Práctica 1

Intercomunicación con sockets Java

1.1 Enunciado

El objetivo de la primera práctica es por una parte, familiarizar al alumno con el uso de las herramientas de construcción programas en Java, y por otra conseguir que el alumno alcance a apreciar el coste de la realización de llamadas a un servidor en diferentes modalidades.

1.1.1 Descripción

La práctica consistirá en la implementación y puesta a punto de un cliente simple, que realizará un número suficiente de llamadas a un servidor utilizando los métodos descritos más adelante. Tras lo anterior, el objetivo es medir el tiempo consumido por cada llamada. Se utilizarán tres tipos de servidores:

1. Un objeto local.

   En este caso, se está midiendo el coste de una llamada a subrutina local.

2. Un proceso esperando peticiones via UDP.

3. Un proceso esperando peticiones TCP.

Adicionalmente, para los tipos 2 y 3, se considerarán dos formas de localizarlos:

1. El servidor y el cliente están en el mismo nodo.

2. El servidor está en un nodo distinto al del cliente.

1.1.2 El servidor

Los servidores de tipo 2, y 3, son implementados dentro de un programa que se suministra con la práctica. Su lectura es aconsejable. Dicho programa podrá ser ejecutado con tres parámetros:

```
servidor (t|u) puerto duración
```

Los parámetros tienen el siguiente significado:

1. El primero indica TCP/IP si su valor es `t`, o UDP/IP si su valor es `u`.

2. El segundo indica el puerto en que esperará la petición.

3. El tercero indica cuánto tiempo se va a tardar en ejecutar la petición. La duración está expresada en milisegundos.

Para la primera práctica, el tercer parámetro será opcional, y su ausencia indicará un valor igual a 0. Eso querrá decir que el objeto servidor no retardará en absoluto la contestación al cliente.
En el caso 1, el servidor es un objeto. Dicho objeto deberá ser implementado en el propio programa cliente, y deberá también tener un método llamado `servicio_local`, que será invocado por el cliente.

1.1.3 El cliente

El código de cliente deberá ser escrito por el alumno. El esquema del código a desarrollar por la función principal aparece en la figura 1.1.
La forma exacta en que sea llamado el servidor dentro del bucle debe ser implantada de la forma que se crea más conveniente, aunque se aconseja la creación de una clase intermedia que incorpore tres métodos. Uno de ellos para la invocación local, otro para la UDP y el tercero para la TCP.
NOTA: Será necesario especificar por medio de parámetros del programa cliente: los valores de MAXLOOP, el puerto de contacto con el servidor, el nodo donde reside el servidor, y el método de contacto (TCP/UDP/local).

```
public class Cliente {
  public static void main( String[] args ) {
    Tomar tiempo
    for (i = 0; i < MAXLOOP; i++) {
      Enviar mensaje al servidor
      Esperar respuesta del servidor
    }
    Tomar tiempo
    tiempo por llamada = Dif.  de tiempo/MAXLOOP
  }
}
```

Figura 1.1: Algoritmo a seguir en el cliente.

1.1.4 Documentación a presentar

Como documentación a entregar deberá elaborarse un pequeño informe donde aparezca el código del programa cliente, una tabla con los valores obtenidos para tres valores razonables de MAXLOOP y una justificación sobre qué factores influyen en los resultados obtenidos y por qué se dan esas diferencias entre los casos estudiados. Este informe debe presentarse en la sesión inicial de la segunda práctica.

1.2 Comentarios

En las siguientes secciones se aclaran algunos puntos necesarios para resolver el enunciado presentado.

1.2.1 Obtención de tiempos

Para poder obtener el tiempo actual en Java, se puede utilizar la siguiente operación de la clase `System`:

```
static long CurrentTimeMillis();
```

Esta operación devuelve el tiempo actual en milisegundos, tomando como origen el 1 de enero de 1970. Por ser una operación estática, no es necesario instanciar ningún objeto de la clase `System` para poderla utilizar. Basta con añadir como

prefijo el nombre de la clase a la que pertenece y utilizar el punto para separar ambos componentes.

Otra cosa a tener en cuenta para obtener el tiempo invertido en un envío y recepción de mensaje está relacionada con cómo expresar el resultado. En Java, si se dividen dos números enteros, el resultado también será un número entero. Como los tiempos a tomar van a ser del orden de unos pocos milisegundos, convendría dar también algunos "decimales". Para ello convendría convertir, antes de efectuar la división correspondiente, tanto el tiempo transcurrido como el número de repeticiones a valores reales.

Por ejemplo, si tenemos el tiempo total transcurrido en una variable `tiempo` de tipo `long` y el número de repeticiones efectuadas en otra variable del mismo tipo llamada `nveces`, bastaría con escribir lo siguiente:

```
double tiempo2;
  tiempo2 = new Long( tiempo ).doubleValue() /
          ñew Long( nveces ).doubleValue();
```

Donde la clase `Long` permite mantener un valor de tipo `long` pero proporciona además un conjunto de operaciones para realizar las conversiones desde o hacia otros tipos. En nuestro caso, se emplea la operación `doubleValue()` para convertir el `long` a `double`. El valor a convertir se pasa como argumento en el constructor de la clase.

1.2.2 Uso de sockets

Para obtener una descripción detallada de las interfaces relacionadas con sockets en Java puede consultarse la documentación que acompaña al JDK. En caso de no tener acceso a ella, a continuación aparecen algunas de las operaciones presentes en estas clases (en concreto, aquellas que van a resultar necesarias para desarrollar la práctica). Téngase en cuenta que estas clases se encuentran en los paquetes `java.net` (para el caso de `DatagramPacket`, `DatagramSocket`, `InetAddress` y `Socket`) y `java.io` (para las clases `InputStream` y `OutputStream`). Por tanto, deberá realizarse el `import` correspondiente para tener acceso a ellas.

Clase `DatagramPacket`

Esta clase modela un paquete a transmitir mediante sockets UDP. Las operaciones a utilizar van a ser:

- `DatagramPacket( byte[] buf, int longitud )`

  Constructor utilizado para poder recibir paquetes de la longitud especificada en el segundo argumento. El contenido del paquete se dejará en el vector de octetos especificado en el primer argumento una vez se haya podido recibir la información.

  Se utilizará para recibir la contestación del servidor.

- `DatagramPacket( byte[] buf, int longitud, InetAddress dirección, int puerto )`

  Constructor utilizado para poder enviar paquetes con la longitud especificada en el segundo argumento hacia el ordenador cuya dirección y puerto se han dado en los dos siguientes.

  Se utilizará para construir el mensaje a enviar al servidor.

Clase `DatagramSocket`

Esta clase modela un socket UDP. Las operaciones a utilizar son:

- `DatagramSocket()`

  Construye un socket UDP y lo asocia a algún puerto libre en la máquina local. Aunque existen variantes para especificar un puerto y/o una máquina, no son de interés para el cliente (ver el código del servidor, en la sección 1.3) ya que aquí únicamente habrá que especificar la dirección destino en los paquetes a enviar.

- `void receive( DatagramPacket p )`

  El hilo de ejecución permanece bloqueado hasta que se ha recibido un paquete a través de este socket. La información recibida se encuentra en el vector asociado al paquete pasado como único argumento.

- `void send( DatagramPacket p )`

  Envía un paquete a través del socket. El paquete debe tener asociada la dirección y el puerto del socket a quien va dirigido el envío.

Clase `InetAddress`

Esta clase modela la dirección IP de una máquina. Se necesita para especificar la ubicación del servidor, bien sea en la máquina local o en una máquina remota. Las operaciones a utilizar son:

- `static InetAddress getByName( String host )`

  Dado un nombre de máquina, devuelve un objeto `InetAddress` con su dirección.

- `static InetAddress getLocalHost()`

  Devuelve un objeto `InetAddress` con la dirección de la máquina local.

Clase `Socket`

Esta clase implementa un socket cliente con comunicación TCP. Los sockets servidores para TCP están modelados por la clase `ServerSocket`, pero ésta ya no resulta necesaria para el caso del programa cliente. Obsérvese, además, que la comunicación TCP está orientada a conexión y en el programa servidor esta conexión se crea y se cierra para cada petición efectuada por el cliente. Por tanto, el cliente deberá estructurarse de igual manera.
Las operaciones a utilizar son:

- `Socket( String host, int puerto )`

  Crea un socket y lo conecta a la máquina remota cuyo nombre sea `host`, en el puerto especificado como segundo argumento.

- `InputStream getInputStream()`

  Obtiene el `InputStream` a utilizar para obtener la información recibida en el socket.

- `OutputStream getOutputStream()`

  Obtiene el `OutputStream` a utilizar para enviar información por el socket.

- `void close()`

  Cierra el socket, cortando la conexión con el servidor.

Clase `InputStream`

La única operación a utilizar en este caso es:

```
int read()
```

que devuelve el siguiente octeto presente en ese canal de entrada.
Aunque existen otras operaciones para obtener información desde el `InputStream`, como el servidor únicamente escribe un octeto, en el cliente sólo habrá que leer ese octeto.

Clase `OutputStream`

Al igual que en la clase anterior, únicamente hay que escribir un octeto en el socket, por lo que la operación a utilizar es:

```
void write( int octeto )
```

1.3 Programas facilitados

A continuación se presenta el código del programa servidor que se mencionaba en el enunciado.

```
/*************************************************************************/
/*                                                                       */
/* FICHERO: servidor.java                                                */
/*                                                                       */
/* DESCRIPCIÓN: Contiene la implementación del servidor de la 1a prác-   */
/*      tica. Ofrece un método por cada una de las peticiones que es     */
/*      capaz de atender (UDP y TCP). Existe una subclase "multi" que    */
/*      permite la atención de múltiples peticiones mediante múltiples   */
/*      hilos de ejecución (TCP).                                        */
/*                                                                       */
/*************************************************************************/

import java.net.*;              // El paquete java.net se necesita para //
                                // las clases Socket, ServerSocket,     //
                                // DatagramSocket y DatagramPacket.     //
import java.io.*;               // El paquete java.io se necesita para  //
                                // las clases InputStream y OutputStream//

/*=======================================================================*/
/* servidor                                                              */
/*      Clase que implementa un método por cada modalidad de llamada.   */
/*                                                                       */
/*=======================================================================*/
```

```
public class servidor {
                              // La subclase multi implementa el      //
                              // código de los hilos de ejecución que //
                              // se utilizan en la variante multi-    //
                              // thread del servidor TCP.             //
    class multi  implements Runnable {
        Socket so;            // El socket "so" se utiliza para con-  //
                              // testar una petición, después se      //
                              // cierra.                              //
        long dur;             // "dur" mantiene la duración de la     //
                              // espera asociada a cada petición.     //

                              // En el constructor se pasa el socket  //
                              // TCP a utilizar y la duración de la   //
                              // espera de la próxima petición.       //
        public multi(Socket s, long duration) {
            so = s;
            dur = duration;
        }
                              // Este es el código que ejecuta cada   //
                              // uno de los hilos servidores.         //
        public synchronized void  run() {
            try {
                              // Se obtiene un stream de entrada      //
                              // desde el socket.                     //
                InputStream  is = so.getInputStream();
                              // Se hace lo mismo para el stream de   //
                              // salida en el que se va a dejar la    //
                              // respuesta.                           //
                OutputStream os = so.getOutputStream();
                int ch;

                              // El cliente sólo envía un carácter.   //
                ch = is.read();
                if ( dur > 0 )
                  wait(dur);  // Esperamos el tiempo indicado.        //
                os.write(65); // La contestación sólo es un carácter. //
                so.close();   // Se cierra el socket, ya no se va a   //
                              // volver a utilizar.                   //
            } catch (IOException e) {
                System.out.println("Algo fue mal");
            } catch (InterruptedException e) {
                System.out.println("Adiós");
            }
        }
    }

                              // Método a utilizar con un cliente     //
                              // UDP.                                 //
    public synchronized void serverudp(int port, long dur) {
            DatagramSocket ds;// Socket para UDP.                     //
                              // El contenido del paquete UDP va a    //
                              // ser un vector de bytes con una sola  //
```

```
                            // componente.                          //
        byte[]         buf = new byte[1];
                            // El paquete UDP se crea con el vector //
                            // anterior.                            //
        DatagramPacket dp  = new DatagramPacket(buf,1);

        try {               // Hay que crear un socket UDP asociado //
                            // al puerto especificado como primer   //
                            // argumento.                           //
            ds = new DatagramSocket(port);
        } catch (Exception e) {
            System.out.println("No se ha podido asociar el socket " +
                    "UDP al puerto " + port);
            return;
        }

        while (true) {
            try {
                            // Se recibe una petición por parte del //
                            // cliente. Una vez obtenida hay que    //
                            // esperar el tiempo indicado para si-  //
                            // mular la ejecución de la petición.   //
                ds.receive(dp);
                if ( dur > 0 )
                    wait(dur);
                            // Al final se contesta, reenviando el  //
                            // mismo paquete recibido.              //
                ds.send(dp);
            } catch (IOException e) {
                System.out.println("No se ha podido recibir.");
                return;
            } catch (InterruptedException e) {
                System.out.println("Adiós");
                return;
            }
        }
}

                            // Método a utilizar con clientes TCP.  //
public synchronized void servertcp(int port, long dur) {
    ServerSocket ss;
    Socket       so;
    InputStream  is;
    OutputStream os;

    try {                   // Se crea el socket servidor ss, aso-  //
                            // ciándolo al número de puerto especi- //
                            // ficado como primer argumento.        //
        ss = new ServerSocket(port);
    } catch (Exception e) {
        System.out.println("No se puede asociar el puerto " + port +
            " al socket TCP.");
        return;
```

```
        }

        while (true) {
            try {
                                // Esperar conexión por parte del       //
                                // cliente y generar un nuevo socket    //
                                // para atenderla cuando se establezca. //
                so = ss.accept();
                                // "is" va a permitir la lectura de     //
                                // información existente en el socket.  //
                is = so.getInputStream();
                                // "os" se utiliza para escribir infor- //
                                // mación que podrá obtener el cliente. //
                os = so.getOutputStream();
                int ch;

                ch = is.read(); // Esperar hasta poder leer un byte.    //
                if ( dur > 0 )
                  wait(dur);    // Simular la atención de la petición.  //
                os.write(65);   // Enviar la respuesta.                 //
                so.close();     // Cerrar el socket utilizado para      //
                                // esta conexión.                       //
            } catch (IOException e) {
                System.out.println("Algo fue mal");
            } catch (InterruptedException e) {
                System.out.println("Adiós");
            }
        }
    }

                                // Versión TCP con mútiples hilos.      //
    public synchronized void servertcpmt(int port, long dur) {
        ServerSocket ss;
        Socket       so;

        try {
                                // Asociar un puerto al socket servidor //
                                // a utilizar para esperar las conexio- //
                                // nes de los clientes.                 //
            ss = new ServerSocket(port);
        } catch (Exception e) {
            System.out.println("No se puede asociar el puerto " + port +
                " al socket TCP.");
            return;
        }

        while (true) {
            try {
                                // Aceptar las conexiones de los clien- //
                                // tes, creando un nuevo socket por     //
                                // cada una de ellas.                   //
                so = ss.accept();
                                // Crear un thread "multi" que sirva    //
```

```
                            // la petición que acaba de llegar.     //
               new Thread(new multi(so,dur)).start();
            } catch (IOException e) {
                System.out.println("Algo fue mal");
            }
        }
    }

    public static void main(String[] args) {
        int        port = 8000;
        boolean    udp  = false;
        boolean    MT   = false;
        long       duration = 0;
        servidor   worker   = new servidor();

                            // Extraccion de argumentos.            //
        if (args.length < 2) {
            System.out.println("Número de argumentos erróneo");
            return;
        }
                            // Comprobar el primer argumento. Si es //
                            // "u", utilizar el servidor UDP. Si es //
                            // "t", utilizar el servidor TCP. En    //
                            // cualquier otro caso, utilizar el     //
                            // servidor multithread.                //
        if (args[0].equalsIgnoreCase("u")) {
            System.out.println("Servidor UDP");
            udp = true;
        } else if (args[0].equalsIgnoreCase("t")) {
            System.out.println("Servidor TCP");
            udp = false;
        } else {
            System.out.println("Servidor TCP MT");
            MT = true;
        }

        try {                   // Obtener el número de puerto a partir //
                                // del segundo argumento.               //
            port = (new Integer(args[1])).intValue();
        } catch (Exception e) {
            System.out.println(args[1] + "no es un número válido de puerto");
            return;
        }

                                // Si existe un tercer argumento será   //
                                // la duración a asignar al servicio de //
                                // cada petición.                       //
        if (args.length == 3) {
            try {
                duration = (new Long(args[2])).longValue();
                                // La duración no puede ser negativa.   //
                if (duration < 0)
                        duration = -duration;
```

```
            } catch (Exception e) {
                System.out.println(args[2] + "no es un valor adecuado de" +
                                   " duración.");
                return;
            }
        } else {
            duration = 0;
        }

                                // Lógica del programa.                 //
        if (udp) {
            worker.serverudp(port,duration);
        } else if (MT) {
            worker.servertcpmt(port,duration);
        } else {
            worker.servertcp(port,duration);
        }
    }
}
```