SD	Sistemas Distribuidos
12/13	Prácticas
#1	Introducción a la tecnología de Sockets Java

Introducción

Es un mecanismo de comunicación punto a punto entre dos procesos sobre el protocolo TCP/IP. Desde el punto de vista de programación, un socket no es más que un "fichero" que se abre de una manera especial. Una vez abierto podemos leer y escribir con las funciones específicas para tal fin. Existen dos tipos de comunicación: la orientada a la conexión y la no orientada a la conexión. En el primer caso se debe establecer la conexión entre ambos procesos mediante un socket para poder transmitir. En este caso se utiliza el protocolo TCP, el cual es un protocolo confiable que asegura que los datos transmitidos llegan de un proceso a otro correctamente. En el segundo caso se utiliza el protocolo UDP, que garantiza que los datos que llegan son correctos pero no que lleguen todos. En este caso un proceso envía datos sin la necesidad de que el otro se encuentre escuchando. En nuestro caso uno de los procesos será el Servidor y se encontrará en estado de escucha a la espera de recibir una conexión del proceso Cliente. Para realizar una conexión entre un cliente y un servidor mediante sockets debemos conocer la dirección IP de la máquina servidor y el puerto que identifica el servicio al cual nos queremos conectar (desde el 1 al 1023 están reservados para servicios conocidos quedando libres hasta el 65535).

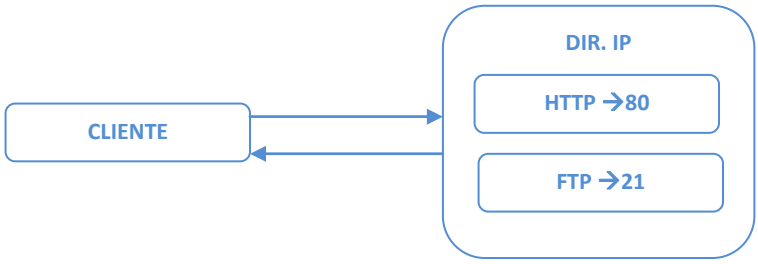

Figura 1. Ejemplo de protocolos utilizando sockets

En Java la tecnología de sockets es proporcionada a través del paquete *java.net*.

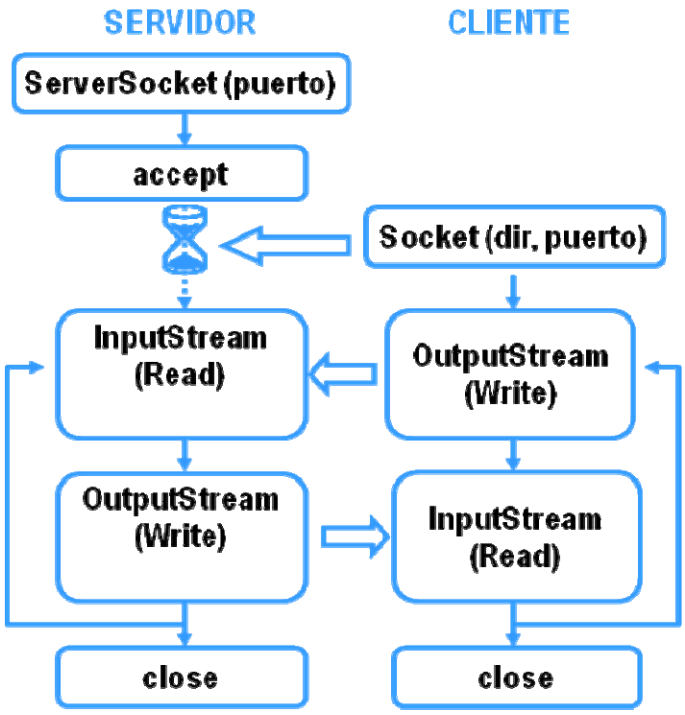

Figura 2. Esquema de comunicación simple mediante sockets.

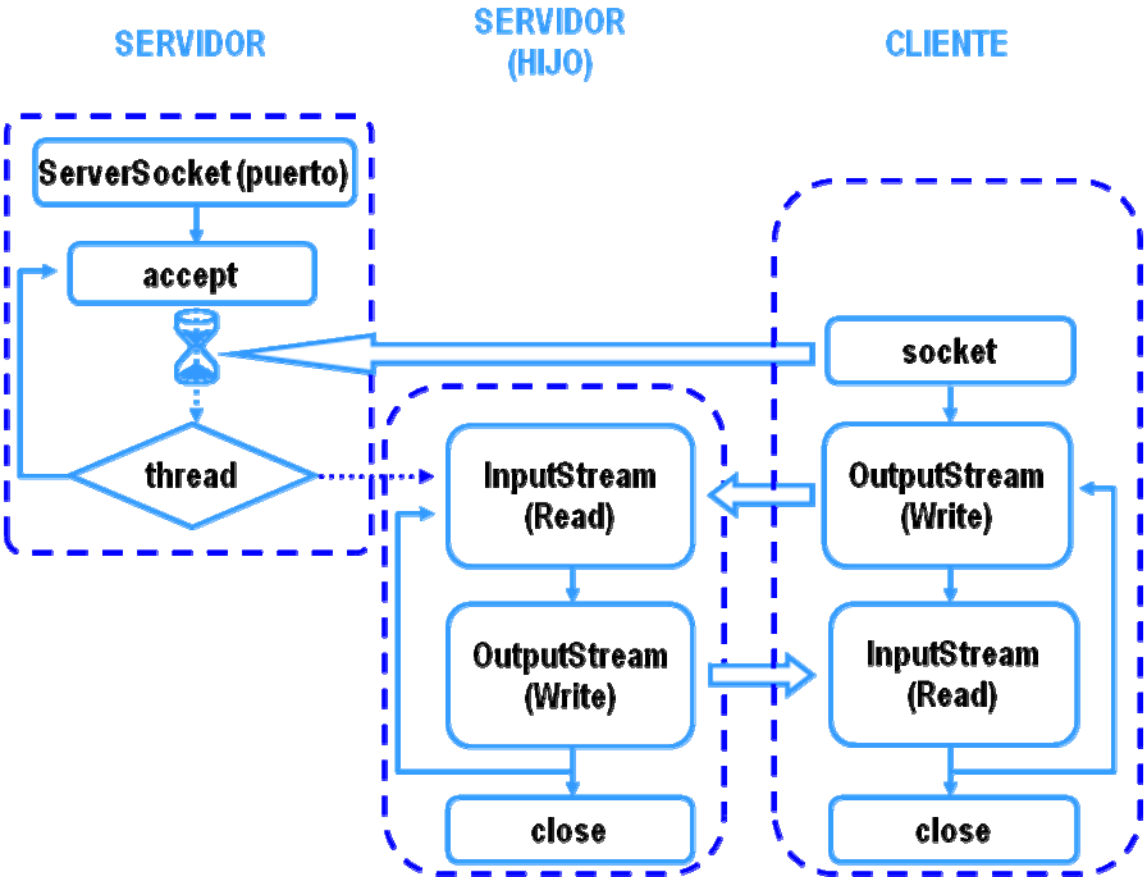

Figura 3. Esquema de comunicación concurrente mediante sockets.

Los pasos a seguir para crear una aplicación cliente/servidor con sockets son:

1. Creación del servidor
2. Creación del cliente
3. Compilación y ejecución En la práctica utilizaremos sockets en C en Linux.

Creación del servidor

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

La clase *ServerSocket* permite abrir una conexión para que el servidor pueda escuchar peticiones a través de un puerto a la espera de una conexión TCP. En Java todas las conexiones que se establecen a través de un socket son de tipo Internet. En el caso de que quisiéramos establecer una comunicación de tipo UDP se deberían utilizar otra clase de la librería como es *DatagramSocket*.

```
for(;;)
{
        Socket skCliente = skServidor.accept();
```

En Java el método *accept* es la que permite aceptar peticiones procedentes de los clientes.

Posteriormente, realizaríamos tantas lecturas y escrituras como el protocolo necesitase. En el ejemplo se crea un bucle que termina en el momento que un cliente envía el comando *fin*. El orden de lectura y escritura es el inverso al caso visto en el cliente.

Para la lectura y la escritura de peticiones concurrentes podemos usar los hilos de Java, *threads*, como se refleja en la figura 3.

En los ejemplos se adjuntan dos archivos que implementan el servidor de forma concurrente, *servidorConcurrente* e *hiloServidor*. El primero de ellos implementa la lógica de conexión y recepción de peticiones. El segundo el flujo de comunicación de una petición concreta.

Creación del cliente

Creamos el socket para conectarnos al servidor utilizando la clase *Socket* de la librería *java.net*. El constructor de esta clase soporta dos parámetros que indican la dirección y el puerto del servidor remoto al que nos queremos conectar. El puerto es simplemente un identificador que hace referencia a la aplicación del servidor a la que va destinada la información a enviar.

```
Socket skCliente = new Socket( HOST , Puerto );
```

Una vez establecida la conexión con el servidor podemos enviar la información necesaria utilizando la clase *Outputstream* (en este caso estamos utilizando socket de tipo stream o flujo de información pero la librería nos ofrece la posibilidad de utilizar sockets de tipo datagrama).

```
OutputStream aux = skCliente.getOutputStream();
DataOutputStream flujo= new DataOutputStream( aux );
flujo.writeUTF( "suma" );
```

Posteriormente, podemos usar la clase *Inputstream* para obtener la información devuelta por el servidor en respuesta a la petición realizada.

```
InputStream aux = skCliente.getInputStream();
DataInputStream flujo = new DataInputStream( aux );
System.out.println( flujo.readUTF() );
```

Finalmente, una vez finalizado el proceso de conversación se debe realizar el cierre de la conexión.

```
skCliente.close();
```

Compilación y ejecución

```
$ javac cliente.java
$ javac servidor.java
```

Una vez compilado procedemos a su ejecución. Para ello nos debemos asegurar que hemos incluido en el CLASSPATH el directorio contenedor o que ejecutamos en dicho directorio.

```
$ java servidor 9999
$ java cliente localhost 9999
```

Para probarlo podemos lanzar el cliente en una máquina y el servidor en otra.