejemplo lee la página de inicio de www.profmatiasgarcia.com.ar. Como se ve se captura una excepción de tipo **MalformedURLException** esta excepción se requiere capturar al construir el nuevo objeto URL. Se trata de una excepción hija de **IOException** (que en el ejemplo también es capturada).

www.profmatiasgarcia.com.ar

CONEXIONES URLCONNECTION

Los objetos **URLConnection** permiten establecer comunicaciones más detalladas con los servidores. Los pasos son:

1> Obtener el objeto de conexión con el método **openConnection** de la clase URL

2> Establecer propiedades para comunicar:

Método	Uso
`void setDoInput(boolean b)`	Permite que el usuario reciba datos desde la URL si b es true (por defecto está establecido a true)
`void setDoOutput(boolean b)`	Permite que el usuario envíe datos si b es true (éste no está establecido al principio)
`void setIfModifiedSince(long tiempo)`	Sólo muestra recursos con fecha posterior a la dada (la fecha se da en milisegundos a partir de 1970, el método getTime de la clase Date consigue este dato).
`void setUseCaches(boolean b)`	Permite recuperar datos desde un caché
`Void setAllowUserInteraction (boolean b)`	Permite solicitar contraseña al usuario. Esto lo debe realizar un programa externo, lo cuál no tiene efecto fuera de un applet (en un navegador, el navegador se encarga de sacar el cuadro).
`void RequestProperty(String clave, String valor)`	Establece un campo de cabecera

3> Conectar (método connect)

Método	Uso
`void connect()`	Conecta con el recurso remoto y recupera información de la cabecera de respuesta

4> Solicitar información de cabecera

Método	Uso
`String getHeaderFieldKey(int n)`	Obtiene el campo clave número n de la cabecera de respuesta
`String getHeaderField(int n)`	Obtiene el valor de la clave número n
`long getDate()`	Fecha del recurso
`long getExpiration()`	Obtiene la fecha de expiración del recurso
`long getLastModifier()`	Fecha de última modificación

5> Obtener la información del recurso

Método	Uso
`InputStream openInputStream()`	Obtiene un flujo para recibir datos desde el servidor (es igual que el método openStream de la clase URL)
`OutputStream openOutputStream()`	Abre un canal de salida hacia el servidor

www.profmatiasgarcia.com.ar

Ejemplo:

```
try {
    URL url = new URL("http://www.profmatiasgarcia.com.ar");
    URLConnection conexión = url.openConnection();
    conexión.setDoOutput(true);
    conexión.connect();
    // Lectura y muestra de los encabezados
    int i = 1;
    while (conexión.getHeaderFieldKey(i) != null) {
          System.out.println(conexión.getHeaderFieldKey(i) + ":" +
conexión.getHeaderField(i));
          i++;
    }
    // Lectura y muestra del contenido
    BufferedReader in = new BufferedReader(new
InputStreamReader(conexión.getInputStream()));
    String s = in.readLine();
    while (s != null) {
          System.out.println(s);
          s = in.readLine();
    }
} catch (Exception e) {
    System.out.println("Fallo");
}
```

7.4 Cómo establecer un cliente simple utilizando sockets de flujo

Para establecer un cliente simple en JAVA se requieren cuatro pasos.

En el paso 1 creamos un objeto **Socket** para conectarse al servidor. El constructor de Socket establece la conexión al servidor. Por ejemplo, la instrucción:

```
Socket conexion = new Socket(direccionServidor, puerto);
```

utiliza el constructor de **Socket** con dos argumentos: la dirección del servidor y el número de puerto. Si el intento de conexión es exitoso, esta instrucción devuelve un objeto **Socket**. Un intento de conexión fallido lanzará una instancia de una subclase de **IOException**. Una excepción **UnknownHostException** ocurre específicamente cuando el sistema no puede resolver la dirección del servidor especificada en la llamada al constructor de **Socket** en una dirección IP que corresponda.

En el paso 2, el cliente utiliza los métodos getInputStream y getOutputStream de la clase **Socket** para obtener referencias a los objetos **InputStream** y **OutputStream** de **Socket**. Podemos utilizar las técnicas para envolver otros tipos de flujos alrededor de los objetos **InputStream** y **OutputStream** asociados con el objeto **Socket**. Si el servidor va a enviar información en el formato de los tipos actuales, el cliente debe recibir esa información en el mismo formato. Por lo tanto, si el servidor envía los valores con un objeto **ObjectOutputStream**, el cliente debe leer esos valores con un objeto **ObjectInputStream**.

El paso 3 es la fase de procesamiento, en la cual el cliente y el servidor se comunican a través de los objetos **InputStream** y **OutputStream**.

www.profmatiasgarcia.com.ar

En el paso 4, el cliente cierra la conexión cuando se completa la transmisión, invocando al método `close` en los flujos y en el objeto `Socket`. El cliente debe determinar cuándo va a terminar el servidor de enviar información, de manera que pueda llamar al método `close` para cerrar la conexión del objeto `Socket`. Por ejemplo, el método `read` de `InputStream` devuelve el valor –1 cuando detecta el fin del flujo (lo que se conoce también como EOF: fin del archivo). Si se utiliza un objeto `ObjectInputStream` para leer información del servidor, se produce una excepción `EOFException` cuando el cliente trata de leer un valor de un flujo en el que se detectó el fin del flujo.

7.3 Cómo establecer un servidor simple utilizando sockets de flujo

Para establecer un servidor simple en JAVA se requieren cinco pasos.

El paso 1 es crear un objeto `ServerSocket`. Una llamada al constructor de *`ServerSocket`* como:

```
ServerSocket servidor = new ServerSocket(numeroPuerto, longitudCola);
```

registra un número de puerto TCP disponible y específica el máximo número de clientes que pueden esperar para conectarse al servidor (es decir, la longitud de la cola). El número de puerto es utilizado por los clientes para localizar la aplicación servidor en el equipo servidor. A menudo, a esto se le conoce como punto de negociación (handshake). Si la cola está llena, el servidor rechaza las conexiones de los clientes. El constructor establece el puerto en donde el servidor espera las conexiones de los clientes; a este proceso se le conoce como enlazar el servidor al puerto. Cada cliente pedirá conectarse con el servidor en este puerto. Sólo una aplicación puede enlazarse a un puerto específico en el servidor, en un momento dado.

Los números de puerto pueden ser entre 0 y 65,535. Algunos sistemas operativos reservan los números de puertos menores que 1024 para los servicios del sistema (como los servidores de e-mail y World Wide Web). Por lo general, estos puertos no deben especificarse como puertos de conexión en los programas de los usuarios. De hecho, algunos sistemas operativos requieren de privilegios de acceso especiales para enlazarse a los números de puerto menores que 1024.

Los programas administran cada conexión cliente mediante un objeto `Socket`.

En el paso 2, el servidor escucha indefinidamente (o bloquea) para esperar a que un cliente trate de conectarse. Para escuchar una conexión de un cliente, el programa llama al método *`accept`* de `ServerSocket`, como se muestra a continuación:

```
Socket conexion = servidor.accept();
```

esta instrucción devuelve un objeto `Socket` cuando se establece la conexión con un cliente. El objeto `Socket` permite al servidor interactuar con el cliente. Las interacciones con el cliente ocurren realmente en un puerto del servidor distinto al del punto de negociación. De esta forma, el puerto especificado en el paso 1 puede utilizarse nuevamente en un servidor con subprocesamiento múltiple, para aceptar otra conexión cliente.

www.profmatiasgarcia.com.ar

El paso 3 es obtener los objetos **OutputStream** e **InputStream** que permiten al servidor comunicarse con el cliente, enviando y recibiendo bytes. El servidor envía información al cliente mediante un objeto **OutputStream** y recibe información del cliente mediante un objeto **InputStream**. El servidor invoca al método *getOutputStream* en el objeto **Socket** para obtener una referencia al objeto **OutputStream** del objeto **Socket**, e invoca al método *getInputStream* en el objeto **Socket** para obtener una referencia al objeto **InputStream** del objeto **Socket**.

Los objetos flujo pueden utilizarse para enviar o recibir bytes individuales, o secuencias de bytes, mediante el método *write* de **OutputStream** y el método *read* de **InputStream**, respectivamente. A menudo es útil enviar o recibir valores de tipos primitivos (como **int** y **double**) u objetos Serializable (como objetos **String** u otros tipos serializables), en vez de enviar bytes. En este caso podemos utilizar las técnicas para envolver otros tipos de flujos (como **ObjectOutputStream** y **ObjectInputStream**) alrededor de los objetos **OutputStream** e **InputStream** asociados con el objeto **Socket**. Por ejemplo,

```
ObjectInputStream entrada = new ObjectInputStream(conexion.getInputStream() );
ObjectOutputStream salida = new ObjectOutputStream(conexion.getOutputStream() );
```

Lo mejor de establecer estas relaciones es que, cualquier cosa que escriba el servidor en el objeto **ObjectOutputStream** se enviará mediante el objeto **OutputStream** y estará disponible en el objeto **InputStream** del cliente, y cualquier cosa que el cliente escriba en su objeto **OutputStream** (mediante su correspondiente objeto **ObjectOutputStream**) estará disponible a través del objeto **InputStream** del servidor. La transmisión de los datos a través de la red es un proceso transparente, y se maneja completamente mediante JAVA.

El paso 4 es la fase de procesamiento, en la cual el servidor y el cliente se comunican a través de los objetos **OutputStream** e **InputStream**.

En el paso 5, cuando se completa la transmisión, el servidor cierra la conexión invocando al método **close** en los flujos y en el objeto **Socket**.

Con los sockets, la E/S de red es vista por los programas de JAVA como algo similar a un archivo de E/S secuencial. Los sockets ocultan al programador gran parte de la complejidad de la programación en red.

7.5 Interacción entre cliente/servidor mediante conexiones de socket de flujo

El ejemplo utiliza sockets de flujo para demostrar una aplicación cliente/servidor para conversar simple. El servidor espera un intento de conexión por parte de un cliente. Cuando se conecta un cliente al servidor, la aplicación servidor envía un objeto **String** al cliente (recuerde que los objetos String son objetos Serializable), indicando que la conexión con el cliente fue exitosa. Después el cliente muestra el mensaje.

Ambas aplicaciones cliente y servidor proporcionan campos de texto que permiten al usuario escribir un mensaje y enviarlo de una a otra aplicación. Cuando el cliente o el servidor envían la cadena

www.profmatiasgarcia.com.ar

"TERMINAR", la conexión entre el cliente y el servidor termina. Después el servidor espera a que se conecte otro cliente.

La clase Servidor

El constructor de Servidor crea la GUI del servidor, la cual contiene un objeto **JTextField** y un objeto **JTextArea**. Servidor muestra su salida en el objeto **JTextArea**. Cuando se ejecuta el método main crea un objeto Servidor, especifica la operación de cierre predeterminada de la ventana y llama al método ejecutarServidor.

El método **ejecutarServidor** configura el servidor para que reciba una conexión y procese una conexión a la vez. Se crea un objeto **ServerSocket** llamado servidor, para que espere las conexiones. El objeto **ServerSocket** escucha en el puerto 12345, en espera de que un cliente se conecte. El segundo argumento para el constructor es el número de conexiones que pueden esperar en una cola para conectarse al servidor (100 en este ejemplo). Si la cola está llena cuando un cliente trate de conectarse, el servidor rechazará la conexión.

```
// Establece un servidor que recibe una conexión de un cliente, envía una cadena
al cliente y cierra la conexión.
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 Servidor extends JFrame {
   private JTextField campoIntroducir; // recibe como entrada un mensaje del
usuario
   private JTextArea areaPantalla; // muestra información al usuario
   private ObjectOutputStream salida; // flujo de salida hacia el cliente
   private ObjectInputStream entrada; // flujo de entrada del cliente
   private ServerSocket servidor; // socket servidor
   private Socket conexion; // conexión al cliente
   private int contador = 1; // contador del número de conexiones
   // establece la GUI
   public Servidor() {
         super("Servidor");
         campoIntroducir = new JTextField(); // crea objeto campoIntroducir
         campoIntroducir.setEditable(false);
         campoIntroducir.addActionListener(new ActionListener() {
               // envía un mensaje al cliente
               public void actionPerformed(ActionEvent evento) {
                     enviarDatos(evento.getActionCommand());
                     campoIntroducir.setText("");
               } // fin del método actionPerformed
         } // fin de la clase interna anónima
         ); // fin de la llamada a addActionListener
         add(campoIntroducir, BorderLayout.NORTH);
```

```
            areaPantalla = new JTextArea(); // crea objeto areaPantalla
            add(new JScrollPane(areaPantalla), BorderLayout.CENTER);
            setSize(300, 150); // establece el tamaño de la ventana
            setVisible(true); // muestra la ventana
      } // fin del constructor de Servidor

      // establece y ejecuta el servidor
      public void ejecutarServidor() {
            try // establece el servidor para que reciba conexiones; procesa las
conexiones
            {
                  servidor = new ServerSocket(12345, 100); // crea objeto
ServerSocket
                  while (true) {
                        try {
                              esperarConexion(); // espera una conexión
                              obtenerFlujos(); // obtiene los flujos de entrada y
salida
                              procesarConexion(); // procesa la conexión
                        } // fin de try
                        catch (EOFException excepcionEOF) {
                              mostrarMensaje("\nServidor termino la conexion");
                        } // fin de catch
                        finally {
                              cerrarConexion(); // cierra la conexión
                              contador++;
                        } // fin de finally
                  } // fin de while
            } // fin de try
            catch (IOException exepcionES) {
                  exepcionES.printStackTrace();
            } // fin de catch
      } // fin del método ejecutarServidor


      // espera a que llegue una conexión, después muestra información sobre ésta
      private void esperarConexion() throws IOException {
            mostrarMensaje("Esperando una conexion\n");
            conexion = servidor.accept(); // permite al servidor aceptar la conexión
            mostrarMensaje("Conexion " + contador + " recibida de: " +
conexion.getInetAddress().getHostName());
      } // fin del método esperarConexion


      // obtiene flujos para enviar y recibir datos
      private void obtenerFlujos() throws IOException {
            // establece el flujo de salida para los objetos
            salida = new ObjectOutputStream(conexion.getOutputStream());
            salida.flush(); // vacía el búfer de salida para enviar información del
encabezado
            // establece el flujo de entrada para los objetos
            entrada = new ObjectInputStream(conexion.getInputStream());
            mostrarMensaje("\nSe obtuvieron los flujos de E/S\n");
      } // fin del método obtenerFlujos

      // procesa la conexión con el cliente
      private void procesarConexion() throws IOException {
            String mensaje = "Conexion exitosa";
            enviarDatos(mensaje); // envía mensaje de conexión exitosa
            // habilita campoIntroducir para que el usuario del servidor pueda
            // enviar mensajes
```

www.profmatiasgarcia.com.ar

```
            setTextFieldEditable(true);
            do // procesa los mensajes enviados desde el cliente
            {
                  try // lee el mensaje y lo muestra en pantalla
                  {
                        mensaje = (String) entrada.readObject(); // lee el nuevo
mensaje
                        mostrarMensaje("\n" + mensaje); // muestra el mensaje
                  } // fin de try
                  catch (ClassNotFoundException excepcionClaseNoEncontrada) {
                        mostrarMensaje("\nSe recibio un tipo de objeto
desconocido");
                  } // fin de catch
            } while (!mensaje.equals("CLIENTE>>> TERMINAR"));
      } // fin del método procesarConexion

      // cierra flujos y socket
      private void cerrarConexion() {
            mostrarMensaje("\nTerminando conexion\n");
            setTextFieldEditable(false); // deshabilita campoIntroducir
            try {
                  salida.close(); // cierra flujo de salida
                  entrada.close(); // cierra flujo de entrada
                  conexion.close(); // cierra el socket
            } // fin de try
            catch (IOException exepcionES) {
                  exepcionES.printStackTrace();
            } // fin de catch
      } // fin del método cerrarConexion

      // envía el mensaje al cliente
      private void enviarDatos(String mensaje) {
            try // envía objeto al cliente
            {
                  salida.writeObject("SERVIDOR>>> " + mensaje);
                  salida.flush(); // envía toda la salida al cliente
                  mostrarMensaje("\nSERVIDOR>>> " + mensaje);
            } // fin de try
            catch (IOException exepcionES) {
                  areaPantalla.append("\nError al escribir objeto");
            } // fin de catch
      } // fin del método enviarDatos

      // manipula areaPantalla en el subproceso despachador de eventos
      private void mostrarMensaje(final String mensajeAMostrar) {
            SwingUtilities.invokeLater(new Runnable() {
                  public void run() // actualiza areaPantalla
                  {
                        areaPantalla.append(mensajeAMostrar); // adjunta el mensaje
                  } // fin del método run
            } // fin de la clase interna anónima
            ); // fin de la llamada a SwingUtilities.invokeLater
      } // fin del método mostrarMensaje manipula a campoIntroducir en el

      // subproceso despachador de eventos
      private void setTextFieldEditable(final boolean editable) {
            SwingUtilities.invokeLater(new Runnable() {
                  public void run() // establece la propiedad de edición de
campoIntroducir
                  {
                        campoIntroducir.setEditable(editable);
```

www.profmatiasgarcia.com.ar

```
                } // fin del método
            } // fin de la clase interna
            ); // fin de la llamada a SwingUtilities.invokeLater
    } // fin del método setTextFieldEditable
} // fin de la clase Servidor
```

Principal

```
import javax.swing.JFrame;

public class PruebaServidor {
    public static void main(String args[]) {
            Servidor aplicacion = new Servidor(); // crea el servidor
            aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            aplicacion.ejecutarServidor(); // ejecuta la aplicación servidor
    } // fin de main
} // fin de la clase PruebaServidor
```

En el método **esperarConexion** se utiliza el método *accept* de **ServerSocket** para esperar una conexión de un cliente. Al ocurrir una conexión, el objeto **Socket** resultante se asigna a conexion. El método *accept* realiza un bloqueo hasta que se reciba una conexión (es decir, el subproceso en el que se haga la llamada a *accept* detiene su ejecución hasta que un cliente se conecte).

El método *getInetAddress* de **Socket** devuelve un objeto **InetAddress** (paquete java.net), el cual contiene información acerca del equipo cliente. El método *getHostName* de **InetAddress** devuelve el nombre de host del equipo cliente. Por ejemplo, hay una dirección IP (127.0.0.1) y nombre de host (localhost) especiales, que son útiles para probar aplicaciones de red en su equipo local [a ésta también se le conoce como dirección de bucle local (loopback)]. Si se hace una llamada a *getHostName* en un objeto **InetAddress** que contenga 127.0.0.1, el nombre de host correspondiente que devuelve el método es localhost.

El método **obtenerFlujos** obtiene las referencias a los flujos de **Socket** y los utiliza para inicializar un objeto **ObjectOutputStream** y un objeto **ObjectInputStream**, respectivamente. El método *flush* de **ObjectInputStream** hace que el objeto **ObjectOutputStream** en el servidor envíe un encabezado de flujo al objeto **ObjectInputStream** correspondiente del cliente. El encabezado de flujo contiene información como la versión de la serialización de objetos que se va a utilizar para enviar los objetos. Esta información es requerida por el objeto **ObjectInputStream**, para que pueda prepararse para recibir estos objetos de manera correcta.

En el método **procesarConexion** se hace una llamada al método **enviarDatos** para enviar la cadena "SERVIDOR>>> Conexión exitosa" al cliente. Se utiliza el método *readObject* de **ObjectInputStream** para leer un objeto String del cliente y el método **mostrarMensaje** para anexar el mensaje al objeto **JTextArea**.

Cuando termina la transmisión, el método **procesarConexion** regresa y el programa llama al método **cerrarConexion** para cerrar los flujos asociados con el objeto **Socket**, y para cerrar también a ese objeto **Socket**. Después, el servidor espera el siguiente intento de conexión por parte de un cliente.

www.profmatiasgarcia.com.ar

Cuando el usuario de la aplicación servidor introduce una cadena en el campo de texto y oprime la tecla *ENTER*, el programa llama al método `actionPerformed`, el cual lee la cadena del campo de texto y llama al método **enviarDatos** para enviar la cadena al cliente. El método **enviarDatos** escribe el objeto, vacía el búfer de salida y anexa la misma cadena al área de texto en la ventana del servidor. No es necesario invocar a **mostrarMensaje** para modificar el área de texto aquí, ya que el método **enviarDatos** es llamado desde un manejador de eventos; por lo tanto, **enviarDatos** se ejecuta como parte del subproceso despachador de eventos.

Observe que el objeto Servidor recibe una conexión, la procesa, cierra la conexión y espera a la siguiente conexión. Un escenario más apropiado sería un objeto Servidor que recibe una conexión, la configura para procesarla como un subproceso de ejecución separado, e inmediatamente se pone en espera de nuevas conexiones. Los subprocesos separados que procesan conexiones existentes pueden seguir ejecutándose, mientras que el Servidor se concentra en nuevas peticiones de conexión. Esto hace al servidor más eficiente, ya que pueden procesarse varias peticiones de clientes en forma concurrente.

La clase Cliente

Al igual que la clase Servidor, el constructor de la clase Cliente crea la GUI de la aplicación (un objeto **JTextField** y un objeto **JTextArea**). Cliente muestra su salida en el área de texto.

Cuando se ejecuta el método main, se crea una instancia de la clase Cliente, se especifica la operación de cierre predeterminada de la ventana y se hace una llamada al método **ejecutarCliente**. En este ejemplo, usted puede ejecutar el cliente desde cualquier computadora en Internet, y especificar la dirección IP o nombre de host del equipo servidor como un argumento de línea de comandos para el programa. Por ejemplo, el comando: `java Cliente 192.168.1.15` intenta realizar una conexión al objeto Servidor en la computadora que tiene la dirección IP 192.168.1.15.

El método **ejecutarCliente** de la clase Cliente establece la conexión con el servidor, procesa los mensajes recibidos del servidor y cierra la conexión cuando termina la comunicación. El método **conectarAlServidor** sirve para realizar la conexión.

Después de realizar la conexión, se hace una llamada al método **obtenerFlujos** para obtener las referencias a los objetos flujo del objeto **Socket**. Después se hace una llamada al método **procesarConexion** para recibir y mostrar los mensajes enviados por el servidor. En el bloque finally se hace una llamada al método **cerrarConexion** para cerrar los flujos y el objeto **Socket**, incluso aunque haya ocurrido una excepción. El método **mostrarMensaje** es llamado desde estos métodos para utilizar el subproceso despachador de eventos para mostrar los mensajes en el área de texto de la aplicación.

```
//Cliente que lee y muestra la información que se envía desde un Servidor.
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 Cliente extends JFrame {
    private JTextField campoIntroducir; // introduce la información del usuario
    private JTextArea areaPantalla; // muestra la información al usuario
    private ObjectOutputStream salida; // flujo de salida hacia el servidor
    private ObjectInputStream entrada; // flujo de entrada del servidor
    private String mensaje = ""; // mensaje del servidor
    private String servidorChat; // aloja al servidor para esta aplicación
    private Socket cliente; // socket para comunicarse con el servidor

    // inicializa el objeto servidorChat y establece la GUI
    public Cliente(String host) {
          super("Cliente");
          servidorChat = host; // establece el servidor al que se conecta este
cliente
          campoIntroducir = new JTextField(); // crea objeto campoIntroducir
          campoIntroducir.setEditable(false);
          campoIntroducir.addActionListener(new ActionListener() {
                // envía el mensaje al servidor
                public void actionPerformed(ActionEvent evento) {
                      enviarDatos(evento.getActionCommand());
                      campoIntroducir.setText("");
                } // fin del método actionPerformed
          } // fin de la clase interna anónima
          ); // fin de la llamada a addActionListener
          add(campoIntroducir, BorderLayout.NORTH);
          areaPantalla = new JTextArea(); // crea objeto areaPantalla
          add(new JScrollPane(areaPantalla), BorderLayout.CENTER);
          setSize(300, 150); // establece el tamaño de la ventana
          setVisible(true); // muestra la ventana
    } // fin del constructor de Cliente
          // se conecta al servidor y procesa los mensajes que éste envía

    // inicializa el objeto servidorChat y establece la GUI
    public void ejecutarCliente() {
          try // se conecta al servidor, obtiene flujos, procesa la conexión
          {
                conectarAlServidor(); // crea un objeto Socket para hacer la
conexión
                obtenerFlujos(); // obtiene los flujos de entrada y salida
                procesarConexion(); // procesa la conexión
          } // fin de try
          catch (EOFException excepcionEOF) {
                mostrarMensaje("\nCliente termino la conexion");
          } // fin de catch
          catch (IOException excepcionES) {
                excepcionES.printStackTrace();
          } // fin de catch
          finally {
                cerrarConexion(); // cierra la conexión
          } // fin de finally
    } // fin del método ejecutarCliente

    // se conecta al servidor
    private void conectarAlServidor() throws IOException {
          mostrarMensaje("Intentando realizar conexion\n");
          // crea objeto Socket para hacer conexión con el servidor
```

www.profmatiasgarcia.com.ar

```
            cliente = new Socket(InetAddress.getByName(servidorChat), 12345);
            // muestra la información de la conexión
            mostrarMensaje("Conectado a: " +
cliente.getInetAddress().getHostName());
      } // fin del método conectarAlServidor

      // obtiene flujos para enviar y recibir datos
      private void obtenerFlujos() throws IOException {
            // establece flujo de salida para los objetos
            salida = new ObjectOutputStream(cliente.getOutputStream());
            salida.flush(); // vacía el búfer de salida para enviar información de
encabezado
            // establece flujo de entrada para los objetos
            entrada = new ObjectInputStream(cliente.getInputStream());
            mostrarMensaje("\nSe obtuvieron los flujos de E/S\n");
      } // fin del método obtenerFlujos

      // procesa la conexión con el servidor
      private void procesarConexion() throws IOException {
            // habilita campoIntroducir para que el usuario cliente pueda enviar
mensajes
            establecerCampoEditable(true);
            do // procesa los mensajes que se envían desde el servidor
            {
                  try // lee el mensaje y lo muestra
                  {
                        mensaje = (String) entrada.readObject(); // lee nuevo
mensaje
                        mostrarMensaje("\n" + mensaje); // muestra el mensaje
                  } // fin de try
                  catch (ClassNotFoundException excepcionClaseNoEncontrada) {
                        mostrarMensaje("nSe recibio un tipo de objeto desconocido");
                  } // fin de catch
            } while (!mensaje.equals("SERVIDOR>>> TERMINAR"));
      } // fin del método procesarConexion

      // cierra flujos y socket
      private void cerrarConexion() {
            mostrarMensaje("\nCerrando conexion");
            establecerCampoEditable(false); // deshabilita campoIntroducir
            try {
                  salida.close(); // cierra el flujo de salida
                  entrada.close(); // cierra el flujo de entrada
                  cliente.close(); // cierra el socket
            } // fin de try
            catch (IOException excepcionES) {
                  excepcionES.printStackTrace();
            } // fin de catch
      } // fin del método cerrarConexion

      // envía un mensaje al servidor
      private void enviarDatos(String mensaje) {
            try // envía un objeto al servidor
            {
                  salida.writeObject("CLIENTE>>> " + mensaje);
                  salida.flush(); // envía todos los datos a la salida
                  mostrarMensaje("\nCLIENTE>>> " + mensaje);
            } // fin de try
            catch (IOException excepcionES) {
                  areaPantalla.append("\nError al escribir objeto");
            } // fin de catch
```

www.profmatiasgarcia.com.ar

```
    } // fin del método enviarDatos

    // manipula el objeto areaPantalla en el subproceso despachador de eventos
    private void mostrarMensaje(final String mensajeAMostrar) {
          SwingUtilities.invokeLater(new Runnable() {
                public void run() // actualiza objeto areaPantalla
                {
                      areaPantalla.append(mensajeAMostrar);
                } // fin del método run
          } // fin de la clase interna anónima
          ); // fin de la llamada a SwingUtilities.invokeLater
    } // fin del método mostrarMensaje

    // manipula a campoIntroducir en el subproceso despachador de eventos
    private void establecerCampoEditable(final boolean editable) {
          SwingUtilities.invokeLater(new Runnable() {
                public void run() // establece la propiedad de edición de
campoIntroducir
                {
                      campoIntroducir.setEditable(editable);
                } // fin del método run
          } // fin de la clase interna anónima
          ); // fin de la llamada a SwingUtilities.invokeLater
    } // fin del método establecerCampoEditable
} // fin de la clase Cliente
```

El método **conectarAlServidor** crea un objeto **Socket** llamado cliente para establecer una conexión. Este método pasa dos argumentos al constructor de **Socket**: la dirección IP del equipo servidor y el número de puerto (12345) en donde la aplicación servidor está esperando las conexiones de los clientes. En el primer argumento, el método static *getByName* de **InetAddress** devuelve un objeto **InetAddress** que contiene la dirección IP especificada como argumento en la línea de comandos para la aplicación (o 127.0.0.1 si no se especifican argumentos en la línea de comandos). El método *getByName* puede recibir una cadena que contiene la dirección IP actual, o el nombre de host del servidor. El primer argumento también podría haberse escrito de otras formas. Para la dirección 127.0.0.1 de localhost, el primer argumento podría especificarse mediante una de las siguientes expresiones:

```
InetAddress.getByName( "localhost" );
InetAddress.getLocalHost();
```

Principal

```
//Prueba la clase Cliente.
import javax.swing.JFrame;

public class PruebaCliente {
    public static void main(String args[]) {
          Cliente aplicacion; // declara la aplicación cliente
          // si no hay argumentos de línea de comandos
          if (args.length == 0)
                aplicacion = new Cliente("127.0.0.1"); // se conecta a localhost
          else
                aplicacion = new Cliente(args[0]); // usa args para conectarse
          aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          aplicacion.ejecutarCliente(); // ejecuta la aplicación cliente
    } // fin de main
} // fin de la clase PruebaCliente
```

El objeto Cliente utiliza un objeto **ObjectOutputStream** para enviar datos al servidor, y un objeto **ObjectInputStream** para recibir datos del servidor. El método **obtenerFlujos** crea los objetos **ObjectOutputStream** y **ObjectInputStream** que utilizan los flujos asociados con el socket del cliente.

El método **procesarConexion** contiene un ciclo que se ejecuta hasta que el cliente recibe el mensaje "SERVIDOR>>> TERMINAR". Cuando la transmisión termina, el método **cerrarConexion** cierra los flujos y el objeto Socket.

Cuando el usuario de la aplicación cliente introduce una cadena en el campo de texto y oprime la tecla *ENTER*, el programa llama al método *actionPerformed* para leer la cadena e invoca al método **enviarDatos** para enviar la cadena al servidor. El método **enviarDatos** escribe el objeto, vacía el búfer de salida y anexa la misma cadena al objeto **JTextArea** en la ventana del cliente. Una vez más, no es necesario invocar al método utilitario **mostrarMensaje** para modificar el área de texto aquí, ya que el método **enviarDatos** es llamado desde un manejador de eventos.

7.6 Interacción entre cliente/servidor sin conexión mediante datagramas

La transmisión orientada a la conexión es como el sistema telefónico en el que una persona marca y recibe una conexión al teléfono de la persona con la que desea comunicarse. La conexión se mantiene todo el tiempo que dure la llamada telefónica, incluso aunque no se esté hablando.

La transmisión sin conexión mediante datagramas es un proceso más parecido a la manera en que el correo se transporta mediante el servicio postal. Si un mensaje extenso no cabe en un sobre, usted lo divide en varias piezas separadas que coloca en sobres separados, numerados en forma secuencial. Cada una de las cartas se envía entonces por correo al mismo tiempo. Las cartas podrían llegar en orden, sin orden o tal vez no llegarían (aunque el último caso es raro, suele ocurrir). La persona en el extremo receptor debe reensamblar las piezas del mensaje en orden secuencial, antes de tratar de interpretarlo. Si su mensaje es lo suficientemente pequeño como para caber en un sobre, no tiene que preocuparse por el problema de que el mensaje esté “fuera de secuencia”, pero aún existe la posibilidad de que su mensaje no llegue. Una diferencia entre los datagramas y el correo postal es que pueden llegar duplicados de datagramas al equipo receptor.

La clase Servidor

La clase Servidor declara dos objetos **DatagramPacket** que son utilizados por el servidor para enviar y recibir información, y un objeto **DatagramSocket** que envía y recibe estos paquetes. El constructor de Servidor crea la interfaz gráfica de usuario en la que se mostrarán los paquetes de información.

A continuación, se crea el objeto **DatagramSocket** en un bloque try. Se utiliza el constructor de **DatagramSocket** que recibe un argumento entero para el número de puerto (5000 en este ejemplo) para enlazar el servidor a un puerto en donde pueda recibir paquetes de los clientes. Los objetos Cliente que envían paquetes a este objeto Servidor especifican el mismo número de puerto en los paquetes que envían. Si el constructor de **DatagramSocket** no puede enlazar el objeto **DatagramSocket** al puerto

especificado, se lanza una excepción **SocketException**.

```
//Servidor que recibe y envía paquetes desde/hacia un cliente.
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;

public class Servidor extends JFrame {
    private JTextArea areaPantalla; // muestra los paquetes recibidos
    private DatagramSocket socket; // socket para conectarse al cliente

    // establece la GUI y el objeto DatagramSocket
    public Servidor() {
          super("Servidor");
          areaPantalla = new JTextArea(); // crea objeto areaPantalla
          add(new JScrollPane(areaPantalla), BorderLayout.CENTER);
          setSize(400, 300); // establece el tamaño de la ventana
          setVisible(true); // muestra la ventana
          try // crea objeto DatagramSocket para enviar y recibir paquetes
          {
                socket = new DatagramSocket(5000);
          } // fin de try
          catch (SocketException excepcionSocket) {
                excepcionSocket.printStackTrace();
                System.exit(1);
          } // fin de catch
    } // fin del constructor de Servidor

    // espera a que lleguen los paquetes, muestra los datos y repite el paquete al
cliente
    public void esperarPaquetes() {
          while (true) {
                try // recibe el paquete, muestra su contenido, devuelve una copia
al cliente
                {
                      byte datos[] = new byte[100]; // establece un paquete
                      DatagramPacket paqueteRecibir = new DatagramPacket(datos,
datos.length);
                      socket.receive(paqueteRecibir); // espera a recibir el
paquete
                      // muestra la información del paquete recibido
                      mostrarMensaje("\nPaquete recibido:" + "\nDe host: " +
paqueteRecibir.getAddress() + "\nPuerto host: "
                            + paqueteRecibir.getPort() + "\nLongitud: " +
paqueteRecibir.getLength() + "\nContiene:\n\t"
                            + new String(paqueteRecibir.getData(), 0,
paqueteRecibir.getLength()));
                      enviarPaqueteAlCliente(paqueteRecibir); // envía el paquete
al

// cliente
                } // fin de try
                catch (IOException excepcionES) {
                      mostrarMensaje(excepcionES.toString() + "\n");
                      excepcionES.printStackTrace();
```

www.profmatiasgarcia.com.ar

```
                } // fin de catch
            } // fin de while
        } // fin del método esperarPaquetes

        // repite el paquete al cliente
        private void enviarPaqueteAlCliente(DatagramPacket paqueteRecibir) throws
    IOException {
            mostrarMensaje("\n\nRepitiendo datos al cliente...");
            // crea paquete para enviar
            DatagramPacket paqueteEnviar = new
    DatagramPacket(paqueteRecibir.getData(), paqueteRecibir.getLength(),
                        paqueteRecibir.getAddress(), paqueteRecibir.getPort());
            socket.send(paqueteEnviar); // envía paquete al cliente
            mostrarMensaje("Paquete enviado\n");
        } // fin del método enviarPaqueteAlCliente

        // manipula objeto areaPantalla en el subproceso despachador de eventos
        private void mostrarMensaje(final String mensajeAMostrar) {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() // actualiza areaPantalla
                {
                    areaPantalla.append(mensajeAMostrar); // muestra mensaje
                } // fin del método run
            } // fin de la clase interna anónima
            ); // fin de la llamada a SwingUtilities.invokeLater
        } // fin del método mostrarMensaje
    } // fin de la clase Servidor
```

Al especificar un puerto que ya esté en uso, o especificar un número de puerto incorrecto al crear un objeto **DatagramSocket** , se produce una excepción **SocketException** .

El método **esperarPaquetes** de la clase Servidor utiliza un ciclo infinito para esperar a que lleguen los paquetes al Servidor. El objeto **DatagramPacket** puede almacenar un paquete de información que se haya recibido. El constructor de **DatagramPacket** para este fin recibe dos argumentos: un arreglo byte en el cual se almacenarán los datos y la longitud del arreglo. El método *receive* de **DatagramSocket** sirve para esperar a que llegue un paquete al Servidor. El método *receive* hace un bloqueo hasta que llega el paquete y después lo almacena en su argumento **DatagramPacket** .

Este método lanza una excepción **IOException** si ocurre un error al recibir un paquete.

```
import javax.swing.JFrame;

public class PruebaServidor {
    public static void main(String args[]) {
        Servidor aplicacion = new Servidor(); // crea el servidor
        aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        aplicacion.esperarPaquetes(); // ejecuta la aplicación servidor
    } // fin de main
} // fin de la clase PruebaServidor
```

Cuando llega un paquete se hace una llamada al método **mostrarMensaje** para anexar el contenido del paquete al área de texto. El método *getAddress* de **DatagramPacket** devuelve un objeto **InetAddress** que contiene el nombre de host del equipo desde el que se envió el paquete. El método *getPort* devuelve un entero que especifica el número de puerto a través del que el equipo host enviará el paquete. El método *getLength* devuelve un entero que representa el número de bytes de datos que se

enviaron. El método *getData* devuelve un arreglo byte que contiene los datos.

Después de mostrar un paquete se hace una llamada al método **enviarPaqueteAlCliente** para crear un nuevo paquete y enviarlo al cliente. Al objeto **DatagramPacket** se le pasan cuatro argumentos a su constructor. El primer argumento especifica el arreglo byte a enviar. El segundo especifica el número de bytes a enviar. El tercer argumento especifica la dirección de Internet del equipo cliente a donde se va a enviar el paquete. El cuarto argumento especifica el puerto en el que el cliente espera recibir los paquetes. El método *send* de **DatagramSocket** lanza una excepción **IOException** si ocurre un error al enviar un paquete.

La clase Cliente

La clase Cliente funciona de manera similar a la clase Servidor, excepto que el Cliente envía paquetes sólo cuando el usuario escribe un mensaje en un campo de texto y oprime la tecla *ENTER*. Cuando esto ocurre, el programa llama al método *actionPerformed*, el cual convierte la cadena que introdujo el usuario en un arreglo byte. Se crea un objeto **DatagramPacket** y se inicializa con el arreglo byte, la longitud de la cadena introducida por el usuario, la dirección IP a la que se enviará el paquete (InetAddress.getLocalHost() en este ejemplo) y el número de puerto en el que el Servidor espera los paquetes (5000 en este ejemplo).

```
//Cliente que envía y recibe paquetes desde/hacia un servidor.
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
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 Cliente extends JFrame {
    private JTextField campoIntroducir; // para introducir mensajes
    private JTextArea areaPantalla; // para mostrar mensajes
    private DatagramSocket socket; // socket para conectarse al servidor

    // establece la GUI y el objeto DatagramSocket
    public Cliente() {
            super("Cliente");
            campoIntroducir = new JTextField("Escriba aqui el mensaje");
            campoIntroducir.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent evento) {
                            try // crea y envía un paquete
                            {
                                    // obtiene mensaje del campo de texto
                                    String mensaje = evento.getActionCommand();
                                    areaPantalla.append("\nEnviando paquete que contiene: 
" + mensaje + "\n");
                                    byte datos[] = mensaje.getBytes(); // convierte en 
bytes
                                    // crea objeto sendPacket
```

```
                                DatagramPacket paqueteEnviar = new
DatagramPacket(datos, datos.length, InetAddress.getLocalHost(),
                                            5000);
                                socket.send(paqueteEnviar); // envía el paquete
                                areaPantalla.append("Paquete enviado\n");

    areaPantalla.setCaretPosition(areaPantalla.getText().length());
                        } // fin de try
                        catch (IOException excepcionES) {
                                mostrarMensaje(excepcionES.toString() + "\n");
                                excepcionES.printStackTrace();
                        } // fin de catch
                } // fin de actionPerformed
        } // fin de la clase interna
        ); // fin de la llamada a addActionListener
        add(campoIntroducir, BorderLayout.NORTH);
        areaPantalla = new JTextArea();
        add(new JScrollPane(areaPantalla), BorderLayout.CENTER);
        setSize(400, 300); // establece el tamaño de la ventana
        setVisible(true); // muestra la ventana
        try // crea objeto DatagramSocket para enviar y recibir paquetes
        {
                socket = new DatagramSocket();
        } // fin de try
        catch (SocketException excepcionSocket) {
                excepcionSocket.printStackTrace();
                System.exit(1);
        } // fin de catch
    } // fin del constructor de Cliente

    // espera a que lleguen los paquetes
    public void esperarPaquetes() {
        while (true) {
                try // recibe paquete y muestra su contenido

                {
                        byte datos[] = new byte[100]; // establece el paquete
                        DatagramPacket paqueteRecibir = new DatagramPacket(datos,
datos.length);

                        socket.receive(paqueteRecibir); // espera el paquete

                        // muestra la informacion del paquete recibido
                        mostrarMensaje("\nPaquete recibido:" + "\nDe host: " +
paqueteRecibir.getAddress() + "\nPuerto host: "
                                    + paqueteRecibir.getPort() + "\nLongitud: " +
paqueteRecibir.getLength() + "\nContiene:\n\t"
                                    + new String(paqueteRecibir.getData(), 0,
paqueteRecibir.getLength()));

                } // fin de try
                catch (IOException excepcion) {
                        mostrarMensaje(excepcion.toString() + "\n");
                        excepcion.printStackTrace();
                } // fin de catch
        } // fin de while
    } // fin del método esperarPaquetes

    // manipula objeto areaPantalla en el subproceso despachador de eventos
    private void mostrarMensaje(final String mensajeAMostrar) {
        SwingUtilities.invokeLater(new Runnable() {
```

www.profmatiasgarcia.com.ar

```
                    public void run() // actualiza objeto areaPantalla
                    {
                          areaPantalla.append(mensajeAMostrar);
                    } // fin del método run
              } // fin de la clase interna
              ); // fin de la llamada a SwingUtilities.invokeLater
        } // fin del método mostrarMensaje
    }// fin de la clase Cliente
```

Observe que la llamada al constructor de **DatagramSocket** en esta aplicación no especifica ningún argumento. Este constructor sin argumentos permite a la computadora seleccionar el siguiente número de puerto disponible para el objeto **DatagramSocket**. El cliente no necesita un número de puerto específico, ya que el servidor recibe el número de puerto del cliente como parte de cada objeto **DatagramPacket** enviado por el cliente.

Por lo tanto, el servidor puede enviar paquetes de vuelta al mismo equipo y número de puerto desde el cual haya recibido un paquete de información.

El método **esperarPaquetes** de la clase Cliente utiliza un ciclo infinito para esperar a que lleguen los paquetes del servidor, se hace un bloqueo hasta que llegue un paquete. Esto no impide al usuario enviar un paquete, ya que los eventos de la GUI se manejan en el subproceso despachador de eventos. Sólo impide que el ciclo while continúe ejecutándose, hasta que llegue un paquete al Cliente. Cuando llega un paquete, se almacena este paquete en paqueteRecibir, y se hace una llamada al método **mostrarMensaje** para mostrar el contenido del paquete en el área de texto.

Principal

```
//Prueba la clase Cliente.
import javax.swing.JFrame;

public class PruebaCliente {
    public static void main(String args[]) {
          Cliente aplicacion = new Cliente(); // crea el cliente
          aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          aplicacion.esperarPaquetes(); // ejecuta la aplicación cliente
    } // fin de main
} // fin de la clase PruebaCliente
```

www.profmatiasgarcia.com.ar

BIBLIOGRAFÍA

- Deitel, Paul y Deitel, Harvey, “JAVA Cómo programar” 9na Ed. (Pearson 2012)
- Sanchez, Jorge, “JAVA 2” (2004)

LICENCIA

Este documento se encuentra bajo Licencia Creative Commons 2.5 Argentina (BY-NC-SA), por la cual se permite su exhibición, distribución, copia y posibilita hacer obras derivadas a partir de la misma, siempre y cuando se cite la autoría del **Prof. Matías E. García** y sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar

www.profmatiasgarcia.com.ar