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 esta práctica solo nos enfocaremos en los sockets TCP.

1 Sockets TCP

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).

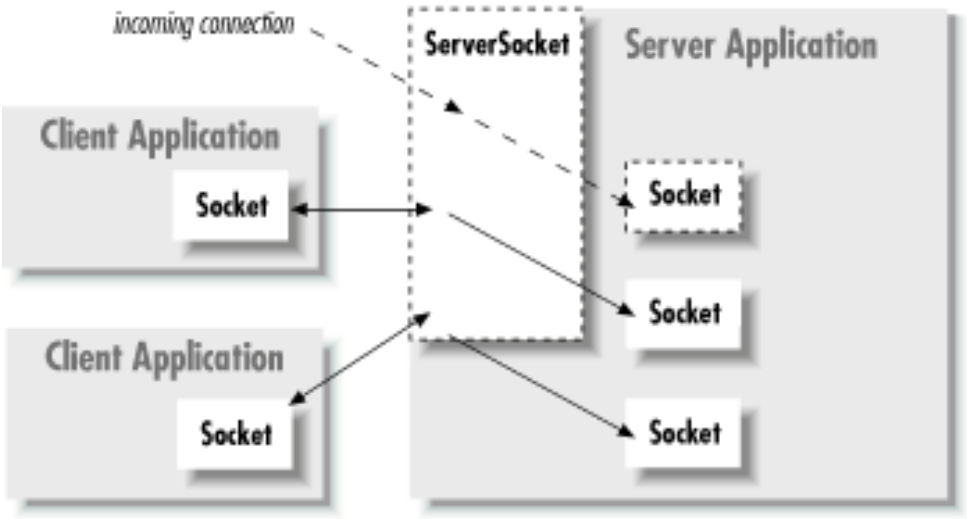

Figure 1: Comunicación por medio de Sockets en Java

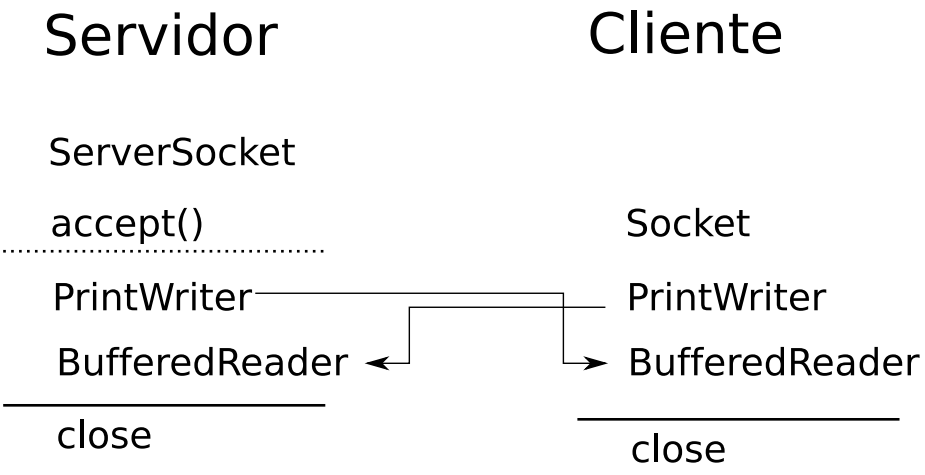

Figure 2: Comunicación por medio de Sockets TCP en Java

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.

1.2 Servidor

En el Código 1 se muestra el servidor en el cual se utiliza la clase *ServerSocket* (línea 9) 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 13); 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 13) 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 16 y 17). Al final, se utilizan el método *close* para cerrar el *Socket*, cerrar el canal de comunicación con el cliente (línea 24).

```
1  // Servidor
2  import java.io.*;
3  import java.net.*;
4
5  public class Servidor {
6      public static void main(String[] args) throws
           IOException {
7          final int PUERTO = 12345;
8
```

```
9           try (ServerSocket serverSocket = new ServerSocket(
                PUERTO)) {
10              System.out.println("Servidor iniciado y
                    escuchando en el puerto: " + PUERTO);
11
12              while (true) {
13                  Socket clienteSocket = serverSocket.accept
                        ();
14                  System.out.println("Cliente conectado: " +
                        clienteSocket.getRemoteSocketAddress());
15
16                  BufferedReader entrada = new BufferedReader
                        (new InputStreamReader(clienteSocket.
                        getInputStream()));
17                  PrintWriter salida = new PrintWriter(
                        clienteSocket.getOutputStream(), true);
18
19                  String mensajeCliente = entrada.readLine();
20                  System.out.println("Recibido del cliente: "
                         + mensajeCliente);
21
22                  salida.println("Hola, Cliente! Recibí tu
                        mensaje: " + mensajeCliente);
23
24                  clienteSocket.close();
25              }
26          }
27      }
28  }
```

Código 1: Servidor

1.3 Cliente

En el Código 2 se muestra el cliente en el cual se utiliza la clase *Socket* (línea 10) 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. Posteriormente, al igual que en el servidor se utilizan las clases *PrintWriter* y *BufferedReader* para la comunicación (lineas 11 y 12).

```
1   // Cliente.java
2   import java.io.*;
3   import java.net.*;
4
5   public class Cliente {
6       public static void main(String[] args) throws
            IOException {
7           final String DIRECCION_SERVIDOR = "localhost";
8           final int PUERTO = 12345;
9
10          try (Socket socket = new Socket(DIRECCION_SERVIDOR,
                 PUERTO)) {
11              PrintWriter salida = new PrintWriter(socket.
                    getOutputStream(), true);
```

```
12              BufferedReader entrada = new BufferedReader(new
                    InputStreamReader(socket.getInputStream()))
                ;
13
14              String mensaje = "Hola, Servidor!";
15              salida.println(mensaje);
16              System.out.println("Mensaje enviado al servidor
                    : " + mensaje);
17
18              String respuestaServidor = entrada.readLine();
19              System.out.println("Respuesta del servidor: " +
                    respuestaServidor);
20          }
21      }
22  }
```

Código 2: Cliente

1.4 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 5). El Servidor cada que acepta una conexión (línea 36) crea un hilo para que se encargue de dicha conexión (líneas 37-39). El hilo se crea y ejecuta el método run (línea 53) mientras el hilo principal regresa en el ciclo para esperar más conexiones (línea 34).

```
1   // Servidor
2   import java.io.*;
3   import java.net.*;
4
5   public class ServidorMultiHilado  implements Runnable {
6
7       static final int PUERTO = 12345;
8       Socket s;
9       public ServidorMultiHilado(){
10
11          try{
12              initServidor();
13          }catch(Exception e){
14              System.out.println("Error en el servidor: "+e.
                    getMessage());
15          }
16      }
17
18      public ServidorMultiHilado(Socket s) {
19
20          this.s=s;
21
22      }
23
24      //SERVIDOR
25      public void initServidor() throws Exception {
26
```

```
27
28          ServerSocket sc;
29          Socket so;
30
31
32           try (ServerSocket serverSocket = new ServerSocket(
                PUERTO)) {
33
34              while(true) {
35                  System.out.println("Esperando una conexion:
                        ");
36                  so = serverSocket.accept();
37                  ServidorMultiHilado hilo = new
                        ServidorMultiHilado(so);
38                  Thread tcliente = new Thread(hilo);
39                  tcliente.start();
40
41                  //Inicia el socket, ahora esta esperando
                        una conexion por parte del cliente
42
43                  System.out.println("Un cliente se ha
                        conectado.");
44
45              }
46
47
48
49          }
50      }
51
52      @Override
53      public void run() {
54          //Canales de entrada y salida de datos
55          PrintWriter salida=null;
56
57          String mensajeRecibido="";
58
59          BufferedReader entrada=null;
60
61          try {
62              entrada = new BufferedReader(new
                    InputStreamReader(s.getInputStream()));
63
64          salida = new PrintWriter(s.getOutputStream(), true)
                ;
65
66          System.out.println("Confirmando conexion al cliente
                ....");
67
68          String mensajeCliente = entrada.readLine();
69
70          System.out.println("Recibido del cliente: " +
                mensajeCliente);
71
```

```
72          System.out.println(mensajeRecibido);
73
74          salida.println("Hola, Cliente! Recibí tu mensaje: "
                + mensajeCliente);
75
76          System.out.println("Cerrando conexion...");
77
78          s.close();
79          } catch (IOException e) {
80              // TODO Auto-generated catch block
81              e.printStackTrace();
82          }
83
84
85      }
86
87
88      public static void main(String[] args) throws
            IOException {
89
90
91          ServidorMultiHilado s = new ServidorMultiHilado();
92      }
93  }
```

Código 3: Servidor Multihilado

2 Ejercicios

1. Implementar el pipeline visto en el examen: La suma global de un conjunto de N números se pueden obtener mediante el uso de un pipeline circular. Cada proceso de inicio genera un valor y después lo envía a su siguiente en el pipeline. Al final de un número de iteraciones, donde cada proceso reenvía cada valor que recibe, los procesos tiene la suma global (ver el ejemplo de la Figura 3 para $N = 4$). Realice un pseudocódigo para N procesos, donde $N > 3$. Los valores de inicio de cada proceso se puede suponer que se generan de forma aleatoria. Implementar sus operaciones *Enviar* y *Recibir*.

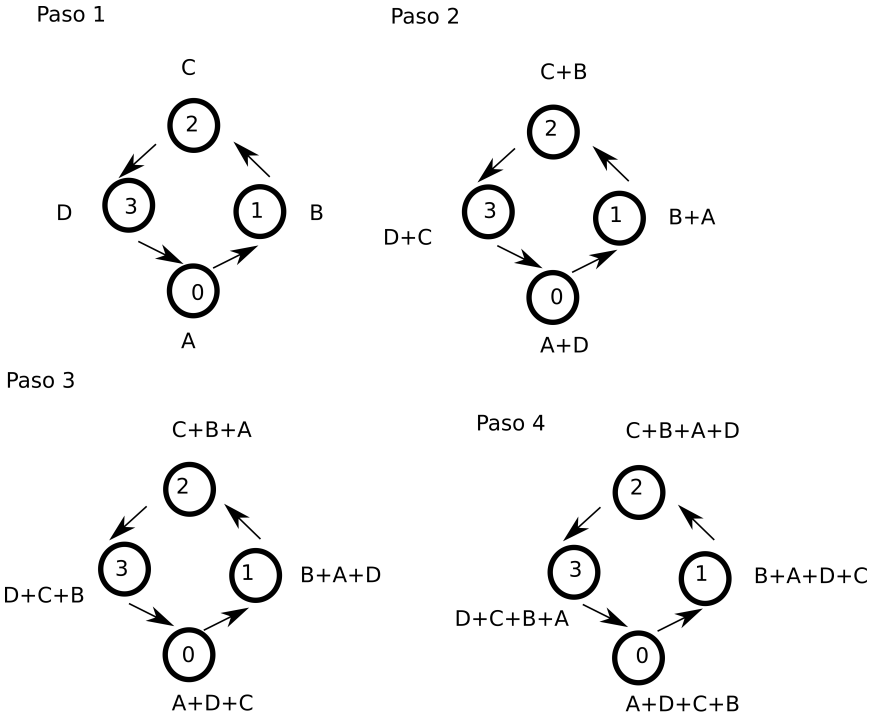

Figure 3: Comunicación por medio de Sockets TCP en Java

2. Implementar una calculadora con Sockets TCP.

   - Cada cliente define de inicio su nickname con el cual se registra en el servidor. El servidor debe verificar que el nickname no exista. 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 y se tiene que quitar su registro (nickname) en el servidor.