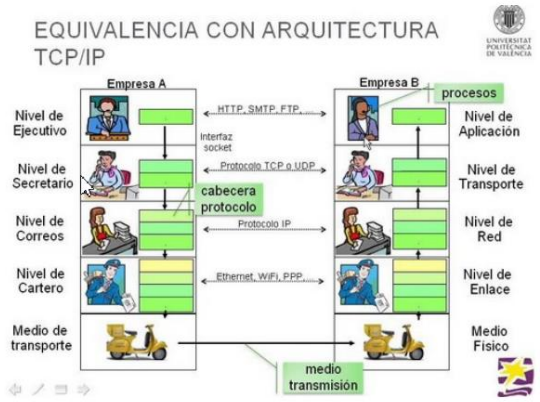
UNIDAD 3: Comunicaciones en Red

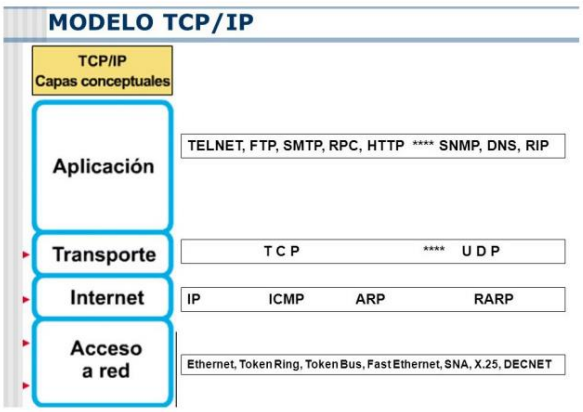
Java dispone de clases para establecer conexiones , crear servidores, enviar y recibir datos, y para el resto de las operaciones utilizadas en las comunicaciones a través de redes de ordenadores. Además, el uso de hilos, nos va a permitir la manipulación simultánea de múltiples conexiones.

1.- Clases java para comunicaciones en Red

TCP/IP es una familia de protocolos desarrollados para permitir la comunicación entre cualquier par de ordenadores de cualquier red o fabricante, respetando los protocolos de cada red individual. Tiene cuatro capas:

- Capa de aplicación: en este nivel se encuentran las aplicaciones disponibles para los usuarios. Por ejemplo, FTP, SMTP, Telnet, HTTP, etc.
- Capa de transporte: suministra a las aplicaciones servicio de comunicaciones extremo a extremo utilizando dos tipos de protocolos: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol).
- Capa de red: tiene como propósito seleccionar la mejor ruta para enviar paquetes por la red. El protocolo principal que funciona en esta capa es el Protocolo de Internet (IP).
- Capa de enlace o interfaz de red: es la interfaz con la red real. Recibe los datagramas de la capa de red y los transmite al hardware de la red.





Los equipos conectados a Internet se comunican entre si utilizando el protocolo TCP o UDP. Cuando se escriben aplicaciones Java que se comunican a través de la red, se está programando en la capa de aplicación . Normalmente , no es necesario preocuparse por las capas TCP y UDP , usando en su lugar las clases del paquete java.net. Sin embargo, hay diferencias entre la capa TCP y UDP que debemos conocer para decidir qué clase usar en los programas:

- **TCP**: Protocolo basado en la conexión, garantiza que los datos enviados desde un extremo de la conexión llegan al otro extremo y en el mismo orden en que fueron enviados. De lo contrario, se notifica un error.

- **UDP**: No está basado en la conexión como TCP. Envía paquetes de datos independientes, denominados datagramas, de una aplicación a otra; el orden de entrega no es importante y no se garantiza la recepción de los paquetes enviados.

El paquete java.net contiene clases e interfaces para la implementación de aplicaciones de red. Estas incluyen:

- La clase URL , Uniform Resource Locator (Localizador Uniforme de Recursos): Representa un puntero a un recurso de la web.

- La clase URL Connection: admite operaciones más complejas en las URL.
- Las clases ServerSocket y Socket: para dar soporte a sockets TCP.

  - **ServerSocket**: utilizada por el programa servidor para crear un socket en el puerto en el que escucha las peticiones de conexión de los clientes.
  - **Socket**: utilizada tanto por el cliente como por el servidor para comunicarse entre si leyendo y escribiendo datos usando streams.

- Las clases DatagramSocket, MulticastSocket y DatagramPacket: para dar soporte a la comunicación via datagramas UDP.

- La clase InetAddress: que representa las direcciones de internet.

1.1.- Los Puertos

Los protocolos TCP y UDP usan puertos para asignar datos entrantes a un proceso en particular que se ejecuta en un ordenador.

En términos generales, un ordenador tiene una única conexión física a la red. Los datos destinados a este ordenador llegan a través de esa conexión . Sin embargo, los datos pueden estar destinados a diferentes aplicaciones que se ejecutan en el ordenador. Entonces, ¿cómo sabe el ordenador a qué aplicación enviar los datos? Mediante el uso de puertos.

Los datos transmitidos a través de Internet van acompañados de información de direccionamiento que identifica la máquina y el puerto para el que está destinada. La máquina se identifica por su dirección IP . Los puertos se identifican mediante un número de 16 bits , que TCP y UDP utilizan para entregar los datos a la aplicación correcta.

En la comunicación basada en TCP , una aplicación de servidor vincula un socket a un número de puerto específico. Esto tiene el efecto de registrar el servidor en el sistema para recibir todos los datos destinados a ese puerto. Una aplicación cliente puede entonces comunicarse con el servidor enviándole peticiones a través de ese puerto.

En la comunicación basada en datagramas, como UDP , el paquete de datagrama contiene el número de puerto de su destino y UDP enruta el paquete a la aplicación adecuada.

1.2.- La clase InetAddress

La clase InetAddress es la abstracción que representa una dirección IP . Tiene dos subclases:

- Inet4Address para direcciones IPv4
- Inet6Address para direcciones IPv6

MÉTODOS	MISIÓN
InetAddress getLocalHost()	Devuelve un objeto *InetAddress* que representa la dirección IP de la máquina donde se esta ejecutando el programa.
InetAddress getByName(String host)	Devuelve un objeto *InetAddress* que representa la dirección IP de la máquina que se especifica como parámetro (*host*). Este parámetro puede ser el nombre de la máquina, un nombre de dominio o una dirección IP.
InetAddress[] getAllByName(String host)	Devuelve un array de objetos de tipo *InetAddress*. Este método es útil para averiguar todas las direcciones IP que tenga asignada una máquina en particular.
String getHostAddress()	Devuelve la dirección IP de un objeto *InetAddress* en forma de cadena.
String getHostName()	Devuelve el nombre del host de un objeto *InetAddress*.
String getCanonicalHostName()	Obtiene el nombre canónico completo (suele ser la dirección real del host) de un objeto *InetAddress*.

Los tres primeros métodos pueden lanzar la excepción UnknownHostException (Cuando se lanza esta excepción , nos indica que no se pudo determinar la dirección IP del host).

La forma más típica de crear instancias de InetAddress, es invocando al método estático getByName(String) pasándole el nombre DNS del host como parámetro. Este objeto representará la dirección IP de ese host , y se podrá utilizar para construir sockets.

1.3.- La clase URL

La clase URL (Uniform Resource Locator "Localizador uniforme de recursos") representa un puntero a un recurso en la Web. Un recurso puede ser un fichero, un directorio, o una referencia a un objeto más complicado, como una consulta a una base de datos o a un motor de búsqueda.

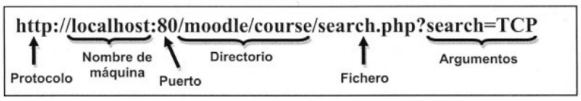
En general una URL que localiza recursos empleando el protocolo HTTP se divide en varias partes:

http://host[:puerto][/nombre del path del servidor][?argumentos]

Las partes encerradas entre corchetes son opcionales

- **host**: Es el nombre de la maquina en la que reside el recurso.
- **[:puerto]** Número de puerto en el que el servidor escucha las peticiones. Este parámetro es opcional y si no se indica, se considera el puerto por defecto (para el protocolo http es el puerto 80)
- **[/dirección del servidor]** Es el path o directorio donde se encuentra el recurso en el sistema de ficheros del servidor. Si no se indica, se proporciona la página por defecto del servidor web.
- **[?argumentos]** Parámetros que se envía al servidor. Por ejemplo, cuando realizamos una consulta se puede enviar parámetros a un fichero PHP para procesarla.

Ejemplo:



El fichero "search.php" está en el directorio "/moodle/course" dentro del servidor, y el argumento "search=TCP" que se envía al fichero "search.php" para realizar una búsqueda.

Constructores de la clase URL

CONSTRUCTOR	MISIÓN
URL (String url)	Crea un objeto URL a partir del String indicado en *url*.
URL(String protocolo, String host, String fichero)	Crea un objeto URL a partir de los parámetros *protocolo, host* y *fichero* (o directorio).
URL(String protocolo, String host, int puerto, String fichero)	Crea un objeto URL en el que se especifica el *protocolo*, *host*, *puerto* y *fichero* (o directorio) representados mediante String.
URL(URL contexto, String especificación)	Crea un objeto URL analizando la especificación dada dentro de un contexto específico.

Estos constructores pueden lanzar la excepción "MalformedURLException" si la URL está mal construida, no se hace ninguna verificación de que realmente exista la máquina o el recurso en la red.

Algunos Métodos de la clase URL

MÉTODOS	MISIÓN
String getAuthority ()	Obtiene la autoridad del objeto URL.
int getDefaultPort()	Devuelve el puerto asociado por defecto al objeto URL.
int getPort()	Devuelve el número de puerto de la URL, -1 si no se indica.
String getHost()	Devuelve el nombre de la máquina.
String getQuery()	Devuelve la cadena que se envía a una página para ser procesada (es lo que sigue al signo? de una URL).
String getPath()	Devuelve una cadena con la ruta hacia el fichero desde el servidor y el nombre completo del fichero.
String getFile()	Devuelve lo mismo que *getPath ()*, además de la concatenación del valor de *getQuery()* si lo hubiese. Si no hay una porción consulta, este método y *getPath()* devolverán los mismos resultados.
String getProtocol()	Devuelve el nombre del protocol asociado al objeto URL.
String getUserInfo()	Devuelve la parte con los datos del usuario o nulo si no existe.
InputStream openStream()	Devuelve un **InputStream** del que podremos leer el contenido del recurso que identifica la URL.
URLConnection openConnection()	Devuelve un objeto **URLConnection** que nos permite abrir una conexión con el recurso y realizar operaciones de lectura y escritura sobre él.

1.4.- LA CLASE URLConnection

Una vez que tenemos un objeto de la Clase URL , si se invoca al método openConnection() para realizar la comunicación con el objeto y la conexión se establece satisfactoriamente , entonces tenemos una instancia de un objeto de la clase URLConnetion.

Las instancias de esta clase se pueden utilizar tanto para leer como para escribir al recurso referenciado por la URL. Puede lanzar la excepción IOException

Ejemplo:

```
URL url = new URL ("http://www.elaltozano.es");
URLConnection urlCon = url.openConnection();
```

Algunos métodos de la clase URLConnection

MÉTODOS	MISIÓN
InputStream getInputStream()	Devuelve un objeto **InputStream** para leer datos de esta conexión.
OutputStream getOutputStream()	Devuelve un objeto **OutputStream** para escribir datos en esta conexión.
void setDoInput (boolean b)	Permite que el usuario reciba datos desde la URL si el parámetro *b* es *true* (por defecto está establecidoa *true*)
void setDoOutput((boolean b)	Permite que el usuario envíe datos si el parámetro *b* es *true* (no está establecido al principio)
void connect()	Abre una conexión al recurso remoto si tal conexión no se ha establecido ya.
int getContentLength()	Devuelve el valor del campo de cabecera *content-lenght* o -1 si no está definido.
String getContentType()	Devuelve el valor del campo de cabecera *content-type* o null si no está definido.
long getDate()	Devuelve el valor del campo de cabecera *date* o 0 si no está definido.
long getLastModified()	Devuelve el valor del campo de cabecera *last-modified*
String getHeaderField(int n)	Devuelve el valor del enésimo campo de cabecera especificado o null si no está definido.
Map< String, List<String> > getHeaderFields()	Devuelve una estructura Map (estructura de Java que nos permite almacenar pares clave/valor.) con los campos de cabecera. Las claves son cadenas que representan los nombres de los campos de cabecera y los valores son cadenas que representan los valores de los campos correspondientes.
URL getURL()	Devuelve la dirección URL.

Formularios

Gran cantidad de páginas HTML contienen formularios a través de los cuales podemos solicitar información a un servidor rellenando los campos requeridos y pulsando el botón de envío. El servidor recibe la petición, la procesa y envía los datos solicitados al cliente normalmente en formato HTML. Por ejemplo, tenemos una página HTML que contiene un formulario con dos campos de entrada y un botón. En el atributo "action" se indica el tipo de acción que va a realizar el formulario, en este caso los datos se envían a un script PHP de nombre "vernombre.php". Con (method=post) indicamos la forma en que se envía el formulario.

```
<html>
<body>
<form action="vernombre.php" method="post">
<p> Escribe tu nombre:
<input name="nombre" type="text" size="15"> </p>
<p> Escribe tus apellidos:
<input name="apellidos" type="text" size="15"></p>
<input type="submit" name="ver" value="Ver">
</form>
</body>
</html>
```

El script PHP que recibe los datos del formulario es el siguiente:

```
<?php
$nom=$_POST["nombre"];
$ape=$_POST["apellidos"];
echo "El nombre recibido es: $nom, y ";
echo "los apellidos son: $ape ";
?>
```

http://localhost/2021/vernombre.php?nombre=Juan&apellidos=Martínez&ver=Ver

Desde Java, usando la clase URLConnection podemos interactuar con scripts del lado del servidor y podemos enviar valores a los campos del script sin necesidad de abrir un formulario HTML, será necesario escribir en la URL para dar los datos al script.

¿Qué tiene que realizar nuestro programa?

- **Crear el objeto URL** al script con el que va a interactuar. Por ejemplo, en nuestra máquina local tenemos instalado un servidor web Apache y dentro de htdocs tenemos la carpeta 2021 con el script PHP "vernombre.php", la URL sería la siguiente:

  URL = new URL(http://localhost/2021/vernombre.php)

- **Abrir una conexión con la UR**L, es decir obtener el objeto URLConnection:

  URLConnection conexión = url.openConnection()

- **Configurar la conexión para que se puedan enviar datos** usando el método setDoOutput():

  conexión.setDoOutput(true)

- **Obtener un stream de salida** sobre la conexión

  PrintWriter output = new PrintWriter(conexion.getOutputStream())

- **Escribir en el stream de salida**, en este caso mandamos una cadena con los datos que necesita el script

  output.write(cadena)

  La cadena tiene el siguiente formato:

  parámetro=valor

  Si el script recibe varios parámetros sería:

  parámetro1=valor1&parámetro2=valor2&parámetro3=valor3 , y así sucesivamente

- **Cerrar el stream de salida**

  output.close()

Nota: Normalmente cuando se pasa información a algún script PHP, éste realiza alguna acción y después envía la información de vuelta por la misma URL. Por tanto si queremos ver lo que devuelve será necesario leer desde la URL. Para ello se abre un stream de entrada sobre esa conexión mediante el método getInputStream()

BufferedReader reader = new BufferedReader(new InputStreamReader(conexion.getInputStream())))

Después se realiza la lectura para obtener los resultados devueltos por el script

UNIDAD 3: Sockets tipos de sockets

Direcciones IP para redes privadas

Tipo	Longitud de la mascara	Redes posible	Rango	Dirección Broadcast
A	8	1 red de tipo A 10.0.0.0 /8	10.0.0.1 10.255.255.254	10.255.255.255
B	16	16 posibles redes tipo B 172.x.0.0/16 con 16<= x <=31	172.x.0.1 172.x.255.254	172.x.255.255
C	24	256 posibles redes C 192.168.x.0 /24 0<= x <= 255	192.168.x.1 192.168.x.254	192.168.x.255

Nivel de transporte

La comunicación se realiza entre un puerto y un host . El puerto se identifica por 16 bits (0 – 65535). La combinación de una IP con un puerto forma un socket. La comunicación a nivel de transporte se realiza a través de dos sockets.

1.- Sockets: Tipos de Sockets

1.1.-Qué son los sockets

Los protocolos TCP y UDP utilizan el concepto de sockets para proporcionar los puntos extremos de la comunicación entre aplicaciones o procesos. La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso, a este conector es a lo que se le denomina socket.

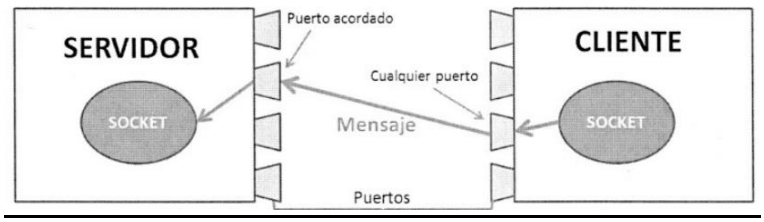
Los socket permiten que un proceso emita o reciba información con otro proceso, incluso estando en otra máquina. Un socket queda definido por un par de IP (local y remota) , un protocolo de transporte (TCP o UDP) y un par de números de puertos (local y remoto).

Para que dos aplicaciones puedan comunicarse entre si es necesario que se cumpla lo siguiente:

- Que un programa sea capaz de localizar al otro.
- Que ambas aplicaciones sean capaces de intercambiarse cualquier secuencia de octetos.

Para los procesos receptores de mensajes, su conector debe tener asociado dos campos:

- La dirección IP del host en el que la aplicación está corriendo.
- El puerto local a través del cual la aplicación se comunica y que identifica el proceso.



Rangos de puertos

Rango de puertos	Asignación
Puerto del Sistema 0 -1023	Asignados a servicios o protocolos de nivel de aplicación Ejemplo: FTP puerto 22 y 23 TCP HTTP puerto 80 TCP y el 443 para HTTPS DNS puerto 53 UDP
Puertos registrados 1024-49151	Reservados por empresas y organizaaciones para sus propios servicios Ejemplo: Servidor de base de datos MySQL acepta peticiones por el puerto 3306 TCP
Puertos efímeros 49152 - 65535	Pueden usarlos libremente procesos clientes y servidores. Los programas servidores ofrecen sus servicios en puertos en los anteriores rangos (de sistema o registrados). Los procesos clientes se comunican con ellos desde puertos efímeros. *Cuando un proceso servidor con el protocolo TCP recibe una petición de conexión de un proceso cliente , reserva un puerto efímero para la comunicación con el cliente, confirma la conexión con el cliente y le indica ese número de puerto . A partir de ese momento, la comunicación entre los procesos cliente-servidor se realiza a través del puerto efímero dejando el otro puerto a la escucha para recibir nuevas peticiones*

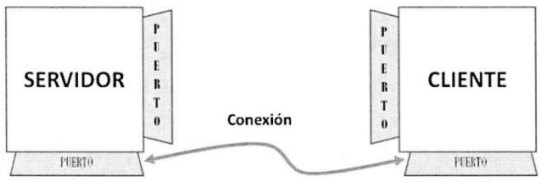
1.2.- Funcionamiento en general de un socket

Un puerto es un punto de destino que identifica hacia qué aplicación o proceso deben dirigirse los datos. Normalmente en una aplicación cliente-servidor, el programa servidor se ejecuta en una máquina específica y tiene un socket que está unido a un número de puerto específico. El servidor queda a la espera “escuchando” las solicitudes de conexión de los clientes sobre ese puerto.

El programa cliente conoce el nombre de la máquina en la que se ejecuta el servidor y el número de puerto por el que escucha las peticiones. Para realizar una solicitud de conexión, el cliente realiza la petición a la máquina a través del puerto.

El cliente también debe identificarse ante el servidor por lo que durante la conexión se utilizará un puerto local asignado por el sistema.

Si todo va bien, el servidor acepta la conexión. Una vez aceptada, el servidor obtiene un nuevo socket sobre un puerto diferente. Esto se debe a que por un lado debe seguir atendiendo las peticiones de conexión mediante el socket original y por otro debe atender las necesidades del cliente que se conectó.



En el lado cliente, si se acepta la conexión , se crea un socket y el cliente puede utilizarlo para comunicarse con el servidor. Este socket utiliza un número de puerto diferente al usado para

conectarse al servidor. El cliente y el servidor pueden ahora comunicarse escribiendo y leyendo por sus respectivos sockets.

1.3.- Tipos de socket

Sockets orientados a conexión (TCP)

Los procesos que se van a comunicar deben establecer antes una conexión mediante un stream. Un stream es una secuencia ordenada de unidades de información (bytes, caracteres, etc.) que puede fluir en dos direcciones: hacia fuera de un proceso (de salida) o hacia dentro de un proceso (de entrada). Están diseñados para acceder a los datos de manera secuencial.

Una vez establecida la conexión , los procesos leen y escriben en el stream sin tener que preocuparse de las direcciones de internet ni de los números de puerto. El establecimiento de la conexión implica:

- Una petición de conexión desde el proceso cliente al proceso servidor.
- Una aceptación de la conexión del proceso servidor al proceso cliente.

En Java hay dos tipos de stream sockets que tienen asociadas las clases Socket para implementar el cliente y ServerSocket para el servidor.

Sockets no orientados a conexión (UDP)

Este tipo de conexión no es fiable y no se garantiza que la información enviada llegue a su destino. Los datagramas se transmiten desde un proceso emisor a otro receptor sin que se haya establecido previamente una conexión , sin acuse de recibo ni reintentos.

Cualquier proceso que necesite enviar o recibir mensajes debe crear primero un conector asociado a una dirección IP y a un puerto local. El servidor enlazará su conector a un puerto de servidor conocido por los clientes. El cliente enlazará su conector a cualquier puerto local libre. Cuando un receptor recibe un mensaje , se obtiene además del mensaje , la dirección IP y el puerto del emisor, permitiendo al receptor enviar la respuesta correspondiente al emisor.

Los sockets UDP se usan cuando una entrega rápida es más importante que una entrega garantizada, o en los casos en que se desea enviar tan poca información que cabe en un único datagrama. Se usan en aplicaciones para la transmisión de audio y vídeo en tiempo real donde no es posible el reenvío de paquetes retrasados. Para implementar en Java este tipo de sockets se utilizan las clases DatagramSocket y DatagramPacket.

1.4.- Clases para sockets TCP

➢ Clase ServerSocket

Esta clase se utiliza para implementar el extremo de la conexión que corresponde al servidor, donde se crea un conector en el puerto de servidor que escucha las peticiones de conexión de los clientes.

Algunos de los constructores de esta clase son los siguientes. Hay que tener en cuenta que pueden lanzar la excepción IOException.

CONSTRUCTOR	MISIÓN
ServerSocket()	Crea un socket de servidor sin ningún puerto asociado.
ServerSocket(int port)	Crea un socket de servidor, que se enlaza al puerto especificado.
ServerSocket(int port, int máximo)	Crea un socket de servidor y lo enlaza con el número de puerto local especificado. El parámetro *máximo* especifica, el número máximo de peticiones de conexión que se pueden mantener en cola.
ServerSocket(int port, int máximo, InetAddress direc)	Crea un socket de servidor en el puerto indicado, especificando un máximo de peticiones y conexiones entrantes y la dirección IP local.

Algunos de los métodos de esta clase son:

MÉTODOS	MISIÓN
Socket accept ()	El método **accept()** escucha una solicitud de conexión de un cliente y la acepta cuando se recibe. Una vez que se ha establecido la conexión con el cliente, devuelve un objeto de tipo **Socket**, a través del cual se establecerá la comunicación con el cliente. Tras esto, el **ServerSocket** sigue disponible para realizar nuevos ***accept()***. Puede lanzar ***IOException***.
close ()	Se encarga de cerrar el **ServerSocket**.
int getLocalPort ()	Devuelve el puerto local al que está enlazado el **ServerSocket**.

Ejemplo:
Crea un socket de servidor y lo enlaza al puerto 6000, visualiza el puerto por el que se esperan las conexiones y espera que se conecten 2 clientes.

```
int Puerto = 6000; // Puerto
ServerSocket Servidor = new ServerSocket(Puerto);
System.out.println("Escuchando en " + Servidor.getLocalPort() );

Socket cliente1=Servidor.accept(); //esperando a un cliente
//realizar acciones con cliente1

Socket cliente2=Servidor.accept(); //esperando otro cliente
//realizar acciones con cliente2

Servidor.close(); //Se cierra el socket del servidor.
```

➢ Clase Socket

La clase Socket implementa un extremo de la conexión TCP . Algunos de sus constructores son los siguientes:

CONSTRUCTOR	MISIÓN
Socket()	Crea un socket sin ningún puerto asociado.
Socket (InetAddress address, int port)	Crea un socket y lo conecta al puerto y dirección IP especificados.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	Permite además especificar la dirección IP local y el puerto local a los que se asociará el socket.
Socket (String host, int port)	Crea un socket y lo conecta al número de puerto y al nombre de host especificados. Puede lanzar ***UnKnownHostException***, ***IOException***

Estos constructores pueden lanzar la excepción IOException.

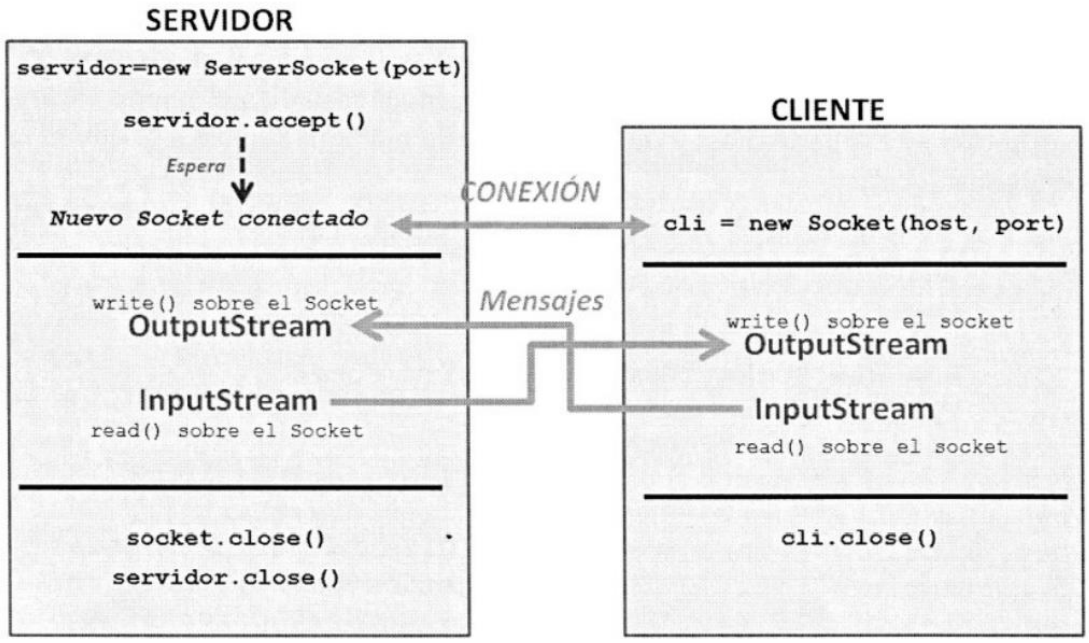
Métodos más importantes de la clase Socket

MÉTODOS	MISIÓN
InputStream getInputStream ()	Devuelve un **InputStream** que permite leer bytes desde el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar *IOException*.
OutputStream getOutputStream ()	Devuelve un **OutputStream** que permite escribir bytes sobre el socket utilizando los mecanismos de streams, el socket debe estar conectado. Puede lanzar ***IOException***.

MÉTODOS	MISIÓN
close ()	Se encarga de cerrar el socket.
InetAddress getInetAddress ()	Devuelve la dirección IP a la que el socket está conectado. Si no lo está devuelve null.
int getLocalPort ()	Devuelve el puerto local al que está enlazado el socket, -1 si no está enlazado a ningún puerto.
int getPort ()	Devuelve el puerto remoto al que está conectado el socket, 0 si no está conectado a ningún puerto.

Gestión de sockets TCP

El modelo de socket más simple se muestra en la siguiente figura:



- El programa servidor crea un socket de servidor definiendo un puerto, mediante el método ServerSocket(port), y espera mediante el método accept() a que el cliente solicite la conexión.
- Cuando el cliente solicita una conexión, el servidor abrirá la conexión al socket con el método accept().
- El cliente establece una conexión con la maquina host a través del puerto especificado mediante el método Socket(host, port). • El cliente y el servidor se comunican con

manejadores InputStream y OutputStream. El cliente escribe los mensajes en el OutputStream asociado al socket y el servidor leerá los mensajes del cliente del InputStream. Igualmente, el servidor escribirá los mensajes al OutputStream y el cliente los leerá del InputStream.

Cuando un proceso envía texto a otro a través de un socket de TCP , al igual que cuando lo hace a través de datagrama UDP , debe de utilizar la misma codificación de texto para cada operación de escritura desde el emisor y consiguiente operación de lectura desde el receptor .

Cuando se utiliza TCP, hay que utilizar stream de texto construidos sobre los streams binarios asociados al socket.

Para los socket TCP hay que:

1. Crear un OutputStreamWriter para la salida de texto a partir de un stream de salida binario OutputStream flujoSalida con new OutputStreamWriter (flujoSalida, "UTF-8"). Sobre el OutputStreamWriter se puede construir un BufferedWriter para poder escribir el fin de línea con el método newLine().

```
InputStream flujoEntrada = socket.getInputStream();
InputStreamReader flujoEntradaLectura = new InputStreamReader (flujoEntrada, "UTF-8");
BufferedReader BufferLectura = new BufferedReader(flujoEntradaLectura);
```

2. 2. Crear un InputStreamReader para entrada de texto a partir de stream de entrada binario InputStream flujoEntrada con new InputStreamReader (flujoEntrada, "UTF-8). Sobre el InputStreamReader se puede construir un BufferedReader para leer el texto línea a línea.

```
OutputStreamReader flujoSalida = socket.getOutputStream();
OutputStreamWriter flujoSalidaEscritura = new OutputStreamWriter (flujoSalida, "UTF-8");
BufferedWriter bufferEsctritura = new BufferedWriter (flujoSalidaEscritura);
```

NOTA: : Se esté escribiendo o leyendo, los tres streams que se manejan (binario, de texto , y de texto con buffering) se pueden utilizar conjuntamente. Pero eso si, el proceso receptor debe saber previamente la longitud de la secuencia de bytes.

Se suele llamar sesión al intercambio de mensaje entre cliente y servidor desde el momento en que se establece una conexión hasta el momento en que se cierra.

Para poder escribir y leer texto línea a línea , se construyen sendos BufferedWriter y BufferedReader sobre los OutputStream e ImputStream asociados al socket. Tras cada operación de escritura se ejecuta el método flush para que el texto se envíe inmediatamente.

Apertura de sockets

En el programa servidor se crea un objeto ServerSocket invocando al método ServerSocket() en el que indicamos el número de puerto por el que el servidor escucha las peticiones de conexión de los clientes (se considera el tratamiento de excepciones)

```
ServerSocket servidor=null;
try{
        servidor = new ServerSocket(numeroPuerto);
} catch (IOException io) {
        io.printStackTrace();
}
```

Se necesita también crear un objeto Socket desde el ServerSocket para aceptar las conexiones, se usa el método accept():

```
Socket clienteConectado=null;
try{
        clienteConectado=servidor.accept();

        } catch (IOException io) {
                io.printStackTrace();
        }
```

En el programa cliente es necesario crear un objeto Socket; el socket se abre de la siguiente manera:

```
Socket cliente;
try{
        cliente=new Socket ("maquina", numeroPuerto);
}catch (IOException io) {
        io.printStackTrace();
}
```

Creación de stream de entrada

En el programa servidor podemos usar DataInputStream para recuperar los mensajes que el cliente escriba en el socket, previamente hay que usar el método getInputStream() para obtener el flujo de entrada del socket del cliente:

```
InputStream entrada=null;
try{
        entrada=clienteConectado.getInputStream();
} catch (IOException e) {
        e.printStackTrace();
}
DataInputStream flujoEntrada = new DataInputStream (entrada);
```

En el programa cliente podemos realizar la misma operación para recibir los mensajes procedentes del programa servidor.

Creación de streams de salida

En el programa servidor podemos usar DataOutputStream para escribir los mensajes que queremos que el cliente reciba,, previamente hay que usar el método getOutputStream() para obtener el flujo de salida del socket del cliente:

```
OutputStream salida=null;
try{
        salida = clienteConectado.getOutputStream();
}catch (IOException e1) {
        e1.printStackTrace();
}
DataOutputStream flujoSalida=new DataOutputStream(salida);
```

NOTA: En el programa cliente tenemos que realizar la misma operación para enviar mensajes al programa servidor.

Cierre de sockets

El orden de cierre de los socket es relevante, primero se han de cerrar los streams relacionados con un socket antes que el propio socket:

```
try{
        entrada.close();
        flujoEntrada.close();
        salida.close();
        flujoSalida.close();
        clienteConectado.close();
        servidor.close()
}catch (IOException e){
        e.printStackTrace();
}
```

1.5.-Clases para sockets UDP

Los sockets UDP son más simples y eficientes que los TCP pero sin embargo, no garantiza la entrega de paquetes. No es necesario establecer una "conexión" entre cliente y servidor, como en el caso de los sockets TCP, por ello cada vez que se envíen datagramas el emisor debe indicar explícitamente la dirección IP y el puerto del destino para cada paquete y el receptor debe extraer la dirección IP y el puerto del emisor del paquete.

El paquete del datagrama está formado por los siguientes campos:

CADENA DE BYTES CONTENIENDO EL MENSAJE	LONGITUD DEL MENSAJE	DIRECCIÓN IP DESTINO	Nº DE PUERTO DESTINO

El paquete java.net proporciona las clases DatagramPacket y DatagramSocket para implementar sockets UDP.

➢ Clase DatagramPacket

Esta clase proporciona constructores para crear instancias de los datagramas que se van a recibir y de los datagramas que van a ser enviados:

CONSTRUCTOR	MISIÓN
DatagramPacket(byte[] buf, int length)	**Constructor para datagramas recibidos**. Se especifica la cadena de bytes en la que alojar el mensaje (*buf*) y la longitud (*length*) de la misma.
DatagramPacket(byte[] buf, int offset, int length)	**Constructor para datagramas recibidos**. Se especifica la cadena de bytes en la que alojar el mensaje, la longitud de la misma y el offset (*offset*) dentro de la cadena.
DatagramPacket(byte[] buf, int length, InetAddress addrss, int port)	**Constructor para el envío de datagramas**. Se especifica la cadena de bytes a enviar (*buf*), la longitud (*length*), el número de puerto de destino (*port*) y el el host especificado en la dirección *addrss*.
DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)	**Constructor para el envío de datagramas**. Igual que el anterior pero se especifica un offset dentro de la cadena de bytes.

mensaje: Enviando Saludos !!,	*Longitud: 19*	*destino: 192.168.21 IP del host local*	*port:12345*

```
Int puerto = 12345; //puerto por el que se realiza la escucha.
InetAddress destino = InetAddress.getLocalHost(); //IP del host localizar

byte[] mensaje = new byte[1024]; // matriz de bytes
String Saludo = "Enviando un saludo soy el alumno ….. !!";
mensaje = Saludo.getBytes();  // codificamos el mensaje a bytes para enviarlo

// Construimos el datagrama a enviar con el tercer constructor.
DatagramPacket envio = new DatagramPacket(mensaje, mensaje.length, destino, puerto);
```

Nota: Para definir el destino de un host con una IP concreta, por ejemplo la 192.168.10.5, escribo lo siguiente:

```
InetAddress destino = InetAddress.getByName("192.168.10.5");
```

Algunos métodos de la clase DatagramPacket

MÉTODOS	MISIÓN
InetAddress getAddress ()	Devuelve la dirección IP del host al cual se le envía el datagrama o del que el datagrama se recibió.
byte[] getData()	Devuelve el mensaje contenido en el datagrama tanto recibido como enviado.
int getLength()	Devuelve la longitud de los datos a enviar o a recibir.
int getPort()	Devuelve el número de puerto de la máquina remota a la que se le va a enviar el datagrama o del que se recibió el datagrama.
setAddress (InetAddress addr)	Establece la dirección IP de la máquina a la que se envía el datagrama.
setData (byte [buf])	Establece el búfer de datos para este paquete.
setLength (int length)	Ajusta la longitud de este paquete.
setPort (int Port)	Establece el número de puerto del host remoto al que este datagrama se envía.

Ejemplo de datagrama de recepción

```
byte[] bufer = new byte [1024];

DatagramPacket recibo = new DatagramPacket (buffer, bufer.length);

int bytesRec = recibo.getLength(); //obtenemos la longitud del mensaje.

String paquete = new String (recibo.getData() ); // obtenemos el mensaje recibido

System.out.println ("Puerto de origen del mensaje : "+ recibo.getPort() );

System.out.println ("IP de origen :"+ recibo.getAddress().getHostAddress() );
```

➢ Clase Datagram Socket

Da soporte a socket para el envío y recepción de datagramas UDP. Algunos de los constructores de esta clase , que pueden lanzar la excepción "SocketException", son:

CONSTRUCTOR	MISIÓN
DatagramSocket ()	Construye un socket para datagramas, el sistema elige un puerto de los que están libres.
DatagramSocket (int port)	Construye un socket para datagramas y lo conecta al puerto local especificado.
DatagramSocket (int port, InetAddress ip)	Permite especificar, además del puerto, la dirección local a la que se va a asociar el socket.

Ejemplo de construcción de un socket para datagrama y no lo conecta a ningún puerto, el sistema elige el puerto:

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

Para enlazar el socket a un puerto específico deberemos escribir lo siguiente:

```
DatagramSocket socket = new DatagramSocket (12345);
```

Algunos métodos importante de la clase DatagramSocket

MÉTODOS	MISIÓN
receive (DatagramPacket paquete)	Recibe un **DatagramPacket** del socket, y llena *paquete* con los datos que recibe (mensaje, longitud y origen). Puede lanzar la excepción ***IOException***.
send (DatagramPacket paquete)	Envía un **DatagramPacket** a través del socket. El argumento *paquete* contiene el mensaje y su destino. Puede lanzar la excepción ***IOException***.
close ()	Se encarga de cerrar el socket.
int getLocalPort ()	Devuelve el número de puerto en el host local al que está enlazado el socket, -1 si el socket está cerrado y 0 si no está enlazado a ningún puerto.
int getPort()	Devuelve el número de puerto al que está conectado el socket, -1 si no está conectado.
connect(InetAddress address, int port)	Conecta el socket a un puerto remoto y una dirección IP concretos, el socket solo podrá enviar y recibir mensajes desde esa dirección.
setSoTimeout(int timeout)	Permite establecer un tiempo de espera límite. Entonces el método ***receive()*** se bloquea durante el tiempo fijado. Si no se reciben datos en el tiempo fijado se lanza la excepción ***InterruptedIOException.***

Siguiendo con el ejemplo anterior, una vez construido el datagram, lo enviamos usando un DatagramSocket, en el ejemplo se enlaza al puerto 34567.

Mediante el método send() se envía el datagrama:

```
//Construimos el datagrama a enviar indicando el host de destino y su puerto
DatagramPacket envio = new DatagramPacket (mensaje, mensaje.length, destino, port);
Datagram socket = new DatagramSocket (34567);
socket.send(envio); //envio datagrama a destino y puerto
```

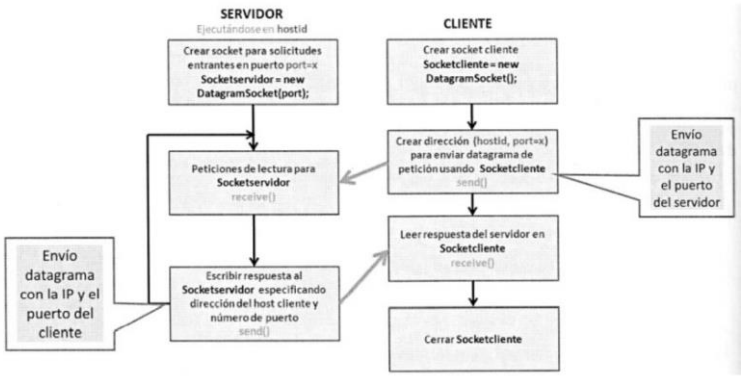
En el otro extremo, para recibir el datagrama usamos también un DatagramSocket. En primer lugar habrá que enlazar el socket al puerto por el que se va a recibir el mensaje. En este caso el puerto 12345.

Después se construye el datagrama para recepción y mediante el método receive() obtenemos los datos. Luego obtenemos la longitud , la cadena y visualizamos los puertos origen y destino del mensaje.

```
DatagramSocket socket = new DatagramSocket(12345);

//construyo datagrama a recibir

DatagramPacket recibo = new DatagramPacket(bufer, bufer.length);

System.out.println("Esperando Datagrama ..... ");


socket.receive(recibo);//recibo datagrama

int bytesRec = recibo.getLength();//obtengo numero de bytes


String paquete= new String(recibo.getData());//obtengo String

System.out.println("Número de Bytes recibidos: "+ bytesRec);

System.out.println("Contenido del Paquete: " + paquete.trim());

System.out.println("Puerto origen del mensaje: " + recibo.getPort());

System.out.println("IP de origen: " + recibo.getAddress().getHostAddress());
System.out.println("Puerto destino del mensaje: " + socket.getLocalPort());

socket.close(); //cierro el socket
```

Gestión de Sockets UDP

En los socket UDP no se establece conexión. Los roles cliente-servidor están un poco más 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 qué ordenador y en qué puerto está escuchando el otro. En la siguiente figura se muestra el flujo de la comunicación entre cliente y 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** creará un socket para comunicarse con el servidor. Para enviar datagramas necesita conocer su IP y el puerto por el que escucha. Utilizará el método send() del socket para enviar la petición en forma de datagrama.
- El **servidor** recibe las peticiones mediante el método receive() del socket. En el datagrama va incluido además del mensaje, el puerto y la IP del cliente emisor de la petición; lo que le 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 más peticiones.

Envio y recepción de datagrama

Para crear los datagramas de envío y recepción usamos la clase DatagramPacket.

Para enviar usamos el método send () de DatagramSocket pasando como parámetro el DatagramPacket que creamos previamente.

```
DatagramPacket datagrama = new DatagramPacket (mensajeEnBytes,
mensajeEnBytes.length, InetAddress.getByName("localhost"), PuertoDelServidor);
```

Donde:

mensajeEnBytes: es el array de bytes

mensajeEnBytes.length: La longitud del array

InetAddress.getByName("localhost"): La máquina de destino

PuertoDelServidor: puerto del destinatario socket.send(datagrama)

Para recibir usamos el método receive() de DatagramSocket pasando como parámetro el DatagramPacket que creamos . Este método se bloquea hasta que se recibe un datagram, a menos que se establezca un tiempo límite (timeout) sobre el socket.

```
DatagramPacket datagrama = new DatagramPacket (new byte[1024], 1024);
socket.receive(datagrama);
```

Cuando transmitimos texto, el proceso que lo envía debe utilizar siempre la misma codificación , y el que lo recibe debe de interpretarlo siempre de acuerdo a esa misma codificación (Normalmente, será UTF-8).

En las aplicaciones con el protocolo UDP, la codificación de texto interviene cuando se obtiene una secuencia de bytes a partir de una cadena de caracteres o viceversa. Para usar la codificación UTF-8 se hace de la siguiente forma:

1. Obtener la secuencia de bytes para un String cadena con cadena.getBytes("UTF-8")

2. Crear un String a partir de un byte[] array con new String (array, "UTF-8") si todos los datos de array son datos válidos. Con UDP , sin embargo, los datos de un DatagramPacket

datagrama están en " datagrama.getData()" y el número de bytes de datos válidos está en "datagrama.getLength()". Por tanto es necesario indicar el número de bytes para crear el String, y se puede crear con "new String (datagrama.getData(), 0 , datagrama.getLength(), "UTF-8") "

➢ MulticastSocket

La clase MulticastSocket es útil para enviar paquetes a múltiples destinos simultáneamente. Para poder recibir estos paquetes es necesario establecer un grupo multicast. Este grupo multicast no es más que un grupo de direcciones IP que comparten el mismo número de puerto. Cuando se envía un mensaje a un grupo multicast, todos los que estén en ese grupo recibirán el mensaje; la pertenencia al grupo es transparente al emisor, es decir, el emisor no conoce el número de miembros del grupo ni sus direcciones IP.

Un grupo multicast se especifica mediante una dirección IP de clase D y un número de puerto UDP estándar. Las direcciones desde la 224.0.0.0 a la 239.255.255.255 están destinadas para ser direcciones de multicast. La dirección 224.0.0.0 está reservada y no debe ser utilizada.

La clase MulticastSocket tiene varios constructores que pueden lanzar la excepción IOException.

CONSTRUCTOR	MISIÓN
MulticastSocket ()	Construye un socket multicast dejando al sistema que elija un puerto de los que están libres.
MulticastSocket (int port)	Construye un socket multicast y lo conecta al puerto local especificado.

Algunos métodos importantes de la clase MulticastSocket ; estos métodos pueden lanzar la excepción IOException.

MÉTODO	MISIÓN
void joinGroup(InetAddress mcastaddr)	Permite al socket multicast unirse al grupo de multicast.
void leaveGroup(InetAddress mcastaddr)	El socket multicast abandona el grupo de multicast.
void send(DatagramPacket p)	Envía el datagrama a todos los miembros del grupo multicast.
void receive(DatagramPacket p)	Recibe el datagrama de un grupo multicast

El esquema general para un servidor multicast que envía paquetes a todos los miembros del grupo es el siguiente:

1. Creamos el socket multicast; no hace falta especificar el puerto: MulticastSocket ms = new MulticastSocket ();

2. Definimos el puerto multicast int Puerto = 12345;

3. Creamos el grupo multicast InetAddress grupo = InetAddress.getByName("225.0.0.1"); String mensaje = "Bienvenidos a grupo ";

4. Creamos el datagrama DatagramPacket paquete = new DatagramPacket (mensaje.getBytes(), mensaje.length(), grupo, Puerto);

5. Enviamos el paquete al grupo mensaje.send(paquete); 6. Cerramos el socket mensaje.close();

Para que un cliente se una al grupo multicast primero crea un MulticastSocket asociado al mismo puerto que el servidor y luego invoca al método joinGroup(). El cliente multicast que recibe los paquetes que le envía el servidor tiene la siguiente estructura:

1. Se crea un socket multicast en el puerto 12345

   MulticastSocket mensaje = new MulticastSocket (12345);

2. Se configura la IP del grupo al que nos conectaremos

   InetAddress grupo = InetAddress.getByName("225.0.0.1"); 3.

3. Se une al grupo

   mensaje.joinGroup (grupo);

4. Recibe el paquete del servidor multicast

   byte[] bufer= new byte[1000];

   DatagramPacket recibido = new DatagramPacket (bufer, bufer.length);
   mensaje.receive(recibido);

5. Abandona el grupo multicast mensaje.leaveGroup (grupo);

6. Se cierra el socket mensaje.close()

UNIDAD 3: Envío de objetos a través de Sockets

Los stream soportan diversos tipos de datos como son los bytes, los tipos de datos primitivos, caracteres y objetos

1.- Objetos en Sockets TCPejemplo1Cliente

Las clases ObjectInputStream y ObjectOutputStream nos permiten enviar objetos a través de socket TCP.
Utilizaremos los siguientes métodos :

- readObject() para leer el objeto del stream.
- WriteObject () para escribir el objeto al stream.
- Usaremos el constructor que admite un InputStream y un OutputStream

- Ejemplo de preparación el flujo de salida para escribir objetos

```
ObjectOutputStream ObjetoSalida = new ObjectOutputStream(socket.getOutputStream());
```

- Ejemplo de preparación de flujo de entrada para leer objetos

```
ObjectInputStream ObjetoEntrada = new ObjectInputStream(socket.getInputStream());
```

Las clases a la que pertenecen estos objetos tienen que implementar la interfaz "Serializable".

Ejemplo de escritura de objeto al flujo de salida

1. **Creamos un objeto de la clase Lista**

```
Lista lista1 = new Lista(new int[] {12 , 15 , 11 , 4 , 32 }
```

2. **Creamos un flujo de salida a disco, pasandole el nombre del archivo en disco o un objeto de la clase**

```
FileOutputStream ficheroSalida = new FileOutputStream ("media.obj");
```

3. **El flujo de salida ObjectOutputStream es el que procesa los datos y se ha de vincular a un objeto ficheroSalida de la clase FileOutputStream**

```
ObjectOutputStream salida = new ObjectOutputStream(ficheroSalida);
```

4. **El método writeObject escribe los objetos al flujo de salida y los guarda en un archivo en disco. Por ejemplo, un string y un objeto de la clase lista**

```
salida.writeObject("guardar este string y un objeto \n");
salida.writeObject(lista1);
```

5. **Por último se cierran los flujos salida.close();**

Ejemplo de lectura de objeto del flujo de entrada

1. **Creamos un flujo de entrada a disco, pasándole el nombre del archivo en disco o un objeto de la clase File**

   FileInputStream ficheroEntrada = new FileInputStream("media.obj");

2. El flujo de entrada ObjectInputStream es el que procesa los datos y se ha de vincular a un objeto "ficheroEntrada" de la clase FileInputStream.

   ObjectInputStream entrada = new ObjectInputStream(ficheroEntrada);

3. El método readObject lee los objetos del flujo de entrada , en el mismo orden en el que ha sido escrito. Primero un string y luego, un objeto de la clase Lista.

   String str=(String) entrada.readObject();

   Lista obj1=(Lista)entrada.readObject();

4. Ahora podemos realizar tareas con dichos objetos, por ejemplo , desde el objeto obj1 de la clase Lista se llama a la función miembro "valorMedio" para obtener el valor medio del array de datos, o se muestran en la pantalla System.out.println("Valor medio "+obj1.valorMedio());

   System.out.println(str+obj1);

5. **Finalmente, se cierra los flujos**

   entrada.close();

2-. Objetos en Sockets UDP

Para intercambiar objetos en socket UDP utilizaremos las clases ByteArrrayOutputStream y ByteArrayInputStream. Se necesita convertir el objeto a un array de bytes. Por ejemplo, para un objeto "Persona" a un array de bytes escribimos las siguientes líneas:

```
Persona = new Persona ("Juan" , 58);

// CONVERTIMOS OBJETO A BYTES

ByteArrayOutputStream bs = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream (bs);

// ESCRIBIMOS EL OBJETO PERSONA EN EL STREAM

out.writeObject(persona);

// CERRAMOS EL STREAM

out.close();

// OBJETO EN BYTES

byte[] bytes = bs.toByteArray();
```

Para convertir los bytes recibidos por el datagrama en un objeto “Persona” escribimos:

```
// RECIBO DATAGRAMA

byte[] recibidos = new byte [1024];

DatagramPacket paqRecibido = new DatagramPacket (recibidos, recibidos.length);

// RECIBO EL DATAGRAMA

socket.receive(paqRecibido);

// CONVERTIMOS BYTES A OBJETO

ByteArrayInputStream bais = new ByteArrayInputStream(recibidos);

ObjectInputStream in = new ObjectInputStream(bais);

// OBTENEMOS EL OBJETO

Persona = (Persona) in.readObject();

in.close();
```

3-.Conexión de múltiples clientes: hilos

La solución para poder atender a múltiples clientes está en el “multihilo”, cada cliente será atendido en un hilo.

Clase Servidor

El esquema básico en sockets TCP sería 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”, éste se usará para crear un hilo cuya misión es atender a ese cliente. Después se vuelve a invocar a “accept()” para esperar a un nuevo cliente; habitualmente la espera de conexiones se hace dentro de un bucle infinito.

Clase HiloServidor que extiende a Thread

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

Ejemplo: Supongamos que el cliente envía una cadena de caracteres al servidor y el servidor se la devuelve en mayúsculas, hasta que recibe un asterisco que finalizará la comunicación con el cliente. El proceso de tratamiento de la cadena se realiza en un hilo, en este caso se llama “HiloServidor”.

Clase Cliente

El programa cliente “Cliente.java” se conectará con el servidor en el puerto 6000 y le enviará cadenas introducidas por teclado; cuando le envíe un asterisco el servidor finalizará la comunicación con el cliente.