**Clase del día - 24/02/2021**

La clase de hoy vamos a ver cómo programar un cliente y un servidor en Java.

Un cliente es un programa que **se conecta** a un programa servidor. Notar que el cliente inicia la conexión con el servidor.

Una vez que el cliente está conectado al servidor, el cliente puede enviar datos al servidor y el servidor puede mandar datos al cliente. A este tipo de comunicación se le conoce como **bi-direccional**, debido a que los datos pueden fluir en ambas direcciones.

En particular, los clientes y servidores que utilizaremos en el curso usan sockets TCP. Más adelante en el curso veremos los tipos de sockets y sus características.

Para compilar y ejecutar los programas del curso vamos a utilizar JDK8 desde la línea de comandos.

Los que quieran utilizar ambientes de desarrollo como Netbeans o Eclipse pueden hacerlo, sin embargo en general vamos a ejecutar los programas en la línea de comandos.

[**Cliente.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1673)

El programa [Cliente.java](https://m4gm.com/moodle/mod/resource/view.php?id=1673) es un ejemplo de un cliente de sockets TCP que se conecta a un servidor y posteriormente envía y recibe datos.

Primeramente vamos a crear un socket que se conectará al servidor. En este caso el servidor se llama "localhost" (computadora local) y el puerto abierto en el servidor es el 50000. En general el nombre del servidor puede ser un nombre de dominio (como midominio.com o una dirección IP). El número de puerto es un número entero entre 0 y 65535.

Socket conexion = new Socket("localhost",50000);

En este caso declaramos una variable de tipo Socket llamada "conexión" la cual va a contener una instancia de la clase Socket.

Es importante aclarar que antes de crear el socket, el programa servidor debe estar en ejecución y esperando una conexión, de otra manera la instrucción anterior produce una excepción, la cual desde luego debería controlarse dentro de un bloque **try**.

Para enviar datos al servidor a través del socket, vamos a crear un stream de salida de la siguiente manera:

DataOutputStream salida = new DataOutputStream(conexion.getOutputStream());

De la misma forma, para leer los datos que envía el servidor a través del socket, creamos un stream de entrada:

DataInputStream entrada = new DataInputStream(conexion.getInputStream());

Ahora podemos enviar y recibir datos del servidor. Veamos algunos ejemplos.

Vamos a enviar un entero de 32 bits, en este caso el número 123, utilizando el método **writeInt**:

salida.writeInt(123);

Ahora vamos a enviar un número punto flotante de 64 bits utilizando el método **writeDouble**:

salida.writeDouble(1234567890.1234567890);

Vamos a enviar  la cadena de caracteres "hola":

salida.write("hola".getBytes());

Debido a que el método **write** envía un arreglo de bytes, para enviar la cadena de caracteres "hola" es necesario convertirla a arreglo de bytes mediante el método **getBytes**.

Ahora supongamos que el servidor envía al cliente una cadena de caracteres. Para que el cliente reciba la cadena de caracteres es necesario que conozca el número de bytes que envía el servidor, en este caso el servidor envía una cadena de caracteres de 4 bytes.

Para recibir los bytes se utiliza el método **read** de la clase DataInputStream, sin embargo es importante tomar en cuenta que el método **read** podría leer solo una fracción del mensaje enviado.

Es un error muy común de los programadores creer que el método **read** siempre regresa el mensaje completo.

En realidad cuando un mensaje es largo, el método **read** debe ser invocado repetidamente hasta recibir el mensaje completo.

Para recibir el mensaje completo implementaremos un nuevo método **read** de la siguiente manera:

static void read(DataInputStream f,byte[] b,int posicion,int longitud) throws Exception

{

while (longitud > 0)

{

int n = f.read(b,posicion,longitud);

posicion += n;

longitud -= n;

}

}

En este caso, el método estático **read** regresará el mensaje completo en el arreglo de bytes "b". Notar que el método **read** de la clase DataInputStream regresa el número de bytes efectivamente leídos. Debido a que el método **read** de la clase DataInputStream puede producir una excepción, es necesario invocar este método dentro de un bloque try o bien. se debe utilizar la cláusula **throws** en el prototipo del método.

Para recibir la cadena de caracteres que envía el servidor,  vamos a invocar el método estático **read**:

byte[] buffer = new byte[4];

read(entrada,buffer,0,4);

System.out.println(new String(buffer,"UTF-8"));

Debido a que la variable buffer contiene los bytes correspondientes a la cadena de caracteres que envió el servidor, para obtener la cadena de caracteres utilizamos el constructor de la clase String para crear una cadena de caracteres a partir del arreglo de bytes indicando la codificación, en este caso UTF-8.

Ahora veremos cómo enviar de manera eficiente un conjunto de números punto flotante de 64 bits.

Supongamos que vamos a enviar cinco números punto flotante de 64 bits.

Primero "empacaremos" los números utilizando un objeto ByteBuffer. Cinco números punto flotante de 64 bits ocupan 5x8 bytes (64 bits=8 bytes). Entonces vamos a crear un objeto de tipo ByteBuffer con una capacidad de 40 bytes:

ByteBuffer b = ByteBuffer.allocate(5\*8);

Utilizamos el método **putDouble** para agregar cinco números al objeto ByteBuffer:

b.putDouble(1.1);

b.putDouble(1.2);

b.putDouble(1.3);

b.putDouble(1.4);

b.putDouble(1.5);

Para enviar el "paquete" de números, convertimos el objeto BytetBuffer a un arreglo de bytes utilizando el método **array** de la clase ByteBuffer:

byte[] a = b.array();

Entonces enviamos el arreglo de bytes utilizando el método **write**:

salida.write(a);

Para terminar el programa cerrar los streams de salida y entrada, así como la conexión con el servidor:

salida.close();

entrada.close();

conexion.close();

[**Servidor.java**](https://m4gm.com/moodle/mod/resource/view.php?id=1674)

El programa [Servidor.java](https://m4gm.com/moodle/mod/resource/view.php?id=1674) va a esperar una conexión del cliente, entonces recibirá los datos que envía el cliente y a su vez, enviará datos al cliente.

Primeramente vamos a crear un socket servidor que va a abrir, en este caso, el puerto 50000:

ServerSocket servidor = new ServerSocket(50000);

Notar que en Windows, por razones de seguridad el firewall solicita al usuario administrador permiso para abrir este puerto.

Ahora invocamos el método **accept** de la clase ServerSocket.

El método **accept** es bloqueante, lo que significa que el thread principal del programa quedará en estado de espera pasiva (una espera que no ocupa ciclos de CPU) hasta recibir una conexión del cliente. Cuando se recibe la conexión el método **accept** regresa un socket, en este caso vamos a declarar una variable de tipo Socket llamada "conexion":

Socket conexion = servidor.accept();

Una vez establecida la conexión con el cliente, el servidor podrá enviar y recibir datos.

Creamos un stream de salida y un stream de entrada:

DataOutputStream salida = new DataOutputStream(conexion.getOutputStream());

DataInputStream entrada = new DataInputStream(conexion.getInputStream());

Recordemos que el cliente envía un entero de 32 bits, entonces el servidor deberá recibir este dato utilizando el método **readInt**:

int n = entrada.readInt();

System.out.println(n);

Ahora el servidor recibe un número punto flotante de 64 bits utilizando el método **readDouble**:

double x = entrada.readDouble();

System.out.println(x);

El servidor recibe una cadena de cuatro caracteres:

byte[] buffer = new byte[4];

read(entrada,buffer,0,4);

System.out.println(new String(buffer,"UTF-8"));

El servidor envía una cadena de cuatro caracteres:

salida.write("HOLA".getBytes());

Ahora vamos a recibir los cinco números punto flotante empacados en un arreglo de bytes.

Recordemos que los cinco número punto flotante de 64 bits ocupan 40 bytes (5x8bytes).

byte[] a = new byte[5\*8];

read(entrada,a,0,5\*8);

Una vez recibido el arreglo de bytes, lo convertimos a un objeto BytetBuffer utilizando el método **wrap** de la clase ByteBuffer:

ByteBuffer b = ByteBuffer.wrap(a);

Para extraer los números punto flotante, utilizamos el método **getDouble** de la clase ByteBuffer:

for (int i = 0; i < 5; i++) System.out.println(b.getDouble());

Finalmente, cerramos los streams de salida y entrada, así como la conexión con el cliente:

salida.close();

entrada.close();

conexion.close();

**Actividades individuales a realizar**

1. Compile los programas [Cliente.java](https://m4gm.com/moodle/mod/resource/view.php?id=1673) y [Servidor.java](https://m4gm.com/moodle/mod/resource/view.php?id=1674)
2. Ejecute el programa [Servidor.java](https://m4gm.com/moodle/mod/resource/view.php?id=1674) en una ventana de comandos de Windows (o terminal de Linux) y ejecute el programa [Cliente.java](https://m4gm.com/moodle/mod/resource/view.php?id=1673) en otra ventana de comandos de Windows (o terminal de Linux).
3. Modifique el programa cliente para que envíe 10000 números punto flotante utilizando el método writeDouble (enviar los números 1.0, 2.0, 3.0 ... 10000.0). Mida el tiempo que tarda el programa cliente en enviar los 10000 números, se sugiere utilizar el método System.currentTimeMillis()
4. Modifique el programa servidor para que reciba los 10000 números que envía el programa cliente. Mida el tiempo que tarda el programa servidor en recibir los 10000 números.
5. Ahora modifique el programa cliente para que envíe los 10000 números utilizando ByteBuffer. Mida el tiempo que tarda el programa cliente en enviar los 10000 números.
6. Modifique el programa servidor para que reciba los 10000 números utilizando ByteBuffer. Mida el tiempo que tarda el programa servidor en recibir los 10000 números.
7. ¿Qué resulta más eficiente, enviar los números de manera individual mediante writeDouble o enviarlos empacados mediante ByteBuffer?