Unidad Didáctica 3:

PROGRAMACIÓN DE COMUNICACIONES EN RED

3.1. El Modelo Cliente-Servidor

Desde el punto de vista funcional, se puede definir la computación Cliente/Servidor como una arquitectura distribuida que permite a los usuarios finales obtener acceso a la información en forma transparente aún en entornos multiplataforma.

En el modelo cliente servidor, el cliente envía un mensaje solicitando un determinado servicio a un servidor (hace una petición), y este envía uno o varios mensajes con la respuesta (provee el servicio). En un sistema distribuido cada máquina puede cumplir el rol de servidor para algunas tareas y el rol de cliente para otras.
La idea es tratar a una computadora como un instrumento, que por sí sola pueda realizar muchas tareas, pero con la consideración de que realice aquellas que son más adecuadas a sus características. Si esto se aplica tanto a clientes como servidores se entiende que la forma más estándar de aplicación y uso de sistemas Cliente/Servidor es mediante la explotación de los PC's a través de interfaces gráficas de usuario; mientras que la administración de datos y su seguridad e integridad se deja a cargo de computadoras centrales tipo mainframe. Usualmente la mayoría del trabajo pesado se hace en el proceso llamado servidor y el o los procesos cliente sólo se ocupan de la interacción con el usuario (aunque esto puede variar). En otras palabras la arquitectura Cliente/Servidor es una extensión de programación modular en la que la base fundamental es separar una gran pieza de software en módulos con el fin de hacer más fácil el desarrollo y mejorar su mantenimiento.

Esta arquitectura permite distribuir físicamente los procesos y los datos en forma más eficiente. Lo que en computación distribuida afecta directamente el tráfico de la red, reduciéndolo ampliamente.

Cliente
El cliente es el proceso que permite al usuario formular los requerimientos y pasarlos al servidor, se le conoce con el término front-end.

Servidor
Es el proceso encargado de atender a múltiples clientes que hacen peticiones de algún recurso administrado por él. Al proceso servidor se le conoce con el término back-end.
El servidor normalmente maneja todas las funciones relacionadas con la mayoría de las reglas del negocio y los recursos de datos.

3.2. Sockets

Un socket es un proceso o hilo existente en la máquina cliente y en la máquina servidor, que sirve en última instancia para que el programa servidor y el cliente lean y escriban la información. Esta información será la transmitida por las diferentes capas de red.
Los procesos envían/reciben mensajes a/desde sus sockets.

Para que dos programas puedan comunicarse entre sí es necesario que se cumplan ciertos requisitos:

- Que un programa sea capaz de localizar al otro.
- Que ambos programas sean capaces de intercambiarse cualquier secuencia de octetos, es decir, datos relevantes a su finalidad.

Para ello son necesarios los dos recursos que originan el concepto de socket:

- Direcciones del protocolo de red (dirección IP, si se utiliza el protocolo TCP/IP), que identifican la maquina de origen y la remota.
- Números de puerto, que identifican a un programa dentro de cada maquina.

Los sockets permiten implementar una arquitectura cliente-servidor. La comunicación debe ser iniciada por uno de los programas que se denomina programa "cliente". El segundo programa espera a que otro inicie la comunicación, por este motivo se denomina programa "servidor".
Las propiedades de un socket dependen de las características del protocolo en el que se implementan. El protocolo más utilizado es Transmission Control Protocol; una alternativa común a éste es User Datagram Protocol.
Cuando se implementan con el protocolo TCP, los sockets tienen las siguientes propiedades:

- Son orientados a la conexión
- Se garantiza la transmisión de todos los octetos sin errores ni omisiones.
- Se garantiza que todo octeto llegará a su destino en el mismo orden en que se ha transmitido.

Estas propiedades son muy importantes para garantizar la corrección de los programas que tratan la información.
El protocolo UDP es un protocolo no orientado a la conexión. Sólo se garantiza que si un mensaje llega, llegue bien. En ningún caso se garantiza que llegue o que lleguen todos los mensajes en el mismo orden que se mandaron. Esto lo hace adecuado para el envío de mensajes frecuentes pero no demasiado importantes, como por ejemplo, un streaming de audio.

Tipos de Sockets

Existen básicamente dos tipos:

- Los no orientados a conexión
  - El programa de aplicación da la fiabilidad.
- Los orientados a conexión.
  - Comunicaciones fiables
  - Circuito Virtual

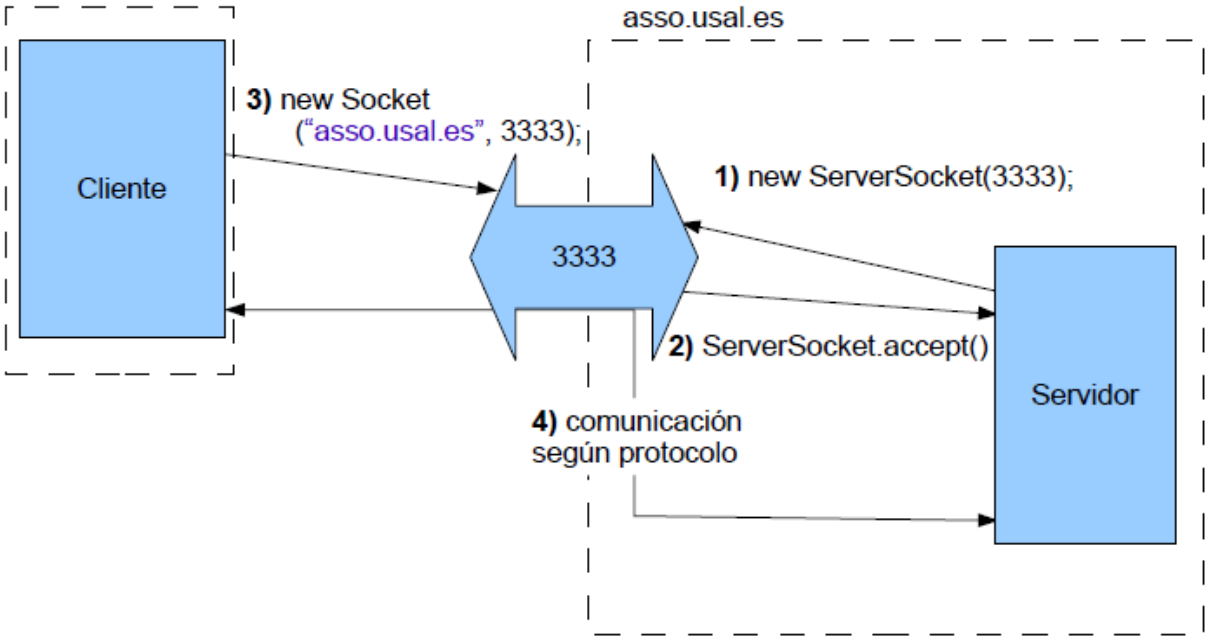
Un socket queda definido por una dirección IP, un protocolo y un número de puerto. En el caso concreto de TCP-IP, un socket se define por una dupla Origen - Destino. Tanto el origen como el destino vienen indicados por un par (ip, puerto).

Los Sockets no orientados a conexión, son los llamados protocolos UDP.

- No es necesario que los programas se conecten.
- Cualquiera de ellos puede transmitir datos en cualquier momento, independientemente de que el otro programa esté "escuchando" o no.
- Garantiza que los datos que lleguen son correctos, pero no garantiza que lleguen todos.
- Se utiliza cuando es muy importante que el programa no se quede bloqueado.
- No importa que se pierdan datos.

Los sockets orientados a conexión.

- Primero hay que establecer correctamente la conexión.
- Se usa el protocolo TCP del protocolo TCP/IP, para gestionar la conexión.
- Se garantiza que todos los datos van a llegar de un programa al otro correctamente.
- Se utiliza cuando la información a transmitir es importante, no se puede perder ningún dato.



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*.

3.2.1. Sockets en java

La programación en red siempre ha sido dificultosa, el programador debía de conocer la mayoría de los detalles de la red, incluyendo el hardware utilizado, los distintos niveles en que se divide la capa de red, las librerías necesarias para programar en cada capa, etc.
Pero, la idea simplemente consiste en obtener información desde otra máquina, aportada por otra aplicación software. Por lo tanto, de cierto modo se puede reducir al mero hecho de leer y escribir archivos, con ciertas salvedades.
El sistema de Entrada/Salida sigue el paradigma que normalmente se designa cómo Abrir-Leer-Escribir-Cerrar. Antes de que un proceso de usuario pueda realizar operaciones de entrada/salida, debe hacer una llamada a Abrir (open) para indicar, y obtener los permisos del fichero o dispositivo que se desea utilizar.
Una vez que el fichero o dispositivo se encuentra abierto, el proceso de usuario realiza una o varias llamadas a Leer (read) y Escribir (write), para la lectura y escritura de los datos.
El proceso de lectura toma los datos desde el objeto y los transfiere al proceso de usuario, mientras que el de escritura los transfiere desde el proceso de usuario al objeto. Una vez concluido el intercambio de información, el proceso de usuario llamará a Cerrar (close) para informar al sistema operativo que ha finalizado la utilización del fichero o dispositivo.
El ciclo de vida de un socket está determinado por tres fases:
- Creación, apertura del socket.
- Lectura y Escritura, recepción y envío de datos por el socket.
- Destrucción, cierre del socket.

3.2.2. Clases para las comunicaciones de red en java: java.net

En las aplicaciones en red es muy común el paradigma cliente-servidor. El servidor es el que espera las conexiones del cliente (en un lugar claramente definido) y el cliente es el que lanza las peticiones a la maquina donde se está ejecutando el servidor, y al lugar donde está esperando el servidor (el puerto(s) específico que atiende). Una vez establecida la conexión, ésta es tratada como un stream (flujo) típico de entrada/salida.
Cuando se escriben programas Java que se comunican a través de la red, se está programando en la capa de aplicación. Típicamente, no se necesita trabajar con las capas TCP y UDP, en su lugar se puede utilizar las clases del

paquete java.net. Estas clases proporcionan comunicación de red independiente del sistema.
A través de las clases del paquete java.net, los programas Java pueden utilizar TCP o UDP para comunicarse a través de Internet. Las clases URL, URLConnection, Socket, y SocketServer utilizan TCP para comunicarse a través de la red. Las clases DatagramPacket y DatagramServer utilizan UDP.
TCP proporciona un canal de comunicación fiable punto a punto, lo que utilizan para comunicarse las aplicaciones cliente-servidor en Internet. Las clases Socket y ServerSocket del paquete java.net proporcionan un canal de comunicación independiente del sistema utilizando TCP, cada una de las cuales implementa el lado del cliente y el servidor respectivamente.
Así el paquete java.net proporciona, entre otras, las siguientes clases, que son las que se verán con detalle:

- Socket: Implementa un extremo de la conexión TCP,
- ServerSocket: Se encarga de implementar el extremo Servidor de la conexión en la que se esperarán las conexiones de los clientes.
- DatagramSocket: Implementa tanto el servidor como el cliente cuando se utiliza UDP.
- DatagramPacket: Implementa un datagram packet, que se utiliza para la creación de servicios de reparto de paquetes sin conexión.
- lnetAddress: Se encarga de implementar la dirección IP.

La clase Socket del paquete java.net es una implementación independiente de la plataforma de un cliente para un enlace de comunicación de dos vías entre un cliente y un servidor. Utilizando la clase java.net.Socket en lugar de tratar con código nativo, los programas Java pueden comunicarse a través de la red de una forma independiente de la plataforma.
El entorno de desarrollo de Java incluye un paquete, java.io, que contiene un juego de canales de entrada y salida que los programas pueden utilizar para leer y escribir datos. Las clases InputStream y OutputStream del paquete java.io son superclases abstractas que definen el comportamiento de los canales de I/O de tipo stream de Java. java.io también incluye muchas subclases de InputStream y OutputStream que implementan tipos específicos de canales de I/O.

3.2.3 Datagram socket(Servicio sin Conexión)

Es el más simple, lo único que se hace es enviar los datos, mediante la creación de un socket y utilizando los métodos de envío y recepción apropiados.
Se trata de un servicio de transporte sin conexión. No está garantizada la fiabilidad: los datos se envían y reciben en paquetes, cuya entrega no está garantizada; los

paquetes pueden ser duplicados, perdidos o llegar en un orden diferente al que se envió.
El protocolo de comunicaciones con datagramas UDP, es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del socket local y la dirección del socket que debe recibir el datagrama. Como se puede ver, hay que enviar datos adicionales cada vez que se realice una comunicación.

public class java.net.DatagramSocket extends java.lang.Object

A) CONSTRUCTORES :
public DatagramSocket () throws SocketException
Se encarga de construir un socket para datagramas y de conectarlo al primer puerto disponible.
public DatagramSocket (int port) throws SocketException
Ídem, pero con la salvedad de que permite especificar el número de puerto asociado.
public DatagramSocket (int port, InetAddress ip) throws SocketException
Permite especificar, además del puerto, la dirección local a la que se va a asociar el socket.

B) MÉTODOS:
public void close()
Cierra el socket.
protected void finalize()
Asegura el cierre del socket si no existen más referencias al mismo.
public int getLocalPort()
Retorna el número de puerto en el host local al que está conectado el socket.
public void receive (DatagramPacket p) throws lOException
Recibe un DatagramPacket del socket, y llena el búfer con los datos que recibe.
public void send (DatagramPacket p) throws lOException
Envía un DatagramPacket a través del socket.

3.2.4 DatagramPacket

Un DatagramSocket envía y recibe los paquetes y un DatagramPacket contiene la información relevante. Cuando se desea recibir un datagrama, éste deberá almacenarse bien en un búfer o un array de bytes. Y cuando preparamos un datagrama para ser enviado, el DatagramPacket no sólo debe tener la información, sino que además debe tener la dirección IP y el puerto de destino, que puede coincidir con un puerto TCP.

public final class java.net. DatagramPacket extends java.lang.Object

A) CONSTRUCTORES:
public DatagrarnPacket(byte ibuf[], int ilength)
Implementa un DatagramPacket para la recepción de paquetes de longitud ilength, siendo el valor de este parámetro menor o igual que ibuf.length.
public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)
Implementa un DatagramPacket para el envío de paquetes de longitud ilength al número de puerto especificado en el parámetro iport, del host especificado en la dirección de destino que se le pasa por medio del parámetro iaddr.

B) MÉTODOS:
public InetAddress getAddress()
Retorna la dirección IP del host al cual se le envía el datagrama o del que el datagrama se recibió.
public byte[] getData()
Retorna los datos a recibir o a enviar.
public int getlength()
Retorna la longitud de los datos a enviar o a recibir.
public int getPort()
Retorna el número de puerto de la máquina remota a la que se le va a enviar el datagrama o del que se recibió.

3.3.5. STREAM SOCKET (Servicio Orientado a Conexión)

Es un servicio orientado a conexión donde los datos se transfieren: sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados. El protocolo de comunicaciones con streams es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de sockets. Mientras uno de los sockets atiende peticiones de conexión (servidor), el otro solicita una conexión (cliente). Una vez que los dos sockets estén conectados, se pueden utilizar para transmitir datos en ambas direcciones.
Permite a las aplicaciones Cliente y Servidor, disponer de un stream que facilita la comunicación entre ambos, obteniéndose una mayor fiabilidad.
El funcionamiento es diferente al anterior ya que cada extremo se comportará de forma diferente, el servidor adopta un papel (inicial) pasivo y espera conexiones de los clientes.

Mientras que el cliente adoptará un papel (inicial) activo, solicitando conexiones al servidor.
En la parte del servidor se tiene:
public final class java. net.ServerSocket extends java,lang.Object

A) CONSTRUCTORES:
public ServerSocket (int port) throws lOException
Se crea un socket local al que se enlaza el puerto especificado en el parámetro port, si se especifica un 0 en dicho parámetro creará el socket en cualquier puerto disponible. Puede aceptar hasta 50 peticiones en cola pendientes de conexión por parte de los clientes.
public ServerSocket (int port, int count) throws lOException
Aquí, el parámetro count sirve para que puede especificarse, el número máximo de peticiones de conexión que se pueden mantener en cola.
Hay que recordar, que es fundamental que el puerto escogido sea conocido por el cliente, en caso contrario, no se podría establecer la conexión.

B) MÉTODOS :
public Socket accept () throws IOException
Sobre un ServerSocket se puede realizar una espera de conexión por parte del cliente mediante el método accept(). Hay que decir, que este método es de bloqueo, el proceso espera a que se realice una conexión por parte del cliente para seguir su ejecución. Una vez que se ha establecido una conexión por el cliente, este método devolverá un objeto tipo Socket, a través del cual se establecerá la comunicación con el cliente.
public void close() throws lOException
Se encarga de cerrar el socket.
public InetAddress getlnetAddress ()
Retorna la dirección IP remota a la cual está conectado el socket. Si no lo está retornará null .
public int getLocalPort ()
Retorna el puerto en el que está escuchando el socket.
public String toString( )
Retorna un string representando el socket.

En la parte del cliente :
public final class java.net.Socket extends java.lang.Object

A) CONSTRUCTORES :
public Socket (InetAddress address, int port) throws lOException
Crea un StreamSocket y lo conecta al puerto remoto y dirección IP remota especificados.
public Socket (String host, int port) throws UnKnownHostException IOException
Crea un StreamSocket y lo conecta al número de puerto y al nombre de host especificados.

B) MÉTODOS :
public void close() throws IOException
Se encarga de cerrar el socket.
public InetAddress getlnetAddress ()
Retorna la dirección IP remota a la que se conecta el socket.
public InputStream getlnputStream () throws lOException
Retorna un input stream para la lectura de bytes desde el socket.
public int getLocalPort()
Retorna el puerto local al que está conectado el socket.
Public OutputStream getOutputStream() throwsIOException
Retorna un output stream para la escritura de bytes hacia el socket.
public int getPort ()
Retorna el puerto remoto al que está conectado el socket.

3.2.6 La Clase InetAddress

Esta clase implementa la dirección IP.
public final class java.net.InetAddress extends java.lang.Object

A) CONSTRUCTORES :
Para crear una nueva instancia de esta clase se debe de llamar a los métodos getLocalHost(), getByname() o getAllByName()

B) MÉTODOS:
public boolean equals (Object obj)
Devuelve un booleano a true si el parámetro que se la pasa no es null e implementa la misma dirección IP que el objeto. Dos instancias de InetAddress implementan la misma dirección IP si la longitud del vector de bytes que nos devuelve el método getAddress() es la misma para ambas y cada uno de los componentes del vector de componentes es el mismo que el vector de bytes.

public static InetAddress[] getAllByName(String host) throws UnknownHostExcePtion
Retorna un vector con todas las direcciones IP del host especificado en el parámetro.
public static InetAddress getByName (string host) throws UnknownHostException
Retorna la dirección IP del nombre del host que se le pasa como parámetro. Este parametro puede ser el nombre de la maquina, un nombre *de dominio o una direccion IP.*
public String getHostName()
Retorna el nombre del host para esta dirección IP.
public static InetAddress getLocalHost() throws UnknownHostException
Retorna la dirección IP para el host local.
public int hashCode ()
Retorna un código hash para esta dirección IP.
public String toString ()
Retorna un String representando la dirección IP.

Un ejemplo de utilización muy sencillo de esta clase es el siguiente,que se encarga de devolver la dirección IP de la máquina.

```
public class QuienSoy {
public static void main (String[] args) throws Exception {
        InetAddress direccion= InetAddress.getByName("www.egibide.org");
        System.out.Println(direccion) ;
}}
```

3.2.7. Envio y recepción a través de sockets

El servidor creará un socket, utilizando ServerSocket, le asignará un puerto y una dirección, una vez haga el accept para esperar llamadas, se quedará bloqueado a la espera de las mismas. Una vez llegue una llamada el accept creará un Socket para procesarla.
A su vez, cuando un cliente desee establecer una conexión, creará un socket y establecerá una conexión al puerto establecido. Sólo es en este momento, cuando se da una conexión real y se mantendrá hasta su liberación mediante close().
Para poder leer y escribir datos, los sockets disponen de unos stream asociados, uno de entrada (InputStream) y otro de salida (OutputStream) respectivamente.
Para obtener estos streams a partir del socket utilizaremos:
ObietoDeTipoSocket.getlnputStream ()
Devuelve un objeto de tipo InputStream.
ObjetoDeTipoSocket.getOutputStrearn ()
Devuelve un objeto de tipo OutputStream.
Para el envío de datos, puede utilizarse OutputStream directamente en el caso de que se quiera enviar un flujo de bytes sin bufer o también puede crearse un objeto de tipo stream basado en el OutputStream que proporciona el socket.
En Java, crear una conexión socket TCP/IP se realiza directamente con el paquete java.net.
El servidor establece un puerto y espera 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 port#. El cliente y el servidor se comunican con manejadores InputStream y OutPutStream.

Si se está programando un **cliente**, el socket se abre de la forma:
Socket miSocket;
miSocket=new Socket(host,puerto);

Donde host es el nombre de la máquina sobre la que se está intentando abrir la conexión y puerto es el puerto (un número) que el servidor está atendiendo. Cuando se selecciona un número de puerto, se debe tener en cuenta que los puertos en el rango 0-1023 están reservados. Estos puertos son los que utilizan los servicios estándar del sistema como email, ftp, http, etc. Por lo que, para aplicaciones de usuario, el programador deberá asegurarse de seleccionar un puerto por encima del 1023.Hasta ahora no se han utilizado excepciones; pero deben tener en cuenta la captura de excepciones cuando se está trabajando con sockets. Así:

```
Socket miSocket;
try {
misocket= new Socket(host, puerto);
}catch(IOExcePtion e) {
System.out.println (e ) ;
}catch (UnknownHostException uhe) {
System.out.println (uhe) ;
}
```

En el caso de estar implementando un **servidor**, la forma de apertura del socket seria como sigue:

```
SocketServer socketSrv;
try{
socketSrv = new ServerSocket( puerto ) ;
}catch(IOException e) {
System.out.println (e ) ;
}
```

Cuando se implementa un servidor se necesita crear un objeto Socket a partir del ServerSocket, para que éste continúe ateniendo las conexiones que soliciten potenciales nuevos clientes y poder servir al cliente, recién conectado, a través del *Socket creado:*

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

3.2.8. Creación de streams

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 );
}
```

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.

3.2.9. Cierre de sockets

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

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

Y en la parte del servidor:

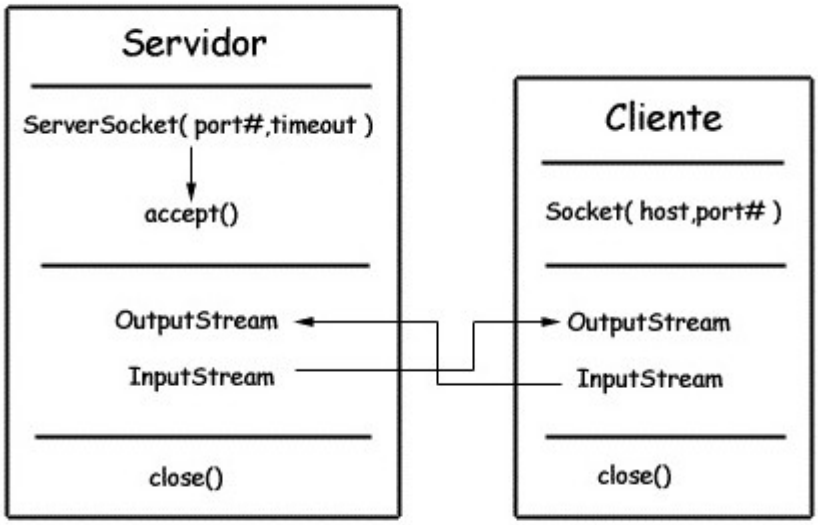
```
try {
salida.close();
entrada.close();
socketServicio.close();
miServicio.close();
} catch( IOException e ) {
System.out.println( e );
}
```

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.

3.2.10 Diferencias entre sockets stream y datagrams

Ahora se presenta un problema, ¿qué protocolo, o tipo de sockets, debe utilizarse - UDP o TCP? La decisión depende de la aplicación cliente/servidor que se esté escribiendo. Se muestran, a continuación, algunas diferencias entre los protocolos para ayudar en la decisión.
En UDP, cada vez que se envía un datagrama, hay que enviar también el descriptor del socket local y la dirección del socket que va a recibir el datagrama, luego éstos son más grandes que los TCP. Como el protocolo TCP está orientado a la conexión, tiene que establecerse esta conexión entre los dos sockets antes de nada, lo que implica un cierto tiempo empleado en el establecimiento de la conexión que no existe en UDP.
En UDP hay un límite de tamaño de los datagramas, establecido en 64 kilobytes, que se pueden enviar a una localización determinada, mientras que TCP no tiene limite; una vez que se ha establecido la conexión, el par de sockets funciona como los streams: todos los datos se leen inmediatamente, en el mismo orden en que se van recibiendo.
UDP es un protocolo desordenado, no garantiza que los datagramas que se hayan enviado sean recibidos en el mismo orden por el socket de recepción. Al contrario, TCP es un protocolo ordenado, garantiza que todos los paquetes que se envíen serán recibidos en el socket destino en el mismo orden en que se han enviado.
Los datagramas son bloques de información del tipo lanzar y olvidar. Para la mayoría de los programas que utilicen la red, el usar un flujo TC.P en vez. de un datagrama UDP es más sencillo y hay menos posibilidades de tener problemas. Sin embargo, cuando se requiere un rendimiento óptimo, y está justificado. el tiempo adicional que supone realizar la verificación de los datos, los datagramas son un mecanismo realmente útil.
En resumen, TCP parece más indicado para la implementación de servicios de red como un control remoto (rlogin, telnet) y transmisión de ficheros (ftp),que necesitan transmitir datos de longitud indefinida. UDP es menos complejo y tiene una menor sobrecarga sobre la conexión, esto hace que sea el indicado en la implementación de aplicaciones cliente/servidor en sistemas distribuidos montados sobre redes de área local.

3.3 Esquema del funcionamiento de los sockets TCP



3.3.1 Cliente

- Abrir la *conexión* (Socket).
- Crear los flujos de entrada y salida asociados a dicha conexión.
- Leer y escribir de los flujos de entrada y salida de acuerdo al protocolo de aplicación que hayamos definido.
- Cerrar los flujos de entrada y salida.
- Cerrar la conexión.

3.3.2 Servidor

- Se crea el *servidor de escucha* (SocketServer).
- Se bloquea esperando una *conexión* (Socket).
  - El bloqueo puede tener un cierto *timeout.*
- Una vez conectado un cliente, se crean los flujos de entrada y salida asociados a la conexión.
- Se lee y escribe en ellos en base al protocolo de aplicación que hayamos definido.
- Se cierran los flujos de entrada y salida, y también la conexión.
- Se cierra el servidor de escucha.

Ejemplo 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.

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.

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);
BufferedReader datos =new BufferedReader( new
InputStreamReader(              servidor.getInputStream() ) );


System.out.println(datos.readLine());
servidor.close();
}
catch (Exception e)
{
e.printStackTrace();
}
}
}
```

3.4 Gestión de sockets UDP

En los sockets UDP no se establece conexión. Los roles cliente-servidor están un poco mas difusos que en el caso de TCP. Podemos considerar al servidor como el que espera un mensaje y responde; y al cliente como el que inicia la comunicación. Tanto uno como otro si desean ponerse en contacto necesitan saber en que ordenador y en que puerto esta escuchando el otro.
En el flujo de comunicación entre cliente-servidor usando UDP, ambos necesitan crear un socket DatagramSocket:

- El servidor crea un socket asociado a un puerto local para escuchar peticiones de clientes. Permanece a la espera de recibir peticiones.
- El cliente creara un socket para comunicarse con el servidor. Para enviar datagramas necesita conocer su IP y el puerto por el que escucha. Utilizara el método send() del socket para enviar la petición en forma de datagrama.
- El servidor recibe peticiones mediante el método receive() del socket. En el datagrama va incluido ademas el mensaje, el puerto y la IP del cliente emisor de la petición; lo que permite al servidor conocer la dirección del emisor del datagrama. Utilizando el método send() del socket puede enviar la respuesta al cliente emisor.
- El cliente recibe la respuesta del servidor mediante el método receive() del socket.
- El servidor permanece a la espera de recibir mas peticiones.

3.4.1 La clase DatagramSocket

Tanto cliente como servidor, para ponerse a la escucha de un puerto UDP, únicamente tienen que instanciar la clase DatagramSocket, pasándole los parámetros adecuados:

- El número de puerto que están escuchando.
- El InetAddress del ordenador en el que se esta ejecutando el programa, habitualmente InetAddress.getLocalHost();

Por ello, el código del servidor es tan sencillo como esto

ServidorUdp.java

```
DatagramSocket socket = new DatagramSocket ( puerto_del_servidor);
```

y el del **cliente** se parece bastante

ClienteUdp.java

DatagramSocket socket = new DatagramSocket ();

3.4.2 La clase DatagramPacket

Esta es la clase que se va a enviar y recibir como mensaje. Lleva dentro un array de bytes que es el que debemos rellenar con lo queremos enviar o en el que estará lo que hemos recibido.

Dependiendo de si es para enviar o recibir, esta clase se instancia de forma distinta. En ambos casos hay que pasarle el array de bytes. En el caso de enviar, hay que pasarle además la InetAdress del destinatario y el puerto en el que está escuchando el destinatario.

Para enviar

DatagramPacket dato = new DatagramPacket(
mensajeEnBytes, // El array de bytes
mensajeEnBytes.length, // Su longitud
InetAddress.getByName("localhost"), // Destinatario
puertodelservidor);// Puerto del destinatario

socket.send(dato);

Para recibir

```
DatagramPacket dato = new DatagramPacket(new byte[100], 100);
socket.receive(dato);
```

EJEMPLO

El servidor implementado escucha por el puerto de comunicaciones 8050, cuando le llega un datagrama procedente de algún cliente, imprime la dirección del cliente, y el puerto por la salida estandar del sistema, y luego envia un datagrama de bienvenida del sistema.

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

```
 import java.util.*;


public class Server{
public static void main(String[] args) throws IOException {
//el puerto de escucha del servidor será el 8050
int PUERTO=8050;

byte msg[]=new byte[1024];

//Creamos el socket UDP del servidor en el puerto asociado
DatagramSocket s = new DatagramSocket(PUERTO);
 System.out.println("Servidor Activo");

//implementacion del protocolo del servidor en un bucle infinito
while (true) {
DatagramPacket recibido = new DatagramPacket(new byte [1024],1024);

//llega un datagrama
 s.receive(recibido);
 System.out.println("Ha llegado una peticion \n");
 System.out.println("Procedente de :" + recibido.getAddress());
 System.out.println("En el puerto :" + recibido.getPort());
 System.out.println("Sirviendo la petición");

//se prepara el mensaje a enviar
 String message=new String("Bienvenido, conexión realizada”);
 msg=message.getBytes();

//se crea el datagrama que contendrá al mensaje
DatagramPacket paquete=new
DatagramPacket(msg,msg.length,recibido.getAddress(),

recibido.getPort());

//se le envia al cliente
 s.send(paquete);

 }
 } }
```

3.5 MulticastSocket

A veces nos interesa que un ordenador pueda enviar un mensaje por red y que este sea recibido por otros ordenadores simultáneamente. Para ello están las direcciones multicast. Son direcciones en el rango 224.0.0.0 a 239.255.255.255. La 224.0.0.0 está reservada y no puede usarse. Enviando mensajes por estas direcciones,

cualquier otro ordenador en la red que las escuche podría leer dicho mensaje, independientemente de cual sea la IP real de ese ordenador. Es decir, si un ordenador quiere enviar un mensaje simultáneamente a varios, puede hacerlo enviando el mensaje a una de estas IPs, los demás ordenadores deben estar a la escucha de dichas IPs para recibir el mensaje.
La clase MulticastSocket es util para enviar paquetes a multiples destinos simultáneamente. Para poder recibir estos paquetes es necesario establecer un grupo multicast, que es un grupo de direcciones IP que comparten el mismo numero de puerto.

3.5.1 Envio de un mensaje Multicast en java

```
MulticastSocket servidor = new MulticastSocket();
// El dato que queramos enviar en el mensaje, como array de bytes.
byte [] dato = new byte[] {1,2,3,4};
// Usamos la direccion Multicast 230.0.0.1, por poner alguna dentro del rango
// y el puerto 55557, uno cualquiera que esté libre.
DatagramPacket dgp = new DatagramPacket(dato, dato.length,
InetAddress.getByName("230.0.0.1"), 55557);
// Envío
servidor.send(dgp);
```

3.5.2 Lectura de un mensaje Multicast en java

```
// El mismo puerto que se uso en la parte de enviar.
MulticastSocket escucha = new MulticastSocket(55557);
// Nos ponemos a la escucha de la misma IP de Multicast que se uso en la parte de
enviar.
escucha.joinGroup(InetAddress.getByName("230.0.0.1"));
// Un array de bytes con tamaño suficiente para recoger el mensaje enviado,
// bastaría con 4 bytes.
byte [] dato = new byte [1024];
// Se espera la recepción. La llamada a receive() se queda
// bloqueada hasta que llegue un mesnaje.
DatagramPacket dgp = new DatagramPacket(dato, dato.length);
escucha.receive(dgp);
// Obtención del dato ya relleno.
byte [] dato = dgp.getData();
```

3.6 Envío de objetos a través de sockets

En java podemos intercambiar objetos entre programas emisor y receptor o entre programas cliente y servidor utilizando sockets.

3.6.1 Objetos en Sockets TCP
Las clases ObjectInputStream y ObjectOutput Stream nos permiten enviar objetos a traves de sockets TCP. Utilizaremos los metodos readObject() para leer el objeto del stream y writeObject() para escribir el objeto al stream. Usaremos el constructor que admite un InputStream y un OutputStream. Para preparar el flujo de salida para escribir objetos:

```
ObjectOutputStream outObjeto= new
ObjectOutputStream(nuevoObjeto.getOutputStream());
```

Para preparar el flujo de entrada para leer objetos:

```
ObjectInputStream inObjeto=new ObjectInputStream(nuevoObjeto
.getInputStream());
```

Las clases a las que pertenecen los objetos que queremos enviar deben implementar la interfaz Serializable. La serialización consiste en convertir un objeto en una secuencia de bytes para guardarlo en un archivo o enviarlo por la red, y luego reconstruirlo, con los valores que tenía al ser serializado, para su posterior utilización.
En Java, esta capacidad de serialización, es decir, de guardar información sobre un objeto para luego recuperarla, se llama persistencia.
Como la interfaz Serializable no tiene métodos, es muy sencillo implementarla, basta con un *implements Serializable* y nada más. Si dentro de la clase hay atributos que son otras clases, éstos a su vez también deben ser Serializable.

A la hora de crear los flujos de datos, obligatoriamente primero el de ObjectOutputStream antes que el ObjectInputStream.

```
public class Datos implements Serializable
{
  public int a;
  public String b;
  public char c;
}
```

3.6.2 Objetos en Sockets UDP
Para intercambiar objetos en sockets UDP utilizaremos las clases ByteArrayOutputStream y ByteArrayInputStream. Se necesita convertir el objeto a un array de bytes. Para ello:

```
Persona persona= new Persona("Maria",22);
//convertimos objeto a bytes
ByteArrayOutputStream bs= new ByteArrayOutputStream ();
ObjectOutputStream out= new ObjectOutputStream (bs);
out.writeObject(persona);//escribir objeto Persona en el Stream
out.close();//cerrar stream
byte[] bytes= bs.toByteArray();//objeto en bytes
```

Para convertir los bytes recibidos en el objeto:

```
//recibo datagrama
byte[] recibidos= new byte[1024];
DatagramPacket paqRecibido= new DatagramPacket(recibidos, recibidos.length);
socket.receive(paqRecibido);//recibo al datagrama
/ /convertimos bytes a objeto
ByteArrayInputStream bais= new ByteArrayInputStream (recibidos);
ObjectInputStream in= new ObjectInputStream (bais);
Persona persona=(Persona) in.readObject();//obtengo objeto
```

3.7 Conexión de múltiples clientes. Hilos

Hasta ahora los programas servidores que hemos creado solo son capaces de atender a un cliente en cada momento, pero lo mas típico es que un programa servidor pueda atender a muchos cliente simultáneamente. La solución es el multihilo, cada cliente sera atendido por un hilo.

El esquema básico en sockets TCP seria construir un único servidor con la clase ServerSocket e invocar al método accept() para esperar las peticiones de conexión de los clientes. Cuando un cliente se conecta, el método accept() devuelve un objeto Socket, este se usara para crear un hilo cuya misión es atender a este cliente. Después se vuelve a invocar accept() para esperar a un nuevo cliente , habitualmente la espera de conexiones se hace dentro de un bucle infinito:

```
public class EcoServ
{
  public static void main(String args[]) throws IOException
  {

    ServerSocket s;
```

```
    Socket c;
    s= new ServerSocket(6000);
    System.ou.println("SErvidor iniciado");
    while(true)
    {
      c = s.accept();    //esperando cliente
      HiloServidor hilo=new HiloServidor(c);
      hilo.start();
    }
    ……
  }
```

Todas las operaciones que sirven a un cliente quedan dentro de la clase hilos. El hilo permite que el servidor se mantenga a la escucha de peticiones y no interrumpa su proceso mientras los clientes son atendidos.

```
class HiloServidor extends Thread
{
  Socket sock=null;
  public HiloServidor(Socket sock)   //constructor
  {
    this.sock=sock;
    // se crean los flujos de entrada y salida
fsalida=new PrintWriter(sock.getOutputStream(),true);
fentrada= new BufferedReader (new InputStreamReader(sock.getInputStream()));
  }
  public void run()                  //tarea a realizar  con los clientes
  {
    //....   }}
```