

Universidad Nacional de Trujillo
Facultad de Ciencias Físicas y Matemáticas
Escuela de Informática



Sockets y su programación en Java

Amaro Calderón, Sarah Dámaris
Valverde Rebaza. Jorge Carlos

Trujillo – Perú
2007

Índice

	Pág.
1. Fundamentos.....	03
2. Definición.....	03
3. Modelos de Capas	04
4. Tipos de Sockets	04
4.1. Socket Stream.....	05
4.2. Socket Datagram.....	05
4.3. Socket Raw.....	05
4.1. Diferencias entre Socket Stream y Datagrama.....	05
5. Modelo de Sockets,.....	06
6. Funcionamiento Genérico de Sockets	06
7. Programación de Sockets en Java	09
7.1. Java Sockets	09
7.2. Modelo de Comunicaciones con Java	10
7.3. Apertura de Sockets	10
7.4. Creación de Streams	11
7.4.1. Streams de Entrada	11
7.4.2. Streams de Salida	12
7.5. Cierre de Sockets	13
8. Aplicación Cliente-Sevidor	14
9. Ejemplo de Aplicación: Transmisión de Archivos	15
10. Conclusiones	21
11. Referencias	21

Sockets

1. Fundamentos

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

Los sockets fueron desarrollados como un intento de generalizar el concepto de pipe (tubería unidireccional para la comunicación entre procesos en el entorno Unix) en 4.2BSD bajo contrato por DARPA (Defense Advanced Research Projects Agency). Sin embargo, fueron popularizados por *Berckley Software Distribution*, de la Universidad Norteamericana de Berkley.

Los sockets utilizan una serie de primitivas para establecer el punto de comunicación, para conectarse a una máquina remota en un determinado puerto que esté disponible, para escuchar en él, para leer o escribir y publicar información en él, y finalmente para desconectarse. Con todas las primitivas que ofrecen los sockets, se puede crear un sistema de diálogo muy completo.

2. Definición

Un socket es un punto final de un proceso de comunicación. Es una abstracción que permite manejar de una forma sencilla la comunicación entre procesos, aunque estos procesos se encuentren en sistemas distintos, sin necesidad de conocer el funcionamiento de los protocolos de comunicación subyacentes.

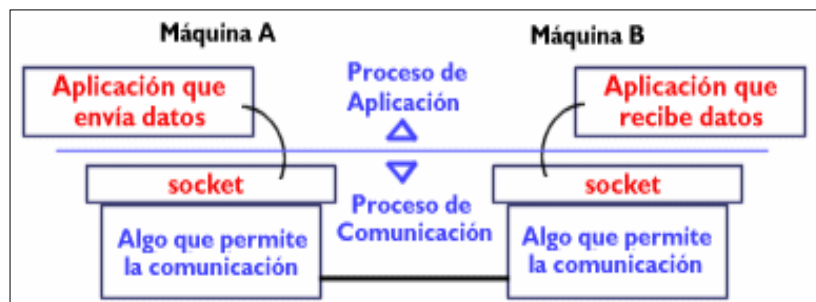


Figura 1. Abstracción del proceso de comunicación entre dos máquinas.

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

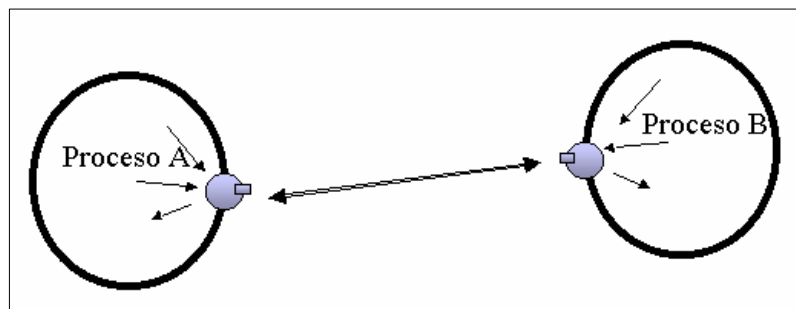


Figura 2. Comunicación entre dos procesos a través de sockets.

Los mecanismos de comunicación entre procesos pueden tener lugar dentro de la misma máquina o a través de una red. Generalmente son usados en forma de cliente-servidor, es decir, cuando un cliente y un servidor establecen una conexión, lo hacen a través de un socket.

3. Modelos de Capas

Antes de continuar, conviene tener en mente una referencia acerca del modelo de capas de redes para de esta manera tener un poco más clara el proceso de comunicación.

Los modelos de capas de dividen el proceso de comunicación en capas independientes. Cada capa proporciona servicios a la capa superior a través de una interfaz y a la vez recibe servicios de la capa inferior a través de la interfaz correspondiente. Este tipo de abstracción permite construir sistemas muy flexibles ya que se esconden los detalles de la implementación: la capa N sólo necesita saber que la capa N-1 le proporciona el servicio X, no necesita saber el mecanismo que utiliza dicha capa para lograr su objetivo.

En la figura 3 se observa que un socket sirve de interfaz entre la capa de Aplicación y la capa de Transporte en el modelo de capas usado en Internet.

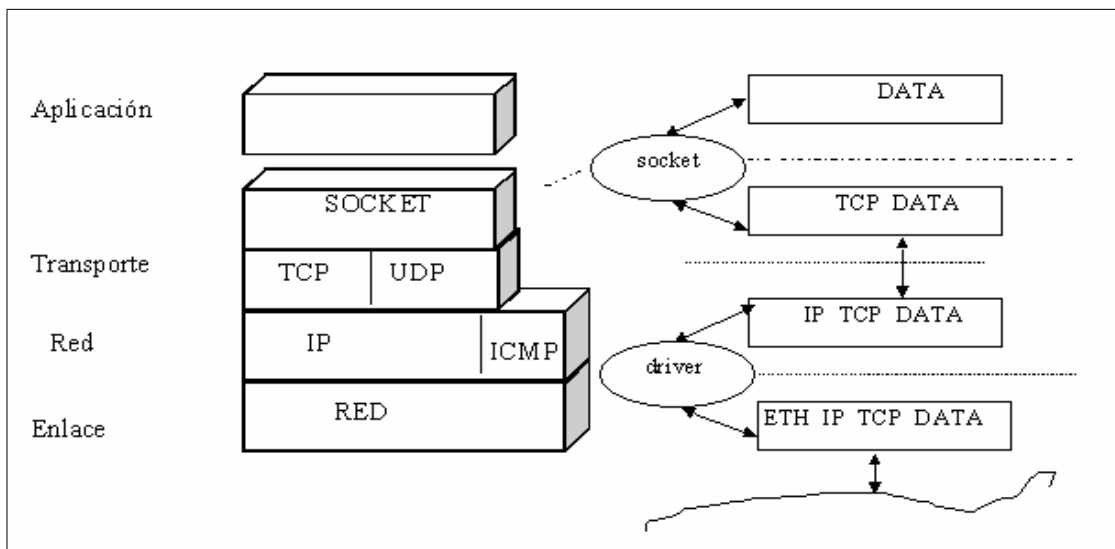


Figura 3. Socket como interfaz entre la capa de Aplicación y la de Transporte.

4. Tipos de Sockets

El tipo de sockets describe la forma en la que se transfiere información a través de ese socket. Existen muchos tipos de sockets, sin embargo, los más populares son:

- Stream (TCP)
- Datagram (UDP)
- Raw (acceso directo al protocolo: root)

4.1. Socket Stream

Son un servicio orientado a la conexión, donde los datos se transfieren sin encuadrarlos en registros o bloques, asegurándose de esta manera que los datos lleguen al destino en el orden de

transmisión. Si se rompe la conexión entre los procesos, éstos serán informados de tal suceso para que tomen las medidas oportunas, por eso se dice que están libres de errores.

El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP (Transmission Control Protocol), 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.

4.2. Socket Datagram

Son un servicio de transporte no orientado a la conexión. Son más eficientes que TCP, pero en su utilización 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.

Las comunicaciones a través de datagramas usan UDP (User Datagram Protocol), lo que significa que, 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, aunque tiene la ventaja de que se pueden indicar direcciones globales y el mismo mensaje llegará a un muchas máquinas a la vez.

4.3. Socket 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.

4.4. Diferencias entre Socket Stream y Datagrama

El problema aparece al momento de decidir por cual protocolo o tipo de socket usar. La decisión depende de la aplicación cliente/servidor que se esté desarrollando; aunque hay algunas diferencias entre los protocolos que sirven para ayudar en la decisión y utilizar un determinado tipo de socket.

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 los mensajes son más grandes que los TCP. Como el protocolo TCP está orientado a conexión, hay 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 es necesario emplear 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, es decir, no hay la seguridad de que el paquete llegue o no al destino. 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, la comunicación a través de sockets TCP es 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.

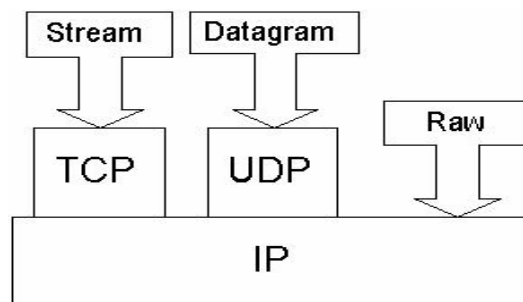


Figura 4. Tipos de socket para el protocolo de Internet

5. Modelo de Sockets

Como ya se menciona, los sockets extienden el concepto de descriptores de archivos para usarlos en conexiones remotas, en otras palabras, una conexión mediante sockets es idéntica a una comunicación mediante un pipe bidireccional, lo único que cambia es la forma de "abrir" el descriptor de archivo. Las funciones para manipular estos descriptores de archivo se presentan en la tabla 1.

6. Funcionamiento Genérico de Sockets

Como se ha mencionado, la comunicación con sockets hace uso de una serie de primitivas, entre las que destacan *socket* para establecer el punto de comunicación, *connect* para conectarse a una máquina remota en un determinado puerto que esté disponible, *bind* para escribir en él y publicar información, *read* para leer de él, *shutdown* o *close* para desconectarse, entre otras que se hacen mención en la tabla 1. Con todas estas primitivas se puede establecer un sistema de comunicaciones muy completo. En la figura 5 se muestra el funcionamiento de una conexión con sockets.

socket	crea un descriptor para usarlo en Tx sobre redes. Toma como parámetro la familia de protocolos, y el tipo de servicio (stream o datagram en en el caso de TCP/IP).
connect	establece una conexión activa, recibe como parámetro la dirección y puerto de destino.
write	generalmente copia los datos a un buffer y los envía a medida que puede. Si los buffers del S.O. están llenos, se bloquea.
read	lee de la conexión, se bloquea si no hay datos, o entrega <u>a lo más</u> length datos (length es un parámetro a la función). En UDP, si hay más datos que length en el datagrama, el resto se pierde, ya que no tiene sentido hacer otro read si no existe el concepto de conexión.
bind	especifica dirección (número IP + puerto local) al cual se asocia el socket.
listen	pone el socket en modo pasivo, y setea el número máximo de conexiones que se encolarán (cuando llegan conexiones simultáneas).
close	termina la conexión y libera el socket. Si es un socket compartido, ref_count - 1
shutdown	Termina la conexión TCP/IP en una o ambas direcciones.
getpeername	retorna dirección remota del socket.
getsockopt	ver opciones del socket.
setsockopt	cambiar opciones del socket.

Tabla 1. Funciones para manipular los descriptors de archivos para conexiones remotas.

Como se puede observar en la figura 5, un sistema de comunicación necesita de dos entidades bien diferenciadas: el Servidor y el Cliente.

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

En el lado del cliente: el cliente conoce el nombre de host de la máquina en la cual el servidor se encuentra ejecutando y el número de puerto en el cual el servidor está conectado. Para realizar una petición de conexión, el cliente intenta encontrar al servidor en la máquina servidora en el puerto especificado. Ver figura 6.

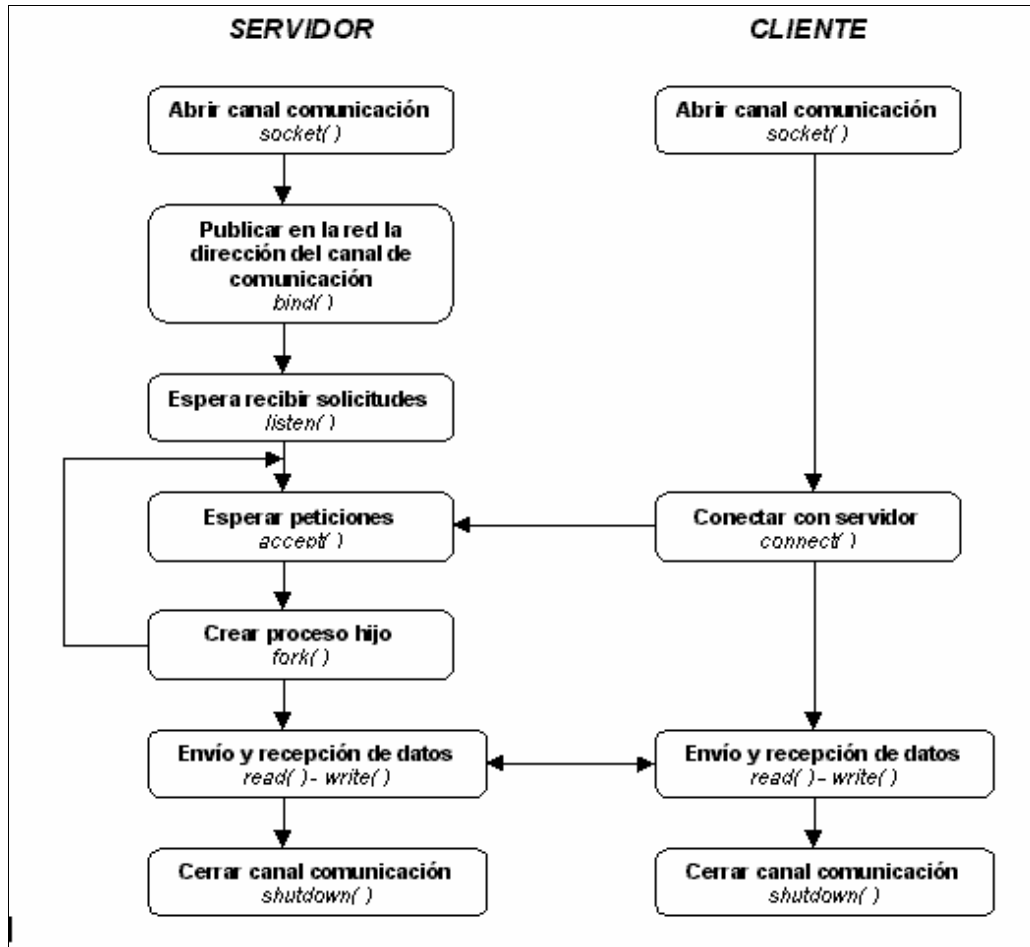


Figura 5. Funcionamiento de una conexión socket.

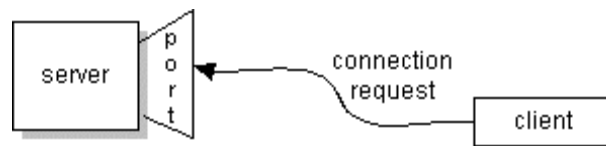


Figura 6. Cliente realiza petición de conexión al servidor.



Figura 7. Servidor acepta la solicitud y establece la conexión con el cliente.

Si todo va bien, el servidor acepta la conexión. Además de aceptar, el servidor obtiene un nuevo *socket* sobre un puerto diferente. Esto se debe a que necesita un nuevo *socket* (y, en consecuencia, un número de puerto diferente) para seguir atendiendo al *socket* original para peticiones de conexión mientras atiende las necesidades del cliente que se conectó. Ver figura 7.

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

7. Programación de Sockets en Java

La programación utilizando sockets involucra principalmente a dos clases: **Socket** y **DatagramSocket**, a la que se incorpora una tercera no tan empleada, **ServerSocket**, que solamente se utiliza para implementar *servidores*, mientras que las dos primeras se pueden usar para crear tanto *clientes* como *servidores*, representando comunicaciones *TCP* la primera y comunicaciones *UDP* la segunda.

La programación con sockets es una aproximación de bastante bajo nivel para la comunicación entre dos ordenadores que van a intercambiar datos. Uno de ellos será el *cliente* y el otro el *servidor*. Aunque la distinción entre cliente y servidor se va haciendo menos clara cada día, en Java hay una clara diferencia que es inherente al lenguaje. El cliente siempre inicia conexiones con servidores y los servidores siempre están esperando que un cliente quiera establecer una conexión. El hecho de que dos ordenadores puedan conectarse no significa que puedan comunicarse, es decir, que además de establecerse la conexión, las dos máquinas deben utilizar un protocolo entendible por ambas para poder entenderse.

La programación de sockets en el mundo Unix es muy antigua, Java simplemente encapsula mucha de la complejidad de su uso en clases, permitiendo un acercamiento a esa programación mucho más orientado a objetos de lo que podía hacerse antes. Básicamente, la programación de sockets hace posible que el flujo de datos se establezca en las dos direcciones entre cliente y servidor. El flujo de datos que se intercambian cliente y servidor se puede considerar de la misma forma que cuando se guardan y recuperan datos de un disco: como un conjunto de bytes a través de un canal. Y como en todo proceso en el que intervienen datos, el sistema es responsable de llevar esos datos desde su punto de origen al destinatario, y es responsabilidad del programador el asignar significado a esos datos.

Y esto de *asignar significado* tiene una especial relevancia en el caso de la utilización de sockets. En particular, como se ha dicho, entra entre las responsabilidades del programador la implementación de un protocolo de comunicaciones que sea mutuamente aceptable entre las dos máquinas a nivel de aplicación, para hacer que los datos fluyan de forma ordenada. Un *protocolo a nivel de aplicación* es un conjunto de reglas a través de las cuales los programas que se ejecutan en los dos ordenadores pueden establecer una conexión e intercambiarse datos.

7.1. Java Sockets

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

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

De forma adicional, java.net incluye la clase **ServerSocket**, la cual implementa un *socket* el cual los servidores pueden utilizar para escuchar y aceptar peticiones de conexión de clientes. Por otra parte, si intentamos conectar a través de la Web, la clase **URL** y clases relacionadas (**URLConnection**, **URLEncoder**) son probablemente más apropiadas que las clases de *sockets*. Pero de hecho, las clases **URL** no son más que una conexión a un nivel más alto a la web y utilizan como parte de su implementación interna a los sockets.

7.2. Modelo de comunicaciones con Java

El modelo de comunicaciones más simple es el siguiente:

- El servidor establece un puerto y espera durante un cierto tiempo (timeout) a que el cliente establezca la conexión. Cundo 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 el parámetro respectivo.
- El cliente y el servidor se comunican con manejadores *InputStream* y *OutputStream*.

En la figura 8 se puede observar el modelo de comunicaciones descrito.

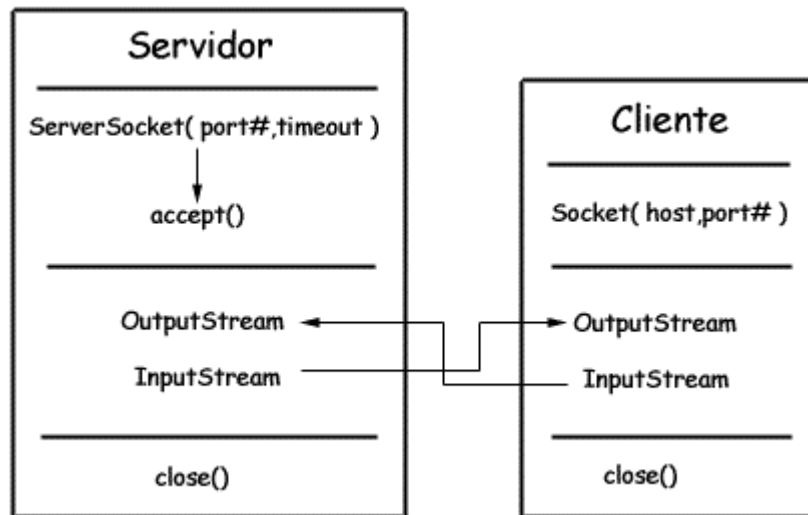


Figura 8. Modelo de comunicación Cliente-Servidor con sockets en Java

7.3. Apertura de Sockets

Si estamos implementando 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 );
}
```

7.4. Creación de Streams

7.4.1. 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()`. Debemos utilizar la función que creamos necesaria dependiendo del tipo de dato que esperamos 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 );
}
```

7.4.2. Streams de Salida

En la parte 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.

7.5. Cierre de Sockets

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

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

Y en la parte del servidor:

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

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

8. Aplicación Cliente-Servidor

Para entender de manera práctica los métodos descritos líneas arriba mostraremos una pequeña aplicación cliente-servidor. Primero crearemos un servidor *Server.java* que atenderá a un cliente. Para hacerlo simple, el servidor sólo le enviará un mensaje al cliente y éste terminará la conexión. El servidor quedará disponible para atender a otro cliente. Es importante saber que, para que el socket funcione, los servicios TCP/IP deben de estar activos, aunque los programas cliente y servidor corran en la misma máquina.

A continuación escribiremos el código del servidor que “correrá para siempre”, así que para detenerlo deberá de cortar manualmente la aplicación.

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

public class Server
{
    public static void main(String argv[])
    {
        ServerSocket servidor;
        Socket cliente;
        int numCliente = 0;
        int PUERTO 5000;
        try {
            servidor = new ServerSocket(PUERTO);
            do {
                numCliente++;
                cliente = servidor.accept();
                System.out.println("Llega el cliente "+numCliente);
                PrintStream ps = new PrintStream(cliente.getOutputStream());
                ps.println("Usted es mi cliente "+numCliente);
                cliente.close();
            } while (true);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Ahora vamos a crear la clase *Cliente.java* que representará al cliente que tan solo establece la conexión, lee a través de un *DataInputStream* mediante el método *readLine()* lo que el servidor le manda, lo muestra y lo corta.

```
// cliente:
import java.io.*;
import java.net.*;
```

```

public class Cliente
{
    public static void main(String argv[])
    {
        InetAddress direccion;
        Socket servidor;
        int numCliente = 0;
        int PUERTO 5000;
        try {
            direccion = InetAddress.getLocalHost(); // dirección local
            servidor = new Socket(direccion, PUERTO);
            DataInputStream datos = new DataInputStream(servidor.getInputStream());
            System.out.println(datos.readLine());
            servidor.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

Para probar esto, asegúrense que los servicios TCP/IP estén activos, ejecute *Servidor.java* en una ventana y ejecute varias veces *Cliente.java* en otras ventanas. Las salidas serán mas o menos así:

Ventana servidor:

```

C:\java\curso>java Servidor
Llega el cliente 1
Llega el cliente 2
Llega el cliente 3
(----- cortar con control-C -----)

```

Ventana cliente:

```

C:\java\curso>java Cliente
Usted es mi cliente 1
C:\java\curso>java Cliente
Usted es mi cliente 2
C:\java\curso>java Cliente
Usted es mi cliente 3
(----- aquí cerramos el servidor -----)

```

9. Ejemplo de aplicación: Transmisión de Archivos

Para entender con mayor claridad la aplicación de los sockets, a continuación se presenta el código de un sistema cliente-servidor para la transmisión de archivos en cualquier formato.

El Servidor tiene la capacidad de transmitir a los clientes que lo deseen un determinado archivo, por ejemplo *trabajo.ppt*, el cual se encuentra localizado en un path específico que el Servidor conoce. Entonces, uno o más clientes acceden a dicho servidor por medio de su dirección ip y colocan el path local en el que desean almacenar el archivo que están solicitando al servidor. El código de la aplicación descrita es el siguiente:

```

//Servidor.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

```

```
import java.net.*;

public class Servidor implements ActionListener
{
    JFrame frameServidor;
    JPanel panelServidor;
    JLabel lblNombreArchivo;
    JTextField txtNombreArchivo;
    JButton btnEscuchar;
    ServerSocket srvSocket;

    static final int PUERTO = 5000;
    boolean escuchando = false;

    public Servidor()
    {
        frameServidor = new JFrame("Servidor de Archivos");
        panelServidor = new JPanel();
        lblNombreArchivo = new JLabel("Nombre del Archivo:");
        txtNombreArchivo = new JTextField(40);
        btnEscuchar = new JButton("Escuchar");
        btnEscuchar.addActionListener(this);
        JPanel panelNorte = new JPanel();
        JPanel panelSur = new JPanel();
        panelNorte.setLayout(new FlowLayout());
        panelNorte.add(lblNombreArchivo);
        panelNorte.add(txtNombreArchivo);
        panelSur.setLayout(new FlowLayout());
        panelSur.add(btnEscuchar);
        panelServidor.setLayout(new BorderLayout());
        panelServidor.add(panelNorte, BorderLayout.NORTH);
        panelServidor.add(panelSur, BorderLayout.SOUTH);
        frameServidor.getContentPane().add(panelServidor);
        frameServidor.pack();
        frameServidor.setVisible(true);
        frameServidor.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void actionPerformed(ActionEvent e)
    {
        //Aqui llamaremos al hilo de aceptaciones
        try{
            if(!escuchando)
            {
                srvSocket = new ServerSocket(PUERTO);
                System.out.println("Servidor activo en el puerto "+ PUERTO);
                new HiloSolicitud(srvSocket, txtNombreArchivo.getText());
                btnEscuchar.setText("Terminar");
                escuchando = true;
            }
            else{
                srvSocket.close();
                btnEscuchar.setText("Escuchar");
                escuchando = false;
            }
        }
    }
}
```

```
        catch(Exception ex)
        {
            System.out.println(ex.getMessage());
        }
    }

    public static void main(String[] args)
    {
        new Servidor();
    }
}
```

```
//HiloSolicitud.java
```

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

```
public class HiloSolicitud extends Thread
{
    ServerSocket srvServidor;
    String nombreArchivo;

    public HiloSolicitud(ServerSocket servidor, String nomArch)
    {
        srvServidor = servidor;
        nombreArchivo = nomArch;
        start();
    }

    public void run()
    {
        try{
            while(true)
            {
                Socket cliente = srvServidor.accept();
                DataOutputStream flujo=new DataOutputStream(cliente.getOutputStream());
                new HiloTransferencia(flujo, nombreArchivo);
            }
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

```
//HiloTransferencia.java
```

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

```
public class HiloTransferencia extends Thread
{
    DataOutputStream cliente;
```

```
String nombreArchivo;

public HiloTransferencia(DataOutputStream flujo, String nomArch)
{
    cliente = flujo;
    nombreArchivo = nomArch;
    start();
}

public void run()
{
    //Abriremos el archivo e iniciaremos la transferencia
    try{
        RandomAccessFile archivo = new RandomAccessFile(nombreArchivo,"rw");
        //Lo primero que le manda el servidor al cliente es el tamaño del archivo
        cliente.writeLong(archivo.length());
        System.out.println("Se envió el tamaño satisfactoriamente...");
        System.out.println("Inicia la transferencia del archivo...");
        while(archivo.getFilePointer()<archivo.length())
        {
            byte dato = archivo.readByte();
            cliente.writeByte(dato);
        }
        System.out.println("Termino la Transferencia...");
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
}
```

```
//Cliente.java
```

```
import javax.swing.*;
import java.awt.*;
import java.net.*;
import java.awt.event.*;

public class Cliente implements ActionListener
{
    JFrame frameCliente;
    JPanel panelCliente;
    JLabel lblArchivo;
    JLabel lblDireccion;
    JTextField txtArchivo;
    JTextField txtDireccion;
    JButton btnRecibir;
    Socket cliSocket;
    String nomArch;
    static final int PUERTO = 5000;

    public Cliente()
    {
        JPanel panelNorte = new JPanel();
        JPanel panelCentro = new JPanel();
```

```

JPanel panelSur = new JPanel();
frameCliente = new JFrame("Cliente de Compartición de Archivos");
panelCliente = new JPanel();

lblArchivo = new JLabel("Archivo:");
txtArchivo = new JTextField(30);
lblDireccion = new JLabel("Direccion:");
txtDireccion = new JTextField(20);
btnRecibir = new JButton("Iniciar");
btnRecibir.addActionListener(this);
frameCliente.getContentPane().add(panelCliente);
panelNorte.setLayout(new FlowLayout());
panelCentro.setLayout(new FlowLayout());
panelSur.setLayout(new FlowLayout());
panelNorte.add(lblArchivo);
panelNorte.add(txtArchivo);
panelCentro.add(lblDireccion);
panelCentro.add(txtDireccion);
panelSur.add(btnRecibir);
panelCliente.setLayout(new BorderLayout());
panelCliente.add(panelNorte, BorderLayout.NORTH);
panelCliente.add(panelCentro, BorderLayout.CENTER);
panelCliente.add(panelSur, BorderLayout.SOUTH);
frameCliente.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frameCliente.pack();
frameCliente.setVisible(true);
}

public void actionPerformed(ActionEvent e)
{
    try{
        // creamos el socket cliente
        cliSocket = new Socket(txtDireccion.getText(), PUERTO);
        // enviamos el socket y la dirección donde se guardará el archivo
        new HiloTransferenciaCliente(cliSocket, txtArchivo.getText()
    }
    catch(Exception ex)
    {
        System.out.println(ex.getMessage());
    }
}

public static void main(String[] args)
{
    new Cliente();
}
}

```

```
//HiloTransferenciaCliente.java
```

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

```
public class HiloTransferenciaCliente extends Thread
{
    Socket cliente;
```

```

DataInputStream entrada;
String nomArchivo;
long tamArchivo;

//recibe el socket(ip y puerto) y el directorio en el que se guardará lo que se reciba
public HiloTransferenciaCliente(Socket cli, String archivo)
{
    cliente = cli;
    nomArchivo = archivo;
    try{
        entrada = new DataInputStream(cliente.getInputStream());
        tamArchivo = entrada.readLong();
        System.out.println("Se leyó correctamente el tamaño del archivo..." + tamArchivo);
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
    start();//inicio el thread
}

public void run()
{
    int numBytes = 0;
    System.out.println("Inicia la transferencia del archivo...");
    try{
        RandomAccessFile archivo = new RandomAccessFile(nomArchivo, "rw");
        while(numBytes < tamArchivo)
        {
            byte dato = entrada.readByte();
            archivo.writeByte(dato);
            numBytes++;
        }
        System.out.println("Termino la transferencia del archivo...");
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
}
}

```

Como se puede notar, tanto el cliente como el servidor hacen uso de Threads para un mejor manejo de la transmisión del archivo, ya que el servidor envía el archivo por paquetes de un determinado tamaño y el cliente se encarga de recibir cada uno de estos paquetes y formar nuevamente el archivo para almacenarlo finalmente en el path destino.

En las figuras 9 y 10 se observan las interfaces al ejecutar *Servidor.java* y *Cliente.java* en una misma maquina. El servidor almacena *trabajo.ppt* en el path *D:\ \ Documentos \ \ trabajo.ppt*. El cliente que es ejecutado en la misma maquina desea que el servidor le envíe *trabajo.ppt* y lo almacene en *F:\ \ CopiaTrabajo.ppt*. Al hacer click en el botón Iniciar de la interfaz del cliente, la solicitud llega al servidor quien inmediatamente realiza la transmisión al path especificado por el cliente. Al terminar la transmisión desde el servidor al cliente, podemos verificar que *trabajo.ppt* que se encontraba en *D:\ \ Documentos \ \ trabajo.ppt* ahora se encuentra en *F:\ \ CopiaTrabajo.ppt*, por lo que la transmisión del archivo ha sido realizada satisfactoriamente.

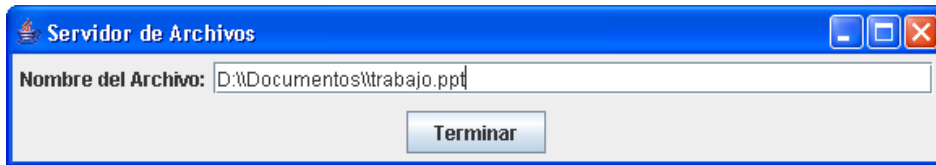


Figura 9. Interfaz del Servidor de Archivos.

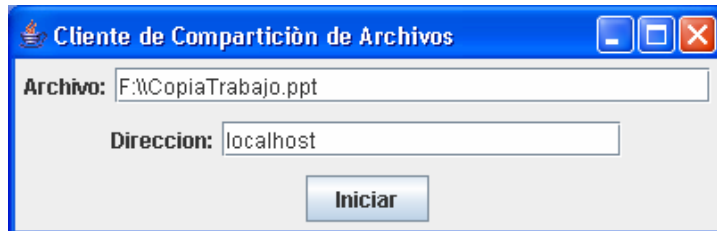


Figura 10. Interfaz del Cliente de Compartición de Archivos.

10. Conclusiones

- Los sockets son mecanismos que permiten una comunicación sencilla entre procesos remotos, otorgando distintos niveles de fiabilidad de acuerdo al tipo de socket que se use, es decir, que dependiendo de la aplicación y del tipo de socket que se use en ella, la comunicación siempre se realizará dentro de los parámetros predefinidos.
- El uso de sockets se ha extendido debido a que han sido diseñadas para servir en la comunicación en el protocolo IP; resultando eficientes al momento de su aplicación.
- El uso de sockets en Java abstrae un conjunto de operaciones de bajo nivel que bajo nivel, lo que resulta beneficioso para el programador que tan sólo se enfoca en la aplicación y la manera en la que enviará y recibirá los distintos mensajes durante la comunicación.

11. Referencias

- [1] Guia Beej de Programación en Redes
<http://www.arrakis.es/~dmrq/beej/theory.html>
- [2] Los Sockets
<http://www.angelfire.com/trek/storwald/Socketts.pdf>
- [3] Tutorial de Java: Los Sockets
<http://www.itapizaco.edu.mx/paginas/JavaTut/froufe/parte20/cap20-3.html>
- [4] Joaquín Salvachúa: Comunicación mediante Sockets
<http://www.lab.dit.upm.es/~cdatlab/cursos/cdatlab/sockets/sld001.htm>
- [5] Tutorial de Java: Dominios de Comunicaciones
<http://www.itapizaco.edu.mx/paginas/JavaTut/froufe/parte20/cap20-4.html>
- [6] Sockets in TCP/IP Networking
<http://www.networkdictionary.com/networking/sockets.php>
- [7] Los Sockets
<http://pisuerga.inf.ubu.es/lsi/Docencia/TFC/ITIG/icruzadn/Memoria/27.htm>
- [8] Comunicación de Datos
<http://www.dcc.uchile.cl/~jpiquer/Docencia/cc51c/apuntes/apuntes.html>