



UNIVERSIDAD DE LAS FUERZAS ARMADAS

COMPUTACIÓN PARALELA

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Programación Concurrente: Sockets

Estudiantes:

Josué Merino, Angelo Sánchez, Justin Villarroel

Docente:

Ing. Carlos Andrés Pillajo Bolagay

8 de junio de 2024

1. Objetivos

1.1. Objetivo General

Realizar la implementación de una arquitectura Cliente – Servidor usando Sockets en Java.

1.2. Objetivos Específicos

- Implementar un servidor en Java que pueda aceptar múltiples conexiones de clientes simultáneamente, manejando las solicitudes de manera eficiente y segura mediante el uso de hilos
- Crear un cliente en Java que pueda conectarse al servidor, enviar y recibir mensajes de manera asíncrona, y manejar errores de comunicación de manera adecuada.

2. Resultados de Aprendizaje

1. **Análisis de ingeniería.** La capacidad de identificar, formular y resolver problemas de ingeniería en su especialidad; elegir y aplicar de forma adecuada métodos analíticos, de cálculo y experimentales ya establecidos; reconocer la importancia de las restricciones sociales, de salud y de seguridad, ambientales, económicas e industriales.
2. **Proyectos de ingeniería.** Capacidad para proyectar, diseñar y desarrollar productos complejos (piezas, componentes, productos acabados, etc.) procesos y sistemas de su especialidad, que cumplen con los requisitos establecidos, incluyendo tener conciencia de los aspectos sociales, de salud y seguridad, ambientales, económicos e industriales; así como seleccionar y aplicar métodos de proyectos apropiados.
3. **Aplicación práctica de la ingeniería.** Comprensión de las técnicas aplicables y métodos de análisis, proyecto e investigación y sus lineamientos en el ámbito de su especialidad.
4. **Comunicación y trabajo en equipo.** Capacidad para comunicar eficazmente información, ideas, problemas y soluciones en el ámbito de ingeniería y con la sociedad en general.

3. Herramientas Utilizadas

- Java

- Netbeans IDE
- Laptop
- Internet

4. Desarrollo

4.1. Implementar la clase Socket con la siguiente codificación

```
1 package servidor;
2 import java.io.DataOutputStream; import java.io.IOException; import
  java.net.ServerSocket; import java.net.Socket;
3
4 public class Servidor {
5
6     private int puerto;
7
8     public Servidor(int puerto) {
9
10        try {
11            ServerSocket servidor = new ServerSocket(puerto);
12            System.out.println("SERVER INICIADO - Esperando conexiones de
              clientes ...");
13
14            for (int i = 1; i <= 3; i++) {
15                Socket cliente = servidor.accept();
16                System.out.println("Se conecto el cliente " + i);
17                DataOutputStream salida = new
                  DataOutputStream(cliente.getOutputStream());
18                salida.writeUTF("Hola cliente " + i);
19                salida.close();
20                cliente.close();
21            }
22
23            servidor.close();
24            System.out.println("SERVER TERMINADO");
25
26        } catch (IOException e) {
27            e.printStackTrace();
28        }
29    }
30
31    public static void main(String[] args) {
32        new Servidor(10000);
33    }
34 }
```

El código implementa un servidor básico en Java que escucha conexiones en el puerto 10000. Utiliza `ServerSocket` para esperar a que hasta tres clientes se conecten. Cuando un cliente se conecta, el servidor acepta la conexión, envía un mensaje de saludo personalizado utilizando `DataOutputStream`, y luego cierra la conexión con el cliente. Este proceso se repite para tres clientes. Finalmente, el servidor cierra el `ServerSocket` y muestra un mensaje indicando que ha terminado. El flujo del programa se inicia en el método `main`, que crea una instancia del servidor con el puerto especificado.

4.2. Implementar la clase Cliente utilizando la siguiente codificación

```
1 package cliente;
2
3 import java.io.DataInputStream;
4 import java.io.IOException;
5 import java.net.Socket;
6
7 public class Cliente {
8     private int puerto;
9     private String ip;
10
11     public Cliente(String ip, int puerto) {
12         this.ip = ip;
13         this.puerto = puerto;
14         try {
15             Socket cliente = new Socket(ip, puerto);
16             DataInputStream entrada = new
17                 DataInputStream(cliente.getInputStream());
18             System.out.println(entrada.readUTF());
19             entrada.close();
20             cliente.close();
21         } catch (IOException e) {
22             e.printStackTrace();
23         }
24     }
25
26     public static void main(String[] args) {
27         new Cliente("localhost", 10000);
28     }
29 }
```

Cliente de socket que se conecta a un servidor en una dirección IP y puerto especificados. La clase `SocketClient` tiene un constructor que inicializa la IP y el puerto del servidor. El método `startClient` establece una conexión de socket con el servidor, envía una solicitud y luego recibe y muestra la respuesta del servidor. En el método `main`, el

Figura 1: Caption

programa solicita al usuario que ingrese una opción (descuentos, productos o salir) a través de la consola y, según la entrada, envía una solicitud correspondiente al servidor utilizando el método `startClient`. El bucle continúa hasta que el usuario ingrese "salir", cerrando la conexión y terminando el programa.

Una vez codificadas las clases anteriores, ejecutar el servidor y, de forma seguida, ejecutar el cliente. Capturas de pantalla:

5. Análisis

A continuación, se presenta el diagrama de conexión propuesto:

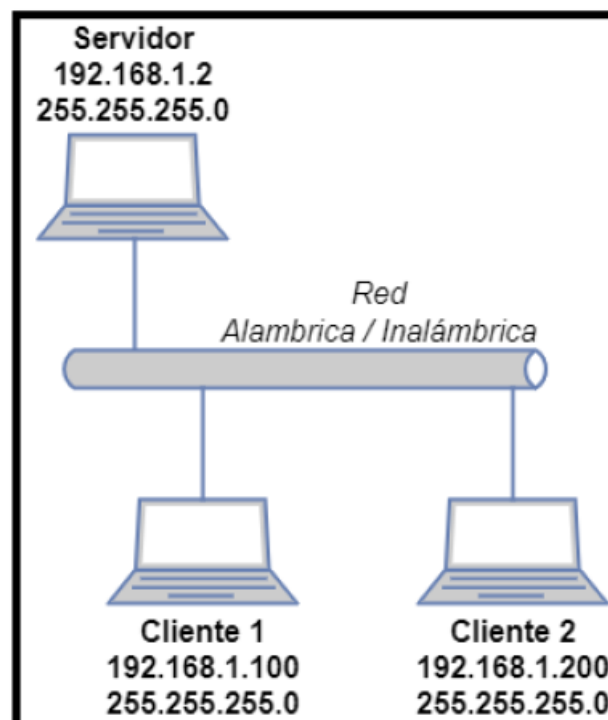


Figura 2: Conexión Propuesta

El código de la clase `Server` para el desarrollo del laboratorio es el siguiente:

```
1 package Server;  
2 import java.io.DataInputStream;  
3 import java.io.DataOutputStream;
```

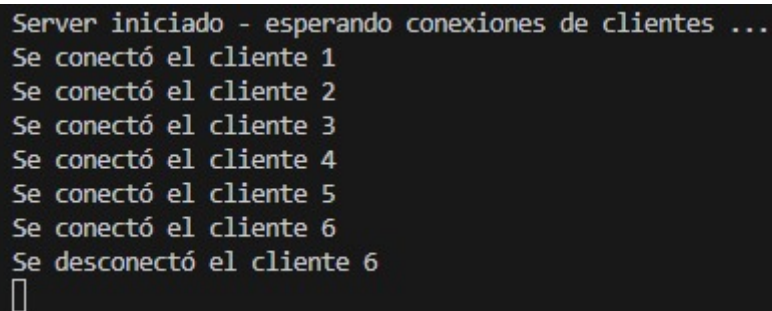
```
4 import java.io.IOException;
5 import java.net.ServerSocket;
6 import java.net.Socket;
7 import java.util.Random;
8 import java.util.concurrent.atomic.AtomicBoolean;
9
10 public class Server{
11
12     // Atributos
13     private AtomicBoolean running = new AtomicBoolean(true);
14     private ServerSocket servidor;
15
16     public Server(int puerto) {
17         this.puerto = puerto;
18     }
19
20     private int puerto;
21
22     // Descuentos y productos aleatorios
23     private static final String[] DESCUENTOS = {
24         "10% de descuento en productos de la marca Apple",
25         "20% de descuento en productos de la marca Samsung",
26         "25% de descuento en productos de la marca Xiaomi",
27         "30% de descuento en productos de la marca Huawei",
28         "40% de descuento en productos de la marca Sony",
29         "50% de descuento en productos de la marca LG",
30         "50% de descuento en todos los productos",
31         "15% de descuento en proximas compras",
32         "30% de descuento en accesorios tecnologicos",
33         "15% de descuento en productos seleccionados"
34     };
35     private static final String[] PRODUCTOS = {
36         "Memorias Ram",
37         "Celulares",
38         "Tablet",
39         "Reloj Inteligente",
40         "Camaras", "Audifonos",
41         "Teclados Mecanicos",
42         "Mouse Gamer",
43         "Cargadores",
44         "Baterias Externas"
45     };
46
47     // Metodo para iniciar el servidor
48     public void startServer() {
49         try {
50             servidor = new ServerSocket(puerto);
51             System.out.println("Server iniciado - esperando
                    conexiones de clientes ...");
```

```
52
53         int i = 1;
54         while (running.get()) {
55             try {
56                 Socket cliente = servidor.accept();
57                 System.out.println("Se conect el cliente " + i);
58                 new Thread(new ClientHandler(cliente, i)).start();
59                 i++;
60             } catch (IOException e) {
61                 if (!running.get()) {
62                     System.out.println("Servidor detenido.");
63                 } else {
64                     e.printStackTrace();
65                 }
66             }
67         }
68     } catch (IOException e) {
69         e.printStackTrace();
70     } finally {
71         if (servidor != null && !servidor.isClosed()) {
72             try {
73                 servidor.close();
74                 System.out.println("Server cerrado.");
75             } catch (IOException e) {
76                 e.printStackTrace();
77             }
78         }
79     }
80 }
81
82 // M todo main para iniciar el servidor
83 public static void main(String[] args) {
84     new Server(10000).startServer();
85 }
86
87 // M todos para obtener descuentos y productos aleatorios
88 private static String getRandomDescuento() {
89     Random rand = new Random();
90     return DESCUENTOS[rand.nextInt(DESCUENTOS.length)];
91 }
92 private static String getRandomProducto() {
93     Random rand = new Random();
94     return PRODUCTOS[rand.nextInt(PRODUCTOS.length)];
95 }
96
97
98 // Clase interna para manejar las solicitudes de los clientes a
99 // travez de hilos
100 class ClientHandler implements Runnable {
```

```
100     private Socket cliente;
101     private int clientNumber;
102
103     public ClientHandler(Socket cliente, int clientNumber) {
104         this.cliente = cliente;
105         this.clientNumber = clientNumber;
106     }
107
108     @Override
109     public void run() {
110         try (DataInputStream entrada = new
111             DataInputStream(cliente.getInputStream());
112             DataOutputStream salida = new
113                 DataOutputStream(cliente.getOutputStream())) {
114
115             String solicitud = entrada.readUTF();
116
117             if (solicitud.equalsIgnoreCase("descuentos")) {
118                 salida.writeUTF(getRandomDescuento());
119
120             } else if (solicitud.equalsIgnoreCase("productos")) {
121                 salida.writeUTF(getRandomProducto());
122
123             } else if (solicitud.equalsIgnoreCase("salir")) {
124                 salida.writeUTF("Adios cliente " + clientNumber);
125                 System.out.println("Se desconect el cliente " +
126                     clientNumber);
127
128             } else {
129                 salida.writeUTF("Solicitud no reconocida");
130
131             }
132         } catch (IOException e) {
133             e.printStackTrace();
134
135         } finally {
136             try {
137                 cliente.close();
138             } catch (IOException e) {
139                 e.printStackTrace();
140             }
141         }
142     }
143 }
```

El código implementa un servidor multihilo en Java que escucha en el puerto 10000. El servidor puede manejar múltiples clientes simultáneamente utilizando hilos. Se crean

arrays con descuentos y productos que se envían aleatoriamente a los clientes según su solicitud. La clase `Server` tiene un atributo `running` para controlar si el servidor está activo y un `ServerSocket` para escuchar conexiones. El método `startServer` inicia el servidor y acepta conexiones en un bucle mientras `running` sea verdadero. Cada cliente se maneja en un hilo separado utilizando la clase interna `ClientHandler`, que procesa las solicitudes de los clientes: "descuentos" para enviar un descuento aleatorio, "productos" para enviar un producto aleatorio, y "salir" para terminar la conexión. En el método `main`, se crea una instancia del servidor y se llama a `startServer` para iniciar el servidor.



```
Server iniciado - esperando conexiones de clientes ...
Se conectó el cliente 1
Se conectó el cliente 2
Se conectó el cliente 3
Se conectó el cliente 4
Se conectó el cliente 5
Se conectó el cliente 6
Se desconectó el cliente 6
[]
```

Figura 3: Servidor

El código del servidor editado para el taller:

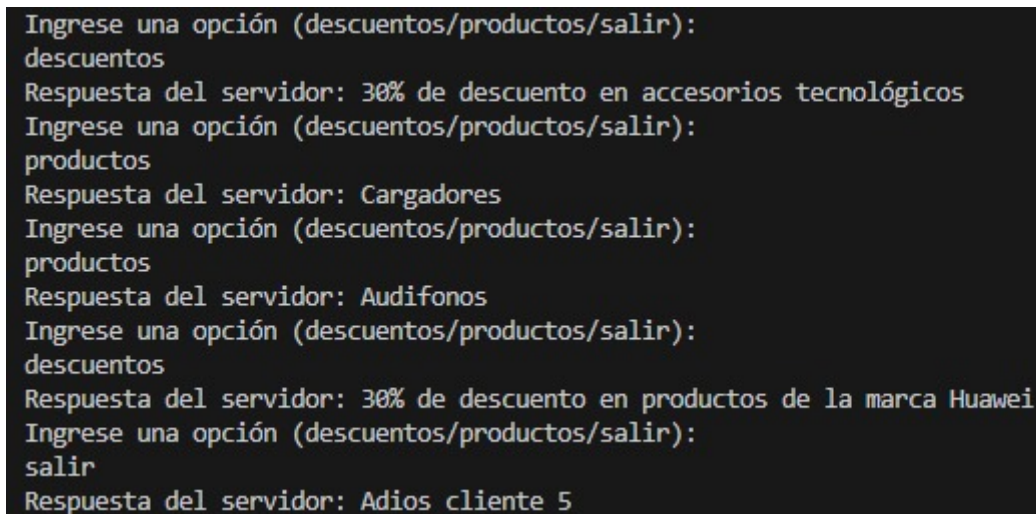
```
1 import java.io.DataInputStream;
2 import java.io.DataOutputStream;
3 import java.net.Socket;
4 import java.util.Scanner;
5
6
7 public class SocketClient {
8
9     private int puerto;
10    private String ip;
11
12    public SocketClient(String ip, int puerto){
13        this.ip = ip;
14        this.puerto = puerto;
15    }
16
17    public void startClient(String solicitud) {
18        try {
19            Socket cliente = new Socket(ip, puerto);
20            DataOutputStream salida = new
                DataOutputStream(cliente.getOutputStream());
```

```
21      DataInputStream entrada = new
22          DataInputStream(cliente.getInputStream());
23
24      // Enviar la solicitud al servidor
25      salida.writeUTF(solicitud);
26
27      // Leer la respuesta del servidor
28      System.out.println("Respuesta del servidor: " +
29          entrada.readUTF());
30
31      salida.close();
32      entrada.close();
33      cliente.close();
34  } catch (Exception e) {
35      e.printStackTrace();
36  }
37
38  }
39
40  public static void main(String[] args) {
41      Scanner scr = new Scanner(System.in);
42      String input = "";
43      SocketClient cliente = new SocketClient("10.40.32.39", 10000);
44
45      do {
46          System.out.println("Ingrese una opcion
47              (descuentos/productos/salir): ");
48          input = scr.next();
49
50          if (input.equals("descuentos")) {
51              cliente.startClient("descuentos");
52          } else if (input.equals("productos")) {
53              cliente.startClient("productos");
54          } else if (input.equals("salir")) {
55              cliente.startClient("salir");
56          } else if (!input.equals("salir")) {
57              System.out.println("Opcion no valida. Intente de
58                  nuevo.");
59          }
60      } while (!input.equals("salir"));
61
62      scr.close();
63  }
64 }
```

El código implementa un cliente de socket en Java que se conecta a un servidor en una dirección IP y puerto específicos, envía una solicitud y recibe una respuesta. La clase `SocketClient` contiene dos atributos: `ip` y `puerto`, que se inicializan en el constructor. El método `startClient` maneja la comunicación con el servidor. Este método crea un

socket utilizando la IP y el puerto del servidor, luego establece un `DataOutputStream` para enviar la solicitud al servidor y un `DataInputStream` para recibir la respuesta. La solicitud se envía utilizando el método `writeUTF`, y la respuesta del servidor se lee y se imprime en la consola utilizando el método `readUTF`. Finalmente, el método cierra el flujo de salida, el flujo de entrada y el socket para liberar los recursos.

En el método `main`, se crea una instancia de `SocketClient` con la IP del servidor y el puerto 10000. Utiliza un `Scanner` para leer la entrada del usuario desde la consola. En un bucle `do-while`, se solicita al usuario que ingrese una opción (descuentos, productos, salir). Dependiendo de la opción ingresada, el método `startClient` se llama con la solicitud correspondiente. Si el usuario ingresa "descuentos" o "productos", el cliente envía la solicitud al servidor y muestra la respuesta recibida. Si el usuario ingresa "salir", el cliente envía la solicitud de salida y finaliza la conexión. Si se ingresa una opción no válida, se muestra un mensaje de error y se solicita una nueva entrada. El bucle continúa hasta que el usuario ingresa "salir", momento en el cual se cierra el `Scanner`. Este cliente de socket permite interactuar con un servidor de manera sencilla, enviando solicitudes y recibiendo respuestas a través de una interfaz de línea de comandos.



```
Ingrese una opción (descuentos/productos/salir):
descuentos
Respuesta del servidor: 30% de descuento en accesorios tecnológicos
Ingresa una opción (descuentos/productos/salir):
productos
Respuesta del servidor: Cargadores
Ingresa una opción (descuentos/productos/salir):
productos
Respuesta del servidor: Audifonos
Ingresa una opción (descuentos/productos/salir):
descuentos
Respuesta del servidor: 30% de descuento en productos de la marca Huawei
Ingresa una opción (descuentos/productos/salir):
salir
Respuesta del servidor: Adios cliente 5
```

Figura 4: Cliente

6. Conclusiones

La implementación de una arquitectura Cliente-Servidor usando Sockets en Java demostró ser un enfoque eficaz para manejar múltiples conexiones de clientes de manera simultánea y segura. El servidor fue capaz de aceptar conexiones de varios clientes utilizando hilos, lo que permitió manejar múltiples solicitudes concurrentemente sin

bloquear otras operaciones. La capacidad de enviar y recibir mensajes de manera eficiente se logró mediante el uso de `DataInputStream` y `DataOutputStream`, garantizando una comunicación clara y estructurada. Además, la implementación del cliente mostró que es posible establecer una conexión confiable con el servidor, enviar solicitudes específicas y recibir respuestas adecuadas, manejando cualquier error de comunicación que pudiera surgir. Este proyecto ilustra cómo los conceptos fundamentales de la programación de redes y la concurrencia pueden aplicarse para construir sistemas robustos y escalables.

7. Recomendaciones

- **Seguridad:** Integrar protocolos de seguridad como SSL/TLS para cifrar la comunicación entre el cliente y el servidor, asegurando que los datos transmitidos no puedan ser interceptados ni modificados por terceros.
- **Optimización de Rendimiento:** Utilizar un pool de hilos en el servidor para manejar conexiones, lo que puede mejorar el rendimiento y la escalabilidad al limitar la creación de nuevos hilos y reutilizar los existentes.
- **Gestión de Errores:** Implementar mecanismos más sofisticados de manejo de errores, incluyendo reconexión automática del cliente en caso de fallos y registro de errores detallados para facilitar la depuración.
- **Interfaz de Usuario:** Desarrollar una interfaz gráfica de usuario (GUI) para el cliente, proporcionando una experiencia más amigable y accesible para el usuario final.