Comunicación Sockets

José Luis Quiroz Fabián

Los *sockets* son el mecanismo a más bajo nivel para comunicar procesos por medio de mensajes, éstos nos permiten comunicar procesos en una misma computadora o bien en computadoras en red. Usando *sockets* podemos establecer un canal de datos fiable TCP (garantizando que todo dato que un emisor envía le llegará el receptor) y no fiable UDP (no hay garantía que los datos lleguen o bien lleguen repetidos). En los siguiente ejemplos veremos la comunicación de procesos mediante Sockets TCP y UDP con Java.

1 Sockets Orientados a Conexión

Las clases **Socket** y **ServerSocket** permiten utiliza el protocolo TCP en Java. Un **Socket** se utiliza para transmitir y recibir datos. Un **ServerSocket** trabaja en el servidor y permite esperar a que un cliente quiera establecer una conexión con el servidor. Para la comunicación, el cliente crea un **Socket** para solicitar una conexión con el servidor al que desea conectarse. Cuando el **ServerSocket** recibe la solicitud, crea un **Socket** en un puerto que no se esté usando y la conexión entre cliente y servidor queda establecida. Entonces, el **SocketServer** vuelve a quedarse escuchando para recibir nuevas peticiones de clientes (ver Figuras 1 y 2).

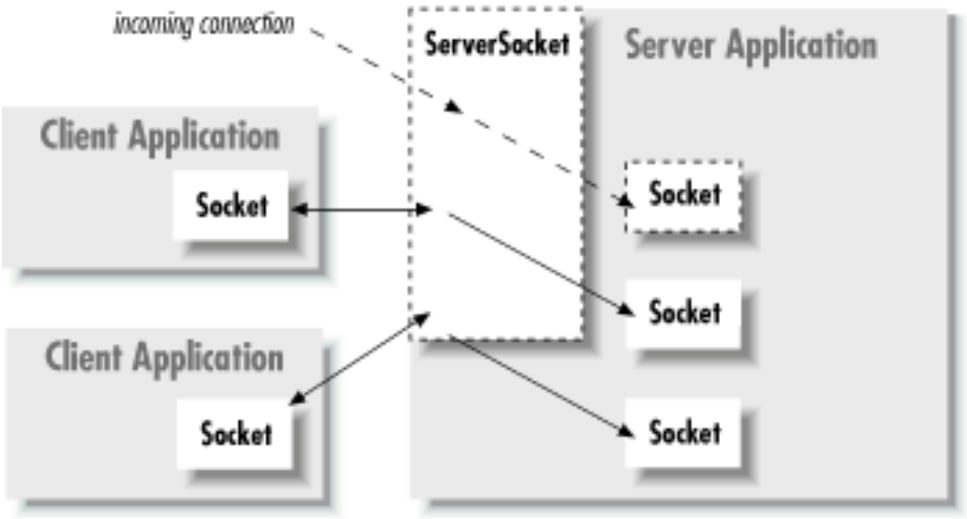

Figure 1: Comunicación por medio de Sockets en Java

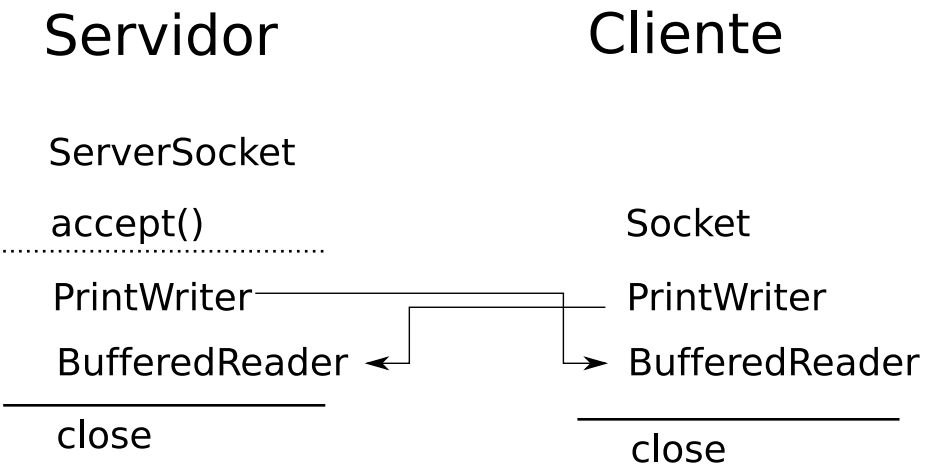

Figure 2: Comunicación por medio de Sockets TCP en Java

En la Figura 1 y 2 se observa el modelo de comunicación fiable mediante Sockets en Java. Como se observa en la Figura, el servidor puede atender varios clientes por medio de objetos de tipo Socket. Estos objetos son creados al establecer una conexión con un cliente. Lo anterior se explica en los Códigos 1 y 2.

1.1 Ejemplo Socket TCP

Este ejemplo muestra la comunicación simple entre un Servidor y un Cliente.

1.1.1 Servidor

En el Código 1 se muestra el servidor en el cual se utiliza la clase *ServerSocket* (línea 24) para recibir conexiones de los clientes mediante en protocolo TCP. Toda aplicación que actué como servidor debe crear una instancia de esta clase y debe invocar a su método *accept()* (línea 26); la invocación a dicho método es bloqueante, esto es, el servidor permanece esperando hasta que llegue una conexión por parte de algún cliente. Cuando sucede esto, el método *accept()* creará una instancia de la clase *Socket* (línea 26) que se utiliza para comunicarse con el cliente. Posteriormente se utilizan las clases *BufferedReader* y *PrintWriter* para la comunicación con el cliente, la entrada/recepción y salida/envío de mensajes (lineas 34 y 36). Al final, se utilizan los métodos *close* para cerrar los *Sockets*, cerrar el canal de comunicación (líneas 52 y 54).

```
1  import java.net.*;
2  import java.io.*;
3
4
5
6  public class Servidor {
7
8      final int PUERTO=4002;
9
10     ServerSocket sc;
```

```
11
12      Socket so;
13
14      PrintWriter salida;
15
16      String mensajeRecibido;
17
18      //SERVIDOR
19      public void initServidor(){
20
21          BufferedReader entrada;
22          try{
23
24              sc = new ServerSocket(PUERTO );/* crea socket
                    servidor que escuchara en un puerto */
25              System.out.println("Esperando una conexion:");
26              so = sc.accept();
27
28              //Inicia el socket, ahora esta esperando una
                    conexion por parte del cliente
29
30              System.out.println("Un cliente se ha conectado.
                    ");
31
32              //Canales de entrada y salida de datos
33
34              entrada = new BufferedReader(new
                    InputStreamReader(so.getInputStream()));
35
36              salida = new PrintWriter(so.getOutputStream(),
                    true);
37
38              System.out.println("Confirmando conexion al
                    cliente....");
39
40              salida.println("Mensaje 1");
41
42              mensajeRecibido = entrada.readLine();
43
44              System.out.println(mensajeRecibido);
45
46              salida.println("Mensaje 2");
47
48              salida.println("Mensaje 3");
49
50              System.out.println("Cerrando conexion...");
51
52              so.close();
53
54              sc.close();
55
56          }catch(Exception e ){
57
58              System.out.println("Error: "+e.getMessage());
```

```
59
60          }
61      }
62      public static void  main(String[] args){
63
64          Servidor s = new Servidor();
65          s.initServidor();
66
67      }
68  }
```

Código 1: Servidor

1.1.2 Cliente

En el Código 2 se muestra el cliente en el cual se utiliza la clase *Socket* (línea 29) para comunicarse con un servidor. En el Socket se especifica el nombre o la dirección IP del servidor y el puerto por donde se comunicará (línea 29). Posteriormente, al igual que en el servidor se utilizan las clases *PrintWriter* y *BufferedReader* para la comunicación, la entrada/recepción y salida/envío de mensajes (lineas 33 y 35). Al final, se utilizan el método *close* para cerrar el canal de comunicación con el servidor (líneas 57).

```
1   import java.net.*;
2
3   import java.io.*;
4
5
6
7   public class Cliente {
8
9       final String HOST = "localhost";
10
11      final int PUERTO=4002;
12
13      Socket sc;
14
15      PrintWriter mensaje;
16
17      BufferedReader entrada;
18
19      String mensajeRecibido;
20
21      //Cliente
22
23      public void initCliente(){
24
25
26
27          try{
28
29              sc = new Socket( HOST , PUERTO ); /*conectar a
                    un servidor en localhost con puerto 5000*/
30
```

```
31              //creamos el flujo de datos por el que se
                    enviara un mensaje
32
33              mensaje = new PrintWriter(sc.getOutputStream(),
                     true);;
34
35              entrada = new BufferedReader(new
                    InputStreamReader(sc.getInputStream()));
36
37              System.out.println("A intercambiar mensajes....
                    ");
38
39              //enviamos el mensaje
40
41              mensaje.println("hola que tal!!");
42
43              mensajeRecibido = entrada.readLine();
44
45              System.out.println(mensajeRecibido);
46
47              mensajeRecibido = entrada.readLine();
48
49              System.out.println(mensajeRecibido);
50
51              mensajeRecibido = entrada.readLine();
52
53              System.out.println(mensajeRecibido);
54
55              //cerramos la conexion
56
57              sc.close();
58
59          }catch(Exception e ){
60
61              System.out.println("Error: "+e.getMessage());
62
63          }
64
65      }
66      public static void  main(String[] args){
67
68          Cliente c = new Cliente();
69          c.initCliente();
70
71      }
72  }
```

Código 2: Cliente

1.2 Servidor multihilado

En el Código 3 se muestra el servidor multihilado. En este ejemplo, para la creación de hilos se utiliza la interfaz Runnable (línea 7). El Servidor cada que acepta una conexión (línea 40) crea un hilo para que se encargue de dicha

conexión (líneas 42-44). El hilo se crea y ejecuta el método run (línea 63) mientras el hilo principal regresa en el ciclo para esperar más conexiones (líneas 38-52).

```
1
2  import java.net.*;
3  import java.io.*;
4
5
6
7  public class ServidorMultiHilado implements Runnable {
8    static final int PUERTO=4002;
9    Socket s;
10
11   public ServidorMultiHilado(){
12
13     initServidor();
14
15   }
16
17   public ServidorMultiHilado(Socket s) {
18
19     this.s=s;
20
21   }
22
23
24
25   //SERVIDOR
26   public void initServidor(){
27
28
29     ServerSocket sc;
30
31     Socket so;
32
33
34     try{
35
36       sc = new ServerSocket(PUERTO );/* crea socket
             servidor que escuchara en puerto 5000*/
37
38       while(true) {
39         System.out.println("Esperando una conexion:");
40         so = sc.accept();
41
42         ServidorMultiHilado hilo = new ServidorMultiHilado(
               so);
43         Thread tcliente = new Thread(hilo);
44         tcliente.start();
45
46
47         //Inicia el socket, ahora esta esperando una
               conexion por parte del cliente
```

```
48
49          System.out.println("Un cliente se ha conectado.");
50
51        }
52
53
54
55      }catch(Exception e ){
56
57        System.out.println("Error: "+e.getMessage());
58
59      }
60    }
61
62    @Override
63    public void run() {
64      //Canales de entrada y salida de datos
65      PrintWriter salida=null;
66
67      String mensajeRecibido="";
68
69      BufferedReader entrada=null;
70
71      try {
72        entrada = new BufferedReader(new InputStreamReader(s.
              getInputStream()));
73      } catch (IOException e3) {
74        // TODO Auto-generated catch block
75        e3.printStackTrace();
76      }
77
78      try {
79        salida = new PrintWriter(s.getOutputStream(), true);
80      } catch (IOException e2) {
81        // TODO Auto-generated catch block
82        e2.printStackTrace();
83      }
84
85      System.out.println("Confirmando conexion al cliente....
          ");
86
87      salida.println("Mensaje 1");
88
89      try {
90        mensajeRecibido = entrada.readLine();
91      } catch (IOException e1) {
92        // TODO Auto-generated catch block
93        e1.printStackTrace();
94      }
95
96      System.out.println(mensajeRecibido);
97
98      salida.println("Mensaje 2");
99
```

```
100       salida.println("Mensaje 3");
101
102       System.out.println("Cerrando conexion...");
103
104       try {
105         s.close();
106       } catch (IOException e) {
107         // TODO Auto-generated catch block
108         e.printStackTrace();
109       }
110
111
112     }
113     public static void  main(String[] args){
114
115       ServidorMultiHilado s = new ServidorMultiHilado();
116
117     }
118   }
```

Código 3: Servidor Multihilado

2 Sockets NO Orientados a Conexión

El tipo más sencillo de sockets son los UDP, puesto que no es necesario establecer ninguna conexión para enviar y recibir datos. En Java, un objeto de tipo **DatagramSocket** representa un socket UDP y puede enviar o recibir datos directamente de otro socket UDP. Los datos se envían y reciben en paquetes denominados datagramas. Los datagramas se representan en Java mediante la clase **DatagramPacket**, que consiste simplemente en un array de bytes dirigido a una dirección IP y a un puerto UDP concreto (ver Figura 3).

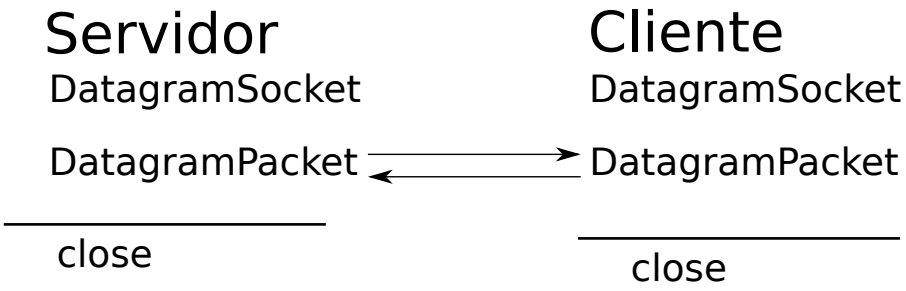

Figure 3: Comunicación por medio de Sockets UDP en Java

2.1 Servidor UDP

```
1  import java.net.*;
2  import java.io.*;
3
4  public class ServidorUDP {
5    static final int MAX=1000;
```

```
6   static final int PUERTO=3000;
7   public static void main (String args[]) {
8
9
10    try {
11
12      DatagramSocket socketUDP = new DatagramSocket(PUERTO)
            ;
13      byte[] bufer = new byte[MAX];
14
15      while (true) {
16        // Construimos el paquete para recibir peticiones
17        DatagramPacket paquete = new DatagramPacket(bufer,
              bufer.length);
18
19        //Leemos una petición
20        socketUDP.receive(paquete);
21
22        System.out.println("Conexion: " + paquete.
              getAddress()+" Puerto: "+paquete.getPort());
23
24        System.out.println("Servidor: He recibido -> "+new
              String(paquete.getData()));
25
26        String respuesta="Bien!\0";
27
28        byte[] respuestaBytes=respuesta.getBytes();
29
30        // Construimos el paquetet para enviar la respuesta
31        paquete = new DatagramPacket(respuestaBytes,
              respuesta.length(),
32                                     paquete.getAddress(), paquete.
                                           getPort());
33
34        //Enviamos la respuesta
35        socketUDP.send(paquete);
36      }
37
38    } catch (SocketException e) {
39      System.out.println("Socket: " + e.getMessage());
40    } catch (IOException e) {
41      System.out.println("IO: " + e.getMessage());
42    }
43  }
44 }
```

Código 4: Servidor UDP

2.2 Cliente

```
1  import java.net.*;
2  import java.io.*;
3
4  public class ClienteUDP {
5
```

```
6   static final String SERVIDOR="127.0.0.1";
7
8   static final int PUERTO=3000;
9
10  static final int MAX=1000;
11
12  // Los argumentos proporcionan el mensaje y el nombre del
        servidor
13  public static void main(String args[]) {
14
15    try {
16      DatagramSocket socketUDP = new DatagramSocket();
17      String mensaje="Hola, ¿como estas?\0";
18      byte[] mensajeBytes = mensaje.getBytes();
19      InetAddress hostServidor = InetAddress.getByName(
            SERVIDOR);
20
21      // Construimos un datagrama para enviar el mensaje al
            servidor
22      DatagramPacket paquete =
23        new DatagramPacket(mensajeBytes, mensaje.length(),
              hostServidor,
24                         PUERTO);
25
26      // Enviamos el datagrama
27      socketUDP.send(paquete);
28
29      // Construimos el DatagramPacket que contendrá la
            respuesta
30      byte[] bufer = new byte[MAX];
31
32      paquete = new DatagramPacket(bufer, bufer.length);
33      socketUDP.receive(paquete);
34
35      // Enviamos la respuesta del servidor a la salida
            estandar
36      System.out.println("Respuesta: " + new String(paquete
            .getData()));
37
38      // Cerramos el socket
39      socketUDP.close();
40
41    } catch (SocketException e) {
42      System.out.println("Socket: " + e.getMessage());
43    } catch (IOException e) {
44      System.out.println("IO: " + e.getMessage());
45    }
46  }
47 }
```

Código 5: Cliente UDP

3 Ejercicio

Implementar una calculadora con Sockets TCP.

- Cada cliente define de inicio su nickname con el cual se registra en el servidor. Si ya se tiene el nickname con otro cliente, el servidor se lo indica al cliente para que lo cambie.
- La calculadora debe ser multihilada
- Las operaciones que realiza el servidor son:
  - Sumar n números
  - Multiplicar n números
  - Restar dos números
  - Dividir dos números
- Los resultados que envía el servidor siempre tiene como prefijo el nickname del cliente.
- El cliente se puede desconectar en cualquier momento.