Comunicaciones en red.

Caso práctico

María y **Juan** han iniciado un nuevo proyecto que le han encargado a la empresa BK, en el que le piden que realicen un programa para que muchos clientes puedan añadir información y ésta se centralice para ser consultada por todos.

María: —Hola **Juan**, nos han encargado un proyecto muy interesante en el que tenemos que realizar una aplicación utilizando sockets.

Juan: —¿Sockets? He oído hablar de ellos, pero todavía nos lo he utilizado.

María: —Los sockets permiten que las aplicaciones puedan comunicarse por red para transmitir datos. Por ejemplo, el servicio más importante de Internet, las páginas web, utiliza sockets. No te preocupes, vamos a realizar la aplicación, pero antes repasaremos los conceptos más importantes de redes.

1.- Conceptos básicos.

Caso práctico

Juan ha empezado a realizar la programación.

María: —Mira **Juan**, las aplicaciones que se comunican con Sockets lo realizan a través de una determinada dirección IP y un determinado puerto. Básicamente, las direcciones IP definen a qué ordenador van los datos y el puerto indica, dentro del ordenador, a qué programa van los datos.

Juan: —Ya me suena…. por ejemplo, la dirección IP indica el servidor web a donde van los datos y el puerto indicaría que es un servicio web, FTP,..

María: —Sí exacto. Veo que esto lo tienes más o menos claro pero vamos a profundizar un poco más. Ahora lo importante es ver los tipos de conexión que existen (TCP o UDP) porque tenemos que seleccionar un tipo.

Con la fuerte expansión que ha tenido Internet se ha generalizado la utilización de redes en las empresas y en nuestros hogares. Hoy en día para un empresa es totalmente necesario disponer de una red interna que le permita compartir información, conectarse a Internet o incluso, ofrecer sus servicios en Internet.

Cada día es más frecuente que las empresas utilicen aplicaciones que se comunican por Internet para poder compartir información entre sus empleados. Por ejemplo, aplicaciones de gestión y facturación que permiten que varias tiendas puedan realizar de forma centralizada la facturación de toda la empresa, gestionar el stock, etc.

A continuación vamos a ver los conceptos más importantes sobre redes, necesarios para más adelante poder programar nuestras aplicaciones. Para ello, primero veremos una pequeña introducción sobre el modelo TCP/IP, los tipos de conexiones que se pueden realizar, así como los modelos más importantes de comunicaciones.

1.1.- Recordando TCP/IP

En 1969 la agencia ARPA (Advanced Research Projects Agency) del Departamento de Defensa de los Estados Unidos inició un proyecto de interconexión de ordenadores mediante redes telefónicas. Al ser un proyecto desarrollado por militares en plena guerra fría un principio básico de diseño era que la red debía poder resistir la destrucción de parte de su infraestructura (por ejemplo a causa de un ataque nuclear), de forma que dos nodos cualesquiera pudieran seguir comunicados siempre que hubiera alguna ruta que los uniera. Esto se consiguió en 1972 creando una red de conmutación de paquetes denominada ARPAnet, la primera de este tipo que operó en el mundo. La conmutación de paquetes unida al uso de topologías malladas mediante múltiples líneas punto a punto dió como resultado una red altamente fiable y robusta.

ARPAnet fue creciendo paulatinamente, y pronto se hicieron experimentos utilizando otros medios de transmisión de datos, en particular enlaces por radio y vía satélite; los protocolos existentes tuvieron problemas para interoperar con estas redes, por lo que se diseñó un nuevo conjunto o pila de protocolos, y con ellos una arquitectura. Este nuevo conjunto se denominó TCP/IP (Transmission Control Protocol/Internet Protocol), nombre que provenía de los dos protocolos más importantes que componían la pila; la nueva arquitectura se llamó sencillamente modelo TCP/IP. A la nueva red, que se creó como consecuencia de la fusión de ARPAnet con las redes basadas en otras tecnologías de transmisión, se la denominó Internet.

La aproximación adoptada por los diseñadores del TCP/IP fue mucho más pragmática que la de los autores del modelo OSI. Mientras que en el caso de OSI se emplearon varios años en definir con sumo cuidado una arquitectura de capas donde la función y servicios de cada una estaban perfectamente definidas, y sólo después se planteó desarrollar los protocolos para cada una de ellas, en el caso de TCP/IP la operación fue a la inversa; primero se especificaron los protocolos, y luego se definió el modelo como una simple descripción de los protocolos ya existentes. Por este motivo el modelo TCP/IP es mucho más simple que el OSI. También por este motivo el modelo OSI se utiliza a menudo para describir otras arquitecturas, como por ejemplo TCP/IP, mientras que el modelo TCP/IP nunca suele emplearse para describir otras arquitecturas que no sean la suya propia.

1.1.1.- Recordando TCP/IP (II).

El modelo TCP/IP tiene sólo cuatro capas:

- **La capa host-red**. Esta capa permite comunicar el ordenador con el medio que conecta el equipo a la red. Para ello primero debe permitir convertir la información en impulsos físicos (p.ej. eléctricos, magnéticos, luminosos) y además, debe permitir las conexiones entre los ordenadores de la red. En esta capa se realiza un direccionamiento físico utilizando las direcciones MAC.
- **La capa de red**. Esta capa es el eje de la arquitectura TCP/IP ya que permite que los equipos envíen paquetes en cualquier red y viajen de forma independiente a su destino (que podría estar en una red diferente). Los paquetes pueden llegar incluso en un orden diferente a aquel en que se enviaron, en cuyo caso corresponde a las capas superiores reordenándolos, si se desea la entrega ordenada. La capa de red define un formato de paquete y protocolo oficial llamado IP (Internet Protocol). El trabajo de la capa de red es entregar paquetes IP a su destino. Aquí la consideración más importante es decidir el camino que tienen que seguir los paquetes (encaminamiento), y también evitar la congestión. En esta capa se realiza el direccionamiento lógico o direccionamiento por IP, ya que ésta es la capa encargada de enviar un determinado mensaje a su dirección IP de destino.
- **La capa de transporte**. La capa de transporte permite que los equipos lleven a cabo una conversación. Aquí se definieron dos protocolos de transporte: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). El protocolo TCP es un protocolo orientado a conexión y fiable, y el protocolo UDP es un protocolo no orientado a conexión y no fiable En esta capa además se realiza el direccionamiento por puertos. Gracias a la capa anterior, los paquetes viajan de un equipo origen a un equipo destino. La capa de transporte se encarga de que la información se envíe a la aplicación adecuada (mediante un determinado puerto).
- **La capa de aplicación**. Esta capa engloba las funcionalidades de las capas de sesión, presentación y aplicación del modelo OSI. Incluye todos los protocolos de alto nivel relacionados con las aplicaciones que se utilizan en Internet (por ejemplo HTTP, FTP, TELNET).

Modelo TCP/IP.

Debes conocer

Si quieres ampliar tus conocimientos sobre el modelo TCP/IP debes leer el siguiente artículo prestando especial atención a la capa de transporte.

Texto del enlace: Modelo TCP/IP.

Autoevaluación

Indica la capa que se encarga de enrutar un paquete por la red para que llegue a su destino.

- [ ] Capa host-red.
- [ ] Capa de red.
- [ ] Capa de transporte.
- [ ] Capa de aplicación.

Mostrar Información

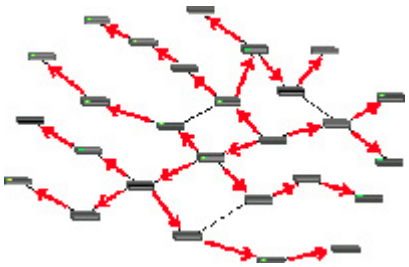
Para saber más

Si quieres ampliar tus conocimientos sobre redes puedes ver la siguiente presentación.

Redes de comunicación. (0.62 MB)

Resumen textual alternativo

1.2.- Conexiones TCP/UDP.



Como se ha visto anteriormente, la capa de transporte cumple la función de establecer las reglas necesarias para establecer una conexión entre dos dispositivos.

Desde la capa anterior, la capa de red, la información se recibe en forma de paquetes desordenados y la capa de transporte debe ser capaz de manejar dichos paquetes y obtener un único flujo de datos. Recuerde, que la capa de red en la arquitectura TCP/IP no se preocupa del orden de los paquetes ni de los errores, es en esta capa donde se deben cuidar estos detalles.

La capa de transporte del modelo TCP/IP es equivalente a la capa de transporte del modelo OSI, por lo que es el encargado de la transferencia libre de errores de los datos entre el emisor y el receptor, aunque no estén directamente conectados, así como de mantener el flujo de la red. La tarea de este nivel es proporcionar un transporte de datos confiable de la máquina de origen a la máquina destino, independientemente de la red física.

Existen dos tipos de conexiones:

- **TCP (Transmission Control Protocol)**. Es un protocolo orientado a la conexión que permite que un flujo de bytes originado en una máquina se entregue sin errores en cualquier máquina destino. Este protocolo fragmenta el flujo entrante de bytes en mensajes y pasa cada uno a la capa de red. En el diseño, el proceso TCP receptor reensambla los mensajes recibidos para formar el flujo de salida. TCP también se encarga del control de flujo para asegurar que un emisor rápido no pueda saturar a un receptor lento con más mensajes de los que pueda gestionar.
- **UDP (User Datagram Protocol)**. Es un protocolo sin conexión, para aplicaciones que no necesitan la asignación de secuencia ni el control de flujo TCP y que desean utilizar los suyos propios. Este protocolo también se utilizan para las consultas de petición y respuesta del tipo cliente-servidor, y en aplicaciones en las que la velocidad es más importante que la entrega precisa, como las transmisiones de voz o de vídeo. Uno de sus usos es en la transmisión de audio y vídeo en tiempo real, donde no es posible realizar retransmisiones por los estrictos requisitos de retardo que se tiene en estos casos.

De esta forma, a la hora de programar nuestra aplicación deberemos elegir el protocolo que queremos utilizar según nuestras necesidades: TCP o UDP.

Para saber más

Es interesante que leas algo más sobre los protocolos más importantes de este nivel, por lo que te proponemos los siguientes enlaces.

TCP.

UDP.

Autoevaluación

Las conexiones _____ garantizan la correcta recepción de los paquetes y las conexiones _____ son más rápidas pero no permiten la corrección de errores.

Enviar

1.3.- Puertos de comunicación

Con la capa de red se consigue que la información vaya de un equipo origen a un equipo destino a través de su dirección IP. Pero para que una aplicación pueda comunicarse con otra aplicación es necesario establecer a qué aplicación se conectará. El método que se emplea es el de definir direcciones de transporte en las que los procesos pueden estar a la escucha de solicitudes de conexión. Estos puntos terminales se llaman puertos.

Aunque muchos de los puertos se asignan de manera arbitraria, ciertos puertos se asignan, por convenio, a ciertas aplicaciones particulares o servicios de carácter universal. De hecho, la IANA (Internet Assigned Numbers Authority) determina, las asignaciones de todos los puertos. Existen tres rangos de puertos establecidos:

- **Puertos conocidos [0, 1023]**. Son puertos reservados a aplicaciones de uso estándar como: 21 – FTP (File Transfer Protocol), 22 – SSH (Secure SHell), 53 – DNS (Servicio de nombres de dominio), 80 – HTTP (Hypertext Transfer Protocol), etc.
- **Puertos registrados [1024, 49151]**. Estos puertos son asignados por IANA para un servicio específico o aplicaciones. Estos puertos pueden ser utilizados por los usuarios libremente.
- **Puertos dinámicos [49152, 65535]**. Este rango de puertos no puede ser registrado y su uso se establece para conexiones temporales entre aplicaciones.

Cuando se desarrolla una aplicación que utilice un puerto de comunicación, optaremos por utilizar puertos comprendidos entre el rango 1024-49151.

Para saber más

Durante el desarrollo de este módulo y de otros de este ciclo, necesitaras conocer cuales son los puertos relacionados con cada una de las aplicaciones, por tanto te recomendamos el siguiente enlace.

Puertos.

Autoevaluación

¿Qué puerto se utiliza para el servicio web?

- [ ] 21
- [ ] 80

8080

22

Mostrar Información

1.4.- Nombres en Internet.

Los equipos informáticos se comunican entre sí mediante una dirección IP como 193.147.0.29. Sin embargo nosotros preferimos utilizar nombres como www.mec.es porque son más fáciles de recordar y porque ofrecen la flexibilidad de poder cambiar la máquina en la que están alojados (cambiaría entonces la dirección IP) sin necesidad de cambiar las referencias a él.

El sistema de resolución de nombres (DNS) basado en dominios, en el que se dispone de uno o más servidores encargados de resolver los nombres de los equipos pertenecientes a su ámbito, consiguiendo, por un lado, la centralización necesaria para la correcta sincronización de los equipos, un sistema jerárquico que permite una administración focalizada y, también, descentralizada y un mecanismo de resolución eficiente.

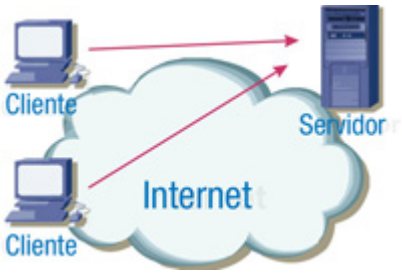
A la hora de comunicarse con un equipo, puedes hacerlo directamente a través de su dirección IP o puede poner su entrada DNS (p.ej. servidor.miempresa.com). En el caso de utilizar la entrada DNS el equipo resuelve automáticamente su dirección IP a través del servidor de nombres que utilice en su conexión a Internet.

Empresa de registro de dominios www.arsys.es

1.5.- Modelos de comunicaciones.

Sin duda alguna, el modelo de comunicación que ha revolucionado los sistemas informáticos es el modelo cliente/servidor. El modelo cliente/servidor esta compuesto por un servidor que ofrece una serie de servicios y unos clientes que acceden a dichos servicios a través de la red.

Por ejemplo, el servicio más importante de Internet, el WWW , utiliza el modelo cliente/servidor. Por un lado tenemos el servidor que alojan las páginas web y por otro lado los clientes que solicitan al servidor una determinada página web.



Modelo Cliente/Servidor

Otro modelo ampliamente utilizado son los Sistemas de Información Distribuidos. Un Sistema de Información Distribuido esta compuesto por un conjunto de equipos que interactúan entre sí y pueden trabajar a la vez como cliente y servidor. Desde el punto de vista externo es igual que un sistema cliente/servidor ya que el cliente ve al Sistema de Información Distribuido como una entidad. Internamente, los equipos del Sistema de Información Distribuido interactúan entre sí (actuando como servidores y clientes de forma simultánea) para compartir información, recursos, realizar tareas, etc.

Modelo Sistema de Información Distribuido

Autoevaluación

Rellena los huecos con las palabras que faltan.

El [ ] [ ] suele tener un servidor mientras que [ ] [ ] [ ] [ ] [ ] esta compuesto por muchos servidores.

Enviar

2.- Sockets TCP.

Caso práctico

Juan va a hablar con **María** para debatir lo aprendido.

Juan: —Hola **María**, ya lo he entendido. Tenemos dos tipos de conexiones: TCP que son orientadas a conexión y se utilizan para enviar datos de aplicaciones, y UDP que son no orientadas a conexión y se utilizan cuando queremos enviar datos muy rápido pero no nos importa que se pierda algún paquete.

María: —Exacto Juan. Para ponerte un ejemplo, las conexiones TCP se utilizan para enviar ficheros (p.ej. el servicio FTP) y las comunicaciones UDP se utilizan para enviar música y vídeo en tiempo real. Como tenemos claro que la aplicación que vamos a realizar va a utilizar conexiones TCP, ya que no queremos que se pierda ningún dato, a continuación vamos a aprender a programar con java los sockets TCP.

Los sockets permiten la comunicación entre procesos de diferentes equipos de una red. Un socket, es un punto de información por el cual un proceso puede recibir o enviar información.

Tal y como hemos visto anteriormente, existen dos tipos de protocolos de comunicación: TCP o UDP. En el protocolo TCP es un protocolo orientado a la conexión que permite que un flujo de bytes originado en una máquina se entregue sin errores en cualquier máquina destino Por otro lado, el protocolo UDP, no orientado a conexión, se utiliza para comunicaciones donde se prioriza la velocidad sobre la pérdida de paquetes.

A la hora de crear un socket hay que tener claro el tipo de socket que se quiere crear (TCP o UDP). En esta sección vamos a aprender a utilizar los sockets TCP y en el siguiente punto veremos los sockets UDP.

Al utilizar sockets TCP, el servidor utiliza un puerto por el que recibe las diferentes peticiones de los clientes. Normalmente, el puerto del servidor es un puerto bajo [1-1023].

Socket TCP - Conexión

Cuando el cliente realiza la conexión con el servidor, a partir de ese momento se crea un nuevo socket que será en el encargado de permitir el envío y recepción de datos entre el cliente/servidor. El puerto se crea de forma dinámica y se encuentra en el rango 49152 y 655335. De ésta forma, el puerto por donde se reciben las conexiones de los clientes queda libre y cada comunicación tiene su propio socket.

TCP: transferencia de datos

El paquete **java.net** de Java proporciona la clase **Socket** que permite la comunicación por red. De forma adicional, **java.net** incluye la clase **ServerSocket**, que permite a un servidor escuchar y recibir las peticiones de los clientes por la red. Y la clase **Socket** permite a un cliente conectarse a un servidor para enviar y recibir información.

Para saber más

A continuación se estudian las llamadas más importantes de Socket en java. Si deseas complementar dicha información, en el siguiente enlace puedes ver la documentación oficial de Sockets en java.

Sockets en Java.

Autoevaluación

Indica si la siguiente información es correcta. Cuando te conectas al puerto 80 TCP, ¿todas las comunicaciones van por ese puerto?

Verdadero. Falso.

2.1.- Servidor.

Los pasos que realiza el servidor para realizar una comunicación son:

- **Publicar puerto**. Se utiliza el comando `ServerSocket` indicando el puerto por donde se van a recibir las conexiones.
- **Esperar peticiones**. En este momento el servidor queda a la espera a que se conecte un cliente. Una vez que se conecte un cliente se crea el socket del cliente por donde se envían y reciben los datos.
- **Envío y recepción de datos**. Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida. Cuando el servidor recibe una petición, éste la procesa y le envía el resultado al cliente.
- Una vez finalizada la comunicación se **cierra el socket del cliente**.

Esquema de funcionamiento interno del modelo cliente/servidor

Para publicar el puerto del servidor se utiliza la función **`ServerSocket`** a la que hay que indicarle el puerto a utilizar. Su estructura es:

```
ServerSocket skServidor = new ServerSocket(Puerto);
```

Una vez publicado el puerto, el servidor utiliza la función **`accept()`** para esperar la conexión de un cliente. Una vez que el cliente se conecta, entonces se crea un nuevo socket por donde se van a realizar todas las comunicaciones con el cliente y servidor.

```
Socket sCliente = skServidor.accept();
```

Una vez recibida la petición del cliente el servidor se comunica con el cliente a través de streams de datos que veremos en el siguiente punto.

Finalmente, una vez terminada la comunicación se cierra el socket de la siguiente forma:

```
sCliente.close();
```

A continuación se muestra el código comentado de un servidor

```
import java.io.* ;
import java.net.* ;
class Servidor {
    static final int Puerto=2000;
    public Servidor( ) {
    try {
        // Inicio la escucha del servidor en un determinado puerto
        ServerSocket skServidor = new ServerSocket(Puerto);
        System.out.println("Escucho el puerto " + Puerto );
        // Espero a que se conecte un cliente y creo un nuevo socket para e
        Socket sCliente = skServidor.accept();
```

```
            // ATENDER PETICIÓN DEL CLIENTE
            // Cierro el socket
            sCliente.close();
            }
        } catch( Exception e ) {
            System.out.println( e.getMessage() );
    }
    public static void main( String[] arg ) {
        }
         new Servidor();
     }
    }
```

2.2.- Cliente.

Los pasos que realiza el cliente para realizar una comunicación son:

- **Conectarse con el servidor**. El cliente utiliza la función Socket para indicarse con un determinado servidor a un puerto específico. Una vez realizada la conexión se crea el socket por donde se realizará la comunicación.
- **Envío y recepción de datos**. Para poder recibir/enviar datos es necesario crear un flujo (stream) de entrada y otro de salida.
- Una vez finalizada la comunicación se **cierra el socket**.

Esquema de funcionamiento interno del modelo cliente/servidor.

Para conectarse a un servidor se utiliza la función **Socket** indicando el equipo y el puerto al que desea conectarse. Su sintaxis es:

```
Socket sCliente = new Socket( Host , Puerto );
```

donde Host es un string que guarda el nombre o dirección IP del servidor y Puerto es una variable del tipo **int** que guarda el puerto. "

Si lo prefiere también puede realizar la conexión directamente:

```
Socket sCliente = new Socket("192.168.1.200", 1500);
```

Una vez establecida la comunicación, se crean los streams de entrada y salida para realizar las diferentes comunicaciones entre el cliente y el servidor. En el siguiente apartado veremos la creación de streams.

Finalmente, una vez terminada la comunicación se cierra el socket de la siguiente forma:

```
sCliente.close();
```

A continuación se muestra el código comentado de un servidor.

```
import java.io.*;
import java.net.*;
class Cliente {
    static final String Host = "localhost";
    static final int Puerto=2000;
    public Cliente( ) {
        try{
            // Me conecto al servidor en un detrminado puerto
            Socket sCliente = new Socket( Host, Puerto );
// TAREAS QUE REALIZA EL  CLIENTE
           // Cierro el socket
            sCliente.close();
        } catch( Exception e ) {
```

```
                System.out.println( e.getMessage() );
            }
        }
        public static void main( String[] arg ) {
            new Cliente();
        }
    }
```

Autoevaluación

¿Qué función permite al servidor publicar un puerto?

- [ ] Socket.
- [ ] ServerSocket.
- [ ] Accept.
- [ ] ExportSocket.

Mostrar Información

2.3.- Flujo de Entrada y de Salida.

Una vez establecida la conexión entre el cliente y el servidor se inicializa la variable del tipo Socket que en el ejemplo se llama sCliente. Para poder enviar o recibir datos a través del socket es necesario establecer un stream (flujo) de entrada o de salida según corresponda.

Continuando con el ejemplo anterior, a continuación se va a establecer un stream de salida llamado flujo_salida.

```
OutputStream aux = sCliente.getOutputStream();

DataOutputStream flujo_salida= new DataOutputStream( aux );
```

o lo que es lo mismo,

```
DataOutputStream flujo_salida= new

DataOutputStream(sCliente.getOutputStream());
```

A partir de éste momento puede enviar información de la siguiente forma:

```
flujo_salida.writeUTF( "Enviar datos");
```

De forma análoga, puede establecer el stream de entrada de la siguiente forma:

```
InputStream aux = sCliente.getInputStream();

DataInputStream flujo_entrada = new DataInputStream( aux );
```

o lo que es lo mismo,

```
DataInputStream flujo_entrada = new

DataInputStream(sCliente.getInputStream());
```

A continuación se muestra una forma cómoda de recibir información:

```
String datos=new String();

Datos=flujo_entrada.readUTF();
```

Para saber más

Además de utilizar las funciones **writeUTF** y **readUTF** es posible recibir información de otras formas. Para más información consulte los siguientes enlaces.

Data Input Stream.

Data Output Stream.

Autoevaluación

Indica si la siguiente afirmación es correcta. ¿Existen flujos de entrada y salida simultáneos?

- [ ] Verdadero.
- [ ] Falso.

Mostrar Información

2.4.- Ejemplo.

Para continuar con el ejemplo anterior y poder utilizar los sockets para enviar información vamos a realizar un ejemplo muy sencillo en el que el servidor va a aceptar tres clientes (de forma secuencial no concurrente) y le va a indicar el número de cliente que es.

Servidor.java

```
import java.io.* ;
import java.net.* ;
class Servidor {
    static final int Puerto=2000;
    public Servidor( ) {
    try {
        ServerSocket skServidor = new ServerSocket(Puerto);
        System.out.println("Escucho el puerto " + Puerto );
        for ( int nCli = 0; nCli < 3; nCli++) {
            Socket sCliente = skServidor.accept();
     System.out.println("Sirvo al cliente " + nCli);
     OutputStream aux = sCliente.getOutputStream();
     DataOutputStream flujo_salida= new DataOutputStream( aux );
     flujo_salida.writeUTF( "Hola cliente " + nCli );
     sCliente.close();
        }
        System.out.println("Ya se han atendido los 3 clientes");
    } catch( Exception e ) {
        System.out.println( e.getMessage() );
    }
}
public static void main( String[] arg ) {
     new Servidor();
}
}
```

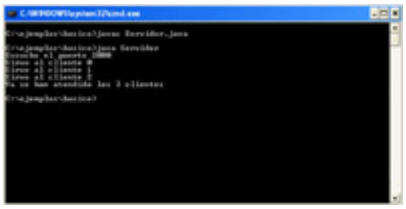
Para compilar el programa ejecuta:

`javac Servidor.java`

y para ejecutarlo ejecuta

`java Servidor`

En la siguiente figura se muestra al servidor como ha procesado las solicitudes de tres clientes.



java Servidor.

2.4.1.- Ejemplo (II)

De la misma forma, creamos el cliente con el siguiente código:

Cliente.java

```
import java.io.*;
import java.net.*;
class Cliente {
    static final String HOST = "localhost";
    static final int Puerto=2000;
    public Cliente( ) {
        try{
              Socket sCliente = new Socket( HOST , Puerto );
     InputStream aux = sCliente.getInputStream();
     DataInputStream flujo_entrada = new DataInputStream( aux );
     System.out.println( flujo_entrada.readUTF() );
     sCliente.close();
        } catch( Exception e ) {
     System.out.println( e.getMessage() );
        }
    }
    public static void main( String[] arg ) {
     new Cliente();
    }
}
```

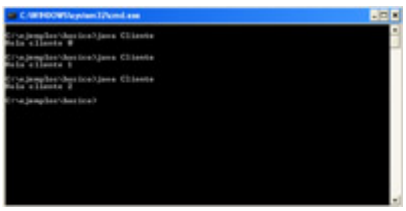
Para compilar el programa ejecuta:

`javac Cliente.java`

y para ejecutarlo ejecuta

`java Cliente`

Como puede ver en la siguiente figura, al ejecutar el cliente éste se conecta al servidor y muestra el mensaje que le envía el servidor.



java Cliente.

3.- Sockets UDP.

Caso práctico

Juan esta terminado de programar la aplicación.

María: —Hola **Juan**. Ya he recibido tu aplicación y va bastante bien. Para que puedas aprender más ahora vamos a ver la programación de sockets UDP. Como ya sabes, los sockets UDP se utilizan para cuando tenemos que enviar datos muy rápido y no nos preocupa que se pierda algún paquete.

Juan: —Estupendo María. Estoy deseando de empezar.

En el caso de utilizar sockets UDP no se crea una conexión (como es el caso de socket TCP) y básicamente permite enviar y recibir mensajes a través de una dirección IP y un puerto. Estos mensajes se gestionan de forma individual y no se garantiza la recepción o envío del mensaje como si ocurre en TCP.

Para utilizar sockets UDP en java tenemos la clase **`DatagramSocket`** y para recibir o enviar los mensajes se utiliza clase **`DatagramPacket`**. Cuando se recibe o envía un paquete se hace con la siguiente información: mensaje, longitud del mensaje, equipo y puerto.

Para saber más

En los siguientes enlaces puedes ver los manuales oficiales de **`DatagramSocket`** y **`DatagramPacket`**.

DatagramSocket.

DatagramPacket.

3.1.- Receptor.

En el caso de querer iniciar el socket en un determinado puerto se realiza de la siguiente forma:

```
DatagramSocket sSocket = new DatagramSocket(puerto);
```

Una vez iniciado el socket ya estamos en disposición de recibir mensajes utilizando la clase **DatagramPacket**. Cuando se recibe o envía un paquete se hace con la siguiente información: mensaje, longitud del mensaje, equipo y puerto.

A continuación se muestra un código de ejemplo para recibir un mensaje:

```
byte [] cadena = new byte[1000] ;

DatagramPacket mensaje = new DatagramPacket(cadena, cadena.length);

sSocket.receive(mensaje);
```

Una vez recibido el mensaje puede mostrar su contenido de la siguiente forma:

```
String datos=new String(mensaje.getData(),0,mensaje.getLength());

System.out.println("\tMensaje Recibido: " +datos);
```

Finalmente, una vez terminado el programa cerramos el socket:

```
sSocket.close();
```

3.2.- Emisor.

Por otro lado, para realizar una aplicación emisora de mensajes UDP debe inicializar primero la estructura **DatagramSocket**.

```
DatagramSocket sSocket = new DatagramSocket();
```

Ahora debe crear el mensaje del tipo **DatagramPacket** al que debe indicar:

- Mensaje a enviar.
- Longitud del mensaje.
- Equipo al que se le envía el mensaje.
- Puerto destino.

A continuación se muestra un ejemplo para crear un mensaje:

```
DatagramPacket mensaje = new DatagramPacket(mensaje,longitud_mensaje, Equipo, Puerto);
```

Para obtener la dirección del equipo al que se le envía el mensaje a través de su nombre se utiliza la función getByName de la clase **InetAddress** de la siguiente forma

```
InetAddress Equipo = InetAddress.getByName("localhost");
```

Una vez creado el mensaje lo enviamos con la función **send():**

```
sSocket.send(mensaje);
```

Finalmente, una vez terminado el programa cerramos el socket:

```
sSocket.close();
```

Autoevaluación

Indica la característica que no pertenece a los sockets UDP.

- [ ] Utiliza un determinado puerto.
- [ ] Establece una conexión entre un cliente/servidor.
- [ ] Permiten enviar y recibir paquetes.
- [ ] En cada paquete va la dirección y puerto destino.

Mostrar Información

3.3.- Ejemplo.

A continuación, para aprender a programar Sockets UDP se va a realizar un ejemplo sencillo donde intervienen dos procesos:

- **ReceptorUDP**. Inicia el puerto 1500 y muestra en pantalla todos los mensajes que llegan a él.
- **EmisorUDP**. Permite enviar por líneas de comandos mensajes al receptor por el puerto 1500.

ReceptorUDP.java

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

public class ReceptorUDP {
  public static void main(String args [] ) {
    // Sin argumentos
    if (args.length != 0) {
      System.err.println("Uso: java ReceptorUDP");
    }
    else try{
      // Crea el  socket
      DatagramSocket sSocket = new DatagramSocket(1500);

      // Crea el espacio para los mensajes
      byte [] cadena = new byte[1000] ;
      DatagramPacket mensaje = new DatagramPacket(cadena, cadena.length);

      System.out.println("Esperando mensajes..");
      while(true){
           // Recibe y muestra el mensaje
           sSocket.receive(mensaje);
           String datos=new String(mensaje.getData(),0,mensaje.getLength())
           System.out.println("\tMensaje Recibido: " +datos);
      }
    } catch(SocketException e) {
      System.err.println("Socket: " + e.getMessage());
    } catch(IOException e) {
      System.err.println("E/S: " + e.getMessage()); }
  }
}
```

3.3.1.- Ejemplo (II).

EmisorUDP.java

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

public class EmisorUDP {
  public static void main(String args [] ) {
    // Comprueba los argumentos
    if (args.length != 2) {
      System.err.println("Uso: java EmisorUDP maquina mensaje");
    }
    else try{
      // Crea el socket
      DatagramSocket sSocket = new DatagramSocket();

      // Construye la dirección del socket del receptor
      InetAddress maquina = InetAddress.getByName(args[0]);
      int Puerto = 1500;

      // Crea el mensaje
      byte [] cadena = args[1].getBytes();
      DatagramPacket mensaje = new DatagramPacket(cadena,args[1].length(),

      // Envía el mensaje
      sSocket.send(mensaje);

      // Cierra el socket
      sSocket.close();
    } catch(UnknownHostException e) {
      System.err.println("Desconocido: " + e.getMessage());
    } catch(SocketException e) {
      System.err.println("Socket: " + e.getMessage());
    } catch(IOException e) {
      System.err.println("E/S: " + e.getMessage());
    }
  }
}
```

3.3.2.- Ejemplo (III).

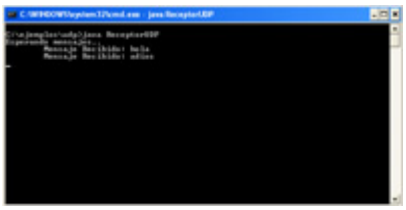
Para realizar la prueba compilamos el código ejecutando:

`javac ReceptorUDP.java`

`javac EmisorUDP.java`

En un terminal lanzamos el **`ReceptorUDP`** ejecutando:

`java ReceptorUDP`



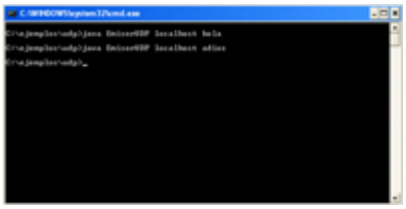
java ReceptorUDP.

Y en el otro terminal del sistema lanzamos el **`EmisorUDP`** varias veces de la siguiente forma

`java EmisorUDP <equipo> <mensaje>`

donde equipo es el nombre del equipo o dirección IP del equipo al que se le van a enviar los mensajes. Por ejemplo, a continuación, vamos a mandar un mensaje de prueba:

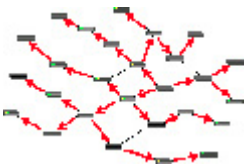
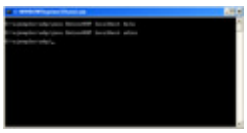
`java EmisorUDP localhost hola`



java EmisorUDP.

Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de T

Recurso (1)	Datos del recurso (1)	Recurso (2)	
	Autoría: ryanlerch. Licencia: Dominio público. Procedencia: http://www.openclipart.org/detail/artwork---paintbrush-scissors-and-glue-by-ryanlerch		Autoría: boi Licencia: D Procedenci /boirac_Loc
	Autoría: baccala@freesoft.org. Licencia: Dominio público. Procedencia: http://commons.wikimedia.org/wiki/File:Broadcast_forwarding.png		Autoría: ww Licencia: D Procedenci 56456.
	Autoría: arsys. Licencia: Copyright (cita). Procedencia: www.arsys.es.		Autoría: Jul Licencia: U Procedenci ejecución d
	Autoría: Julio Gómez López. Licencia: Uso Educativo no comercial. Procedencia: Terminal del sistema mostrando la ejecución del programa cliente.		Autoría: Jul Licencia: U Procedenci ejecución d
	Autoría: Julio Gómez López. Licencia: Uso Educativo no comercial. Procedencia: Terminal del sistema mostrando la ejecución del programa emisor.		