1.- FUNDAMENTOS 2

2.- FUNCIONAMIENTO GENÉRICO 3

3.- JAVA SOCKETS 4

3.1.- INTRODUCCION 4

3.2.- MODELO DE COMUNICACIONES CONJAVA 5

3.3.- APERTURA DE SOCKETS 6

3.4.- CREACIÓN DE STREAMS 7

3.4.1.- Creación de Streams de Entrada 7

3.4.2.- Creación de Streams de Salida 8

3.5.- CIERRE DE SOCKETS 9

3.6.- CLASES ÚTILES EN COMUNICACIONES 10

3.7.- EJEMPLO DE USO 11

3.7.1.- Programa Cliente 11

3.7.2.- Programa Servidor 12

3.7.3.- Ejecución 14

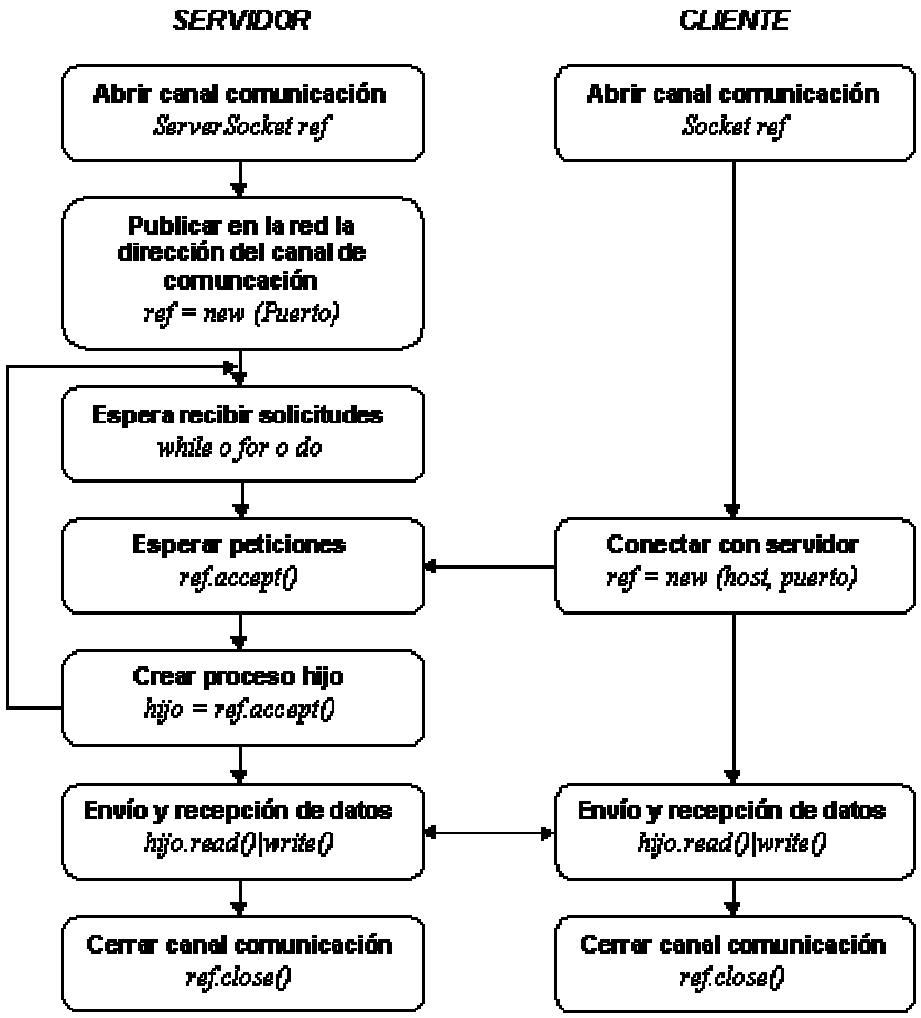
1.- Fundamentos

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

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

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

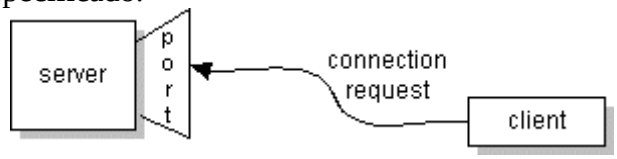
Con todas las primitivas se puede crear un sistema de diálogo muy completo.



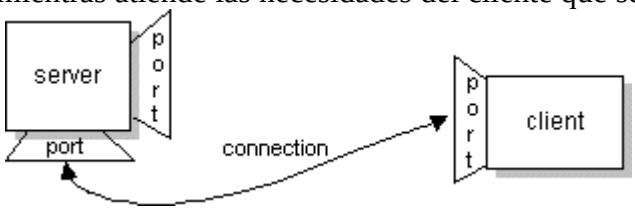
2.- Funcionamiento genérico

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

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



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



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*.

3.- JAVA Sockets

3.1.- Introduccion

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

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

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

Nuestro objetivo será conocer cómo utilizar las clases **Socket** y **ServerSocket**.

Por otra parte, si intentamos conectar a través de la Web, la clase **URL** y clases relacionadas (**URLConnection**, **URLEncoder**) son probablemente más apropiadas que las clases de *sockets*. Pero de hecho , las clases **URL** no son más que una conexión a un nivel más alto a la Web y utlilizan como parte de su implementación interna los *sockets*.

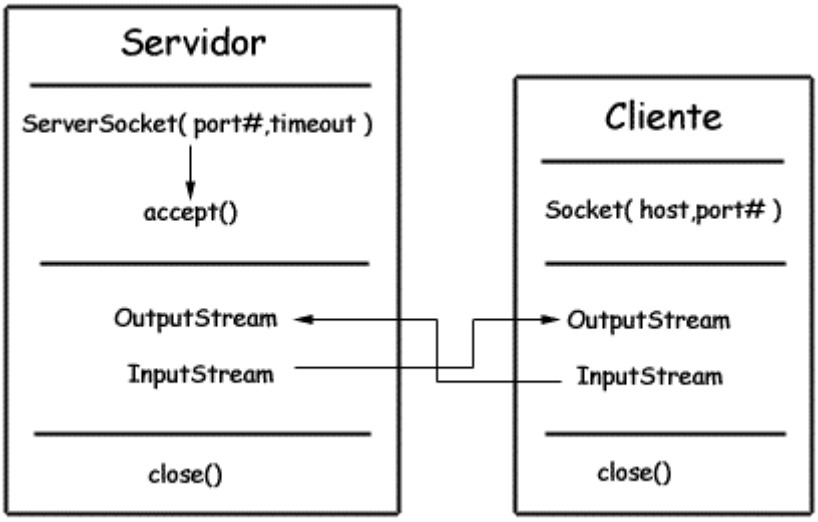
3.2.- Modelo de comunicaciones con Java

El modelo de *sockets* más simple es:

· El servidor establece un puerto y espera durante un cierto tiempo (*timeout* segundos), a que el cliente establezca la conexión. Cuando el cliente solicite una conexión, el servidor abrirá la conexión *socket* con el método *accept()*.

· El cliente establece una conexión con la máquina host a través del puerto que se designe en *puerto#*

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



3.3.- Apertura de Sockets

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

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

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

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

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

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

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

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

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

3.4.- Creación de Streams

3.4.1.- Creación de Streams de Entrada

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

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

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

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

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

3.4.2.- Creación de Streams de Salida

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

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

La clase **PrintStream** tiene métodos para la representación textual de todos los datos primitivos de Java. Sus métodos *write* y *println()* tienen una especial importancia en este aspecto. No obstante, para el envío de información al servidor también podemos utilizar **DataOutputStream**:

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

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

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

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

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

3.5.- 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.

3.6.- Clases útiles en comunicaciones

Socket

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

ServerSocket

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

DatagramSocket

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

DatagramPacket

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

MulticastSocket

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

NetworkServer

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

NetworkClient

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

SocketImpl

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

3.7.- Ejemplo de uso

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

3.7.1.- Programa Cliente

El programa cliente se conecta a un servidor indicando el nombre de la máquina y el número puerto (tipo de servicio que solicita) en el que el servidor está instalado.

Una vez conectado, lee una cadena del servidor y la escribe en la pantalla:

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

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

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

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

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

3.7.2.- Programa Servidor

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

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

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

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

```
import java.io.* ;
import java.net.* ;
class Servidor {
  static final int PUERTO=5000;
  public Servidor( ) {
    try {
          ServerSocket skServidor = new ServerSocket(PUERTO);
          System.out.println("Escucho el puerto " + PUERTO );
          for ( int numCli = 0; numCli < 3; numCli++; ) {
            Socket skCliente = skServidor.accept(); // Crea objeto
            System.out.println("Sirvo al cliente " + numCli);
            OutputStream aux = skCliente.getOutputStream();
            DataOutputStream flujo= new DataOutputStream( aux );
            flujo.writeUTF( "Hola cliente " + numCli );
            skCliente.close();
          }
    System.out.println("Demasiados clientes por hoy");
    } catch( Exception e ) {
          System.out.println( e.getMessage() );
    }
  }
  public static void main( String[] arg ) {
    new Servidor();
  }
}
```

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

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

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

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

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

3.7.3.- Ejecución

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

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

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

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

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

Mientras tanto en la ventana del servidor se ha impreso:

```
Sirvo al cliente 1
Sirvo al cliente 2
Sirvo al cliente 3
Demasiados clientes por hoy
```

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

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