UNIVERSIDAD POLITÉCNICA SALESIANA

MATERIA: SISTEMAS DISTRIBUIDOS

Mecanismos de comunicación - Sockets

Realizado por:
Esteban Gustavo Novillo Quinde

ÍNDICE DE FIGURAS

I. Introducción

Se ha hablado continuamente de la necesidad de comunicación entre los diversos terminales, bien sean estos de una organización o externos; por esta razón entre una de las diversas soluciones hacen su aparición, como una analogía al término en inglés para conector, los Sockets. Limi Kalita en [1] define a un socket como un punto final de un enlace de comunicación entre procesos. Los procesos aprovechan esta capacidad para intercambiar datos con otros transmitiendo y recibiendo información identificando el destinatario con una dirección IP y un puerto de enlace. Se los puede clasificar acorde a la forma en la cual se transfiere la información a través de este:

- Sockets Stream (TCP):
  Son un tipo de comunicación orientada a la conexión, es decir los datos se transfieren sin necesidad de un registro mientras exista una comunicación establecida. En caso de que esta se vea interrumpida se notifica tanto al cliente como al servidor. El protocolo que se emplea para cumplir con este esquema de servicio es el TCP, puesto que debe existir un servidor que atienda las peticiones de conexión y uno o varios clientes que soliciten la misma. Una vez se establezca una conexión cliente-servidor inicia la transferencia bidireccional de los datos.

- Sockets Datagrama (UDP):
  Son un servicio sin conexión, por lo que están mucho más optimizados que la comunicación basada en TCP; no obstante, su principal inconveniente es que no se garantiza la entrega del mensaje. La comunicación se efectúa mediante paquetes, por lo que no existe un control de si estos se encuentran duplicados, perdidos o desordenados. Además, como la comunicación no se da en un canal cerrado bidireccional, sino de manera global entre paquetes, es necesario añadir cabeceras con la información del remitente y destinatario.

Finalmente, se puede mencionar a una tercera clasificación, pero esta se usa mayormente para la depuración de código por lo que a nivel corporativo y de implementación no es necesariamente relevante.

- Sockets Raw
  Este tipo de sockets otorgan acceso sin restricciones a capas superiores e inferiores, es decir establecen una comunicación directa. Como no contienen ningún patrón de seguridad u optimización se los emplea mayormente para depurar los nuevos protocolos desarrollados.

II. Descripción de la práctica

La práctica a desarrollar consta de los siguientes puntos:

1. Ampliar lo estudiado por medio de una investigación referente a la implementación de Sockets en Java.
2. Desarrollar una aplicación de chat utilizando Sockets.
3. Realizar un informe de la investigación realizada y de los pasos para la implementación de la aplicación.
4. Realizar un vídeo referente al funcionamiento de la aplicación.

III. Resolución de la práctica

Para hacer uso de los sockets en Java solo hace falta llamar al paquete java.net [2], el mismo mismo que incluye una clase denominada Socket, la cual directamente implementa el lado del cliente para la conexión; así como la clase ServerSocket, que permite definir el servidor. Es importante aclarar desde ya que el lado del cliente y del servidor deben ser programados y ejecutados en programas distintos, o desde dos consolas separadas para poder simular correctamente el funcionamiento entre dos computadores en red al iniciarse dos procesos separados. Los métodos más importantes para esta conexión son:

- socket(): permite crear un nuevo socket con un uid único.
- bind(): dado por el lado del servidor, permitiendo darle un número de puerto y una dirección ip para escuchar las peticiones.

- listen(): de igual manera se ejecuta del lado del servidor. Habilita al TCP en modo de escucha.
- conect(): se emplea en el lado del cliente para iniciar una conexión.
- acept(): del lado del servidor, permite aceptar una conexión entrante que reuna las características especificadas.
- Los métodos para enviar y recibir datos de o hacia un socket remoto son los siguientes:
  - send()
  - recv()
  - write()
  - read()
  - sendto()
  - recvfrom()
- close(): permite cerrar la conexión así como devolver el puerto y liberar el buffer empleado para los datos salientes de manera local

Este paquete es recomendable ejecutarlo con una gestión de Hilos, de manera que el programa continúe escuchando ininterrumpidamente el puerto especificado para establecer comunicación, enviar o recibir mensajes sin estar congelado en el proceso. Además es adecuado asignar el proceso a un hilo del procesador determinado para poder detenerlo en caso de que el mismo se colapse, la comunicación se interrumpa u ocurra algún error inesperado. De esta manera se garantiza liberar el puerto empleado, evitando conflictos con otras instancias del mismo u otro programa.

Con estas consideraciones se presenta la resolución de la siguiente manera, en ambos casos (cliente y servidor) se ha empleado el modelo MVC:

III-A. Creación del servidor

Para la instancia del servidor se especifica el siguiente código en el objeto servidor:

Listing 1: Modelo de servidor

```
1   public class Servidor {
2
3       private Socket socket;
4       private ServerSocket serverSocket;
5       private DataInputStream bufferEntrada;
6       private DataOutputStream bufferSalida;
7
8       public Servidor() {
9           bufferEntrada = null;
10          bufferSalida = null;
11      }
12
13      public DataInputStream getBufferEntrada() {
14          return bufferEntrada;
15      }
16
17      public DataOutputStream getBufferSalida() {
18          return bufferSalida;
19      }
20
21      public Socket getSocket() {
22          return socket;
23      }
24
25      public ServerSocket getServerSocket() {
26          return serverSocket;
27      }
28
29      public void setBufferEntrada(DataInputStream bufferEntrada) {
30          this.bufferEntrada = bufferEntrada;
31      }
32
33      public void setBufferSalida(DataOutputStream bufferSalida) {
34          this.bufferSalida = bufferSalida;
35      }
36
37      public void setServerSocket(ServerSocket serverSocket) {
38          this.serverSocket = serverSocket;
```

```
39      }
40
41      public void setSocket(Socket socket) {
42          this.socket = socket;
43      }
44
45  }
```

Notese que los elementos de buffer van a permitir el intercambio de mensajes, pues la idea de resolución es intercambiar los datos que se almacenen en el buffer continuamente, separandolos por entrada y salida para evitar confusión al momento de presentarlos al usuario.

En el controlador, por otra parte, se ha implementado esta lógica descrita a lo largo del informe.

Listing 2: Controlador del servidor

```
1  public class Controlador {
2      private Servidor servidor;
3      private Ventana v;
4      private final String terminarPrograma = "terminar()";
5
6      public Controlador(Ventana v) {
7          servidor = new Servidor();
8          this.v = v;
9      }
10
11     public void iniciarConexion(int puerto) {
12         try {
13             servidor.setServerSocket(new ServerSocket(puerto));
14             v.mostrarAlerta("Esperando conexi n del puerto: " + puerto + " \
                   nPresione OK para continuar:)");
15             servidor.setSocket(servidor.getServerSocket().accept());
16             v.mostrarAlerta("Conexi n establecida con: " + servidor.getSocket().
                   getInetAddress().getHostName());
17             v.habilitarChat();
18         } catch (Exception e) {
19             v.mostrarAlerta("Error al conectar: \n" + e.getMessage());
20             System.exit(0);
21         }
22     }
23
24     public void establecerFlujo() {
25         try {
26             servidor.setBufferEntrada(new DataInputStream(servidor.getSocket().
                   getInputStream()));
27             servidor.setBufferSalida(new DataOutputStream(servidor.getSocket().
                   getOutputStream()));
28             servidor.getBufferSalida().flush();
29         } catch (Exception e) {
30             v.mostrarAlerta("Error al establecer un flujo de datos: \n" + e.
                   getMessage());
31         }
32     }
33
34     public void recibirDatos() {
35         String s = "";
36         try {
37             do {
38                 s = (String) servidor.getBufferEntrada().readUTF();
39                 v.mostrarMensaje("\n[Remoto]: " + s);
40                 // v.mostrarMensaje("\n[ T  ]: ");
41             } while (!s.equals(this.terminarPrograma));
42         } catch (Exception e) {
43             terminarConexion();
44         }
45     }
46
47     public void enviar(String s) {
48         try {
49             servidor.getBufferSalida().writeUTF(s);
50             servidor.getBufferSalida().flush();
51         } catch (Exception e) {
52             v.mostrarAlerta("Error al enviar mensaje\n\n" + e.getMessage());
53         }
54     }
```

```
55
56      public void escribirDatos(String s) {
57          v.mostrarMensaje("\n[ T  ]: " + s);
58          enviar(s);
59      }
60
61      public void terminarConexion() {
62          try {
63              servidor.getBufferEntrada().close();
64              servidor.getBufferSalida().close();
65              servidor.getSocket().close();
66          } catch (Exception e) {
67              v.mostrarAlerta("No se pudo cerrar la conexion: \n\n" + e.getMessage());
68          } finally {
69              v.mostrarAlerta("Se ha finalizado la conexi n de forma exitosa");
70          }
71
72      }
73
74      public void ejecutarConexion(int puerto) {
75          Thread h1 = new Thread(new Runnable() {
76              @Override
77              public void run() {
78                  while (true) {
79                      try {
80                          iniciarConexion(puerto);
81                          establecerFlujo();
82                          recibirDatos();
83                      } finally {
84                          terminarConexion();
85                      }
86                  }
87              }
88          });
89          h1.start();
90      }
91  }
```

Finalizando con la descripción del código base del servidor podemos observar que existe una gestión con un único hilo. Esto es debido a que únicamente se mantendrá una comunicación con un solo cliente a la vez mediante el localhost (dirección 127.0.0.1) ahorrando procesamiento de una mayor cantidad de hilos que a largo plazo no serán empleados por la dimensión del enunciado propuesto.

III-B. Creación del cliente

En el caso del cliente se puede notar que en el modelo solo se especifica el atributo de Socket, esto es debido a que el ServerSocket será escuchado desde el puerto indicado por el servidor y la comunicación iniciará una vez este requisito sea cumplido.

Listing 3: Modelo de Cliente

```
1  public class Cliente {
2      private Socket socket;
3      private DataInputStream bufferEntrada;
4      private DataOutputStream bufferSalida;
5
6      public Cliente() {
7          bufferEntrada = null;
8          bufferSalida = null;
9      }
10
11     public DataInputStream getBufferEntrada() {
12         return bufferEntrada;
13     }
14
15     public DataOutputStream getBufferSalida() {
16         return bufferSalida;
17     }
18
19     public Socket getSocket() {
20         return socket;
```

```
21      }
22
23      public void setBufferEntrada(DataInputStream bufferEntrada) {
24          this.bufferEntrada = bufferEntrada;
25      }
26
27      public void setBufferSalida(DataOutputStream bufferSalida) {
28          this.bufferSalida = bufferSalida;
29      }
30
31      public void setSocket(Socket socket) {
32          this.socket = socket;
33      }
34  }
```

En el controlador del cliente observamos los mismos métodos que en el lado del servidor, únicamente variando al momento de iniciar la comunicación por lo anteriormente mencionado.

Listing 4: Controlador del cliente

```
1  public class Controlador {
2      private Cliente cliente;
3      private Ventana v;
4      private final String terminarPrograma = "terminar()";
5
6      public Controlador(Ventana v) {
7          cliente = new Cliente();
8          this.v = v;
9      }
10
11     public void iniciarConexion(int puerto) {
12         try {
13             cliente.setSocket(new Socket("Localhost", puerto));
14             v.mostrarAlerta("Conexi n establecida con: " + cliente.getSocket().
                   getInetAddress().getHostName());
15             v.habilitarChat();
16         } catch (Exception e) {
17             v.mostrarAlerta("Error al conectar: \n" + e.getMessage());
18             System.exit(0);
19         }
20
21     }
22
23     public void establecerFlujo() {
24         try {
25             cliente.setBufferEntrada(new DataInputStream(cliente.getSocket().
                   getInputStream()));
26             cliente.setBufferSalida(new DataOutputStream(cliente.getSocket().
                   getOutputStream()));
27             cliente.getBufferSalida().flush();
28         } catch (Exception e) {
29             v.mostrarAlerta("Error al establecer un flujo de datos: \n" + e.
                   getMessage());
30         }
31     }
32
33     public void enviar(String s) {
34         try {
35             cliente.getBufferSalida().writeUTF(s);
36             cliente.getBufferSalida().flush();
37         } catch (Exception e) {
38             v.mostrarAlerta("Error al enviar mensaje\n\n" + e.getMessage());
39         }
40     }
41
42     public void terminarConexion() {
43         try {
44             cliente.getBufferEntrada().close();
45             cliente.getBufferSalida().close();
46             cliente.getSocket().close();
47         } catch (Exception e) {
48             v.mostrarAlerta("No se pudo cerrar la conexion: \n\n" + e.getMessage());
49         } finally {
50             v.mostrarAlerta("Se ha finalizado la conexi n de forma exitosa");
51         }
```

```
52      }
53
54      public void recibirDatos() {
55          String s = "";
56          try {
57              do {
58                  s = (String) cliente.getBufferEntrada().readUTF();
59                  v.mostrarMensaje("\n[Remoto]: " + s);
60                  // v.mostrarMensaje("\n[ T  ]: ");
61              } while (!s.equals(this.terminarPrograma));
62          } catch (Exception e) {
63              terminarConexion();
64          }
65      }
66
67      public void escribirDatos(String s) {
68          v.mostrarMensaje("\n[ T  ]: " + s);
69          enviar(s);
70      }
71
72      public void ejecutarConexion(int puerto) {
73          Thread h1 = new Thread(new Runnable() {
74              @Override
75              public void run() {
76                  while (true) {
77                      try {
78                          iniciarConexion(puerto);
79                          establecerFlujo();
80                          recibirDatos();
81                      } finally {
82                          terminarConexion();
83                      }
84                  }
85              }
86          });
87          h1.start();
88      }
89  }
```

Las ventanas en cada caso se visualizan de la siguiente manera:

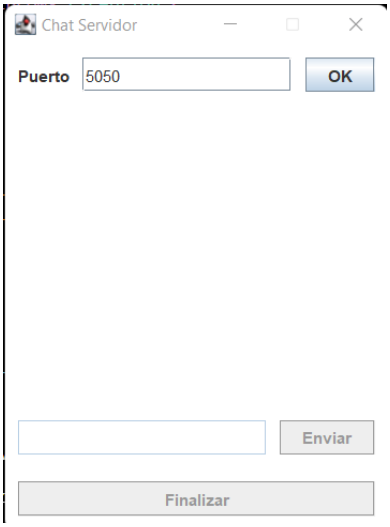

Figura 1: Ventana del chat del lado del servidor

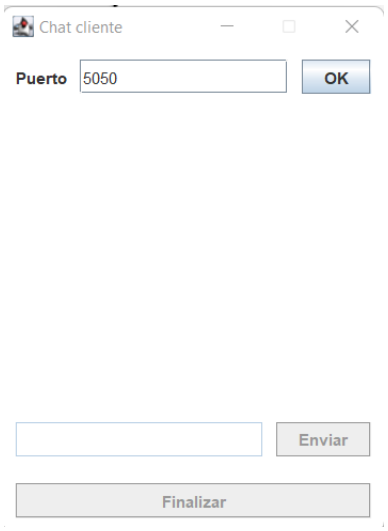

Figura 2: Ventana del chat del lado del cliente

IV. Conclusiones

Con base a lo investigado en esta práctica se pudo poner en marcha un chat haciendo uso de los sockets bajo el esquema de comunicación TCP de una manera eficiente. La implementación de lo aprendido en ciclos anteriores permitió establecer la necesidad de un hilo de control para mantener el proceso resguardado y que un malfuncionamiento no afecte al resto del programa, por tanto se establece como conclusión la importancia de analizar correctamente el esquema de comunicación adecuado para resolver cada problemática, sus pros y contras así como su costo a nivel físico y de código de implementación. El video del funcionamiento se ha adjuntado en la entrega de esta investigación.

REFERENCIAS

[1] L. Kalita, "Socket programming," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 3, pp. 4802–4807, 2014.

[2] Oracle, "Class socket," digital, https://docs.oracle.com/javase/7/docs/api/java/net/Socket.html, 2020.