Entrega 1: Calculadora distribuida

Cómputo Distribuido

Profesor: Carlos Pérez Leguízamo

Sebastián Godínez Borja

Esteban Viniegra Pérez Olagaray

ID: 0235320

Fecha de entrega: 18 de septiembre del 2023

Introducción

El cómputo distribuido ha revolucionado la manera en que se procesan y gestionan datos, permitiendo dividir tareas y procesos en múltiples componentes y máquinas que trabajan de forma simultánea. Esta calculadora distribuida es una representación simplificada de cómo se pueden utilizar estos principios para operaciones básicas. A continuación, se describen en detalle las tecnologías y códigos que componen este proyecto.

Desarrollo

El proyecto se divide en 3 partes:

- **Cliente (Calculadora con interfaz gráfica)**

  Diseño y arquitectura:

  La interfaz de cliente está diseñada para ser intuitiva. La elección de JavaFX como tecnología principal es para aprovechar su capacidad de proporcionar interfaces ricas y dinámicas, además de ser multiplataforma.

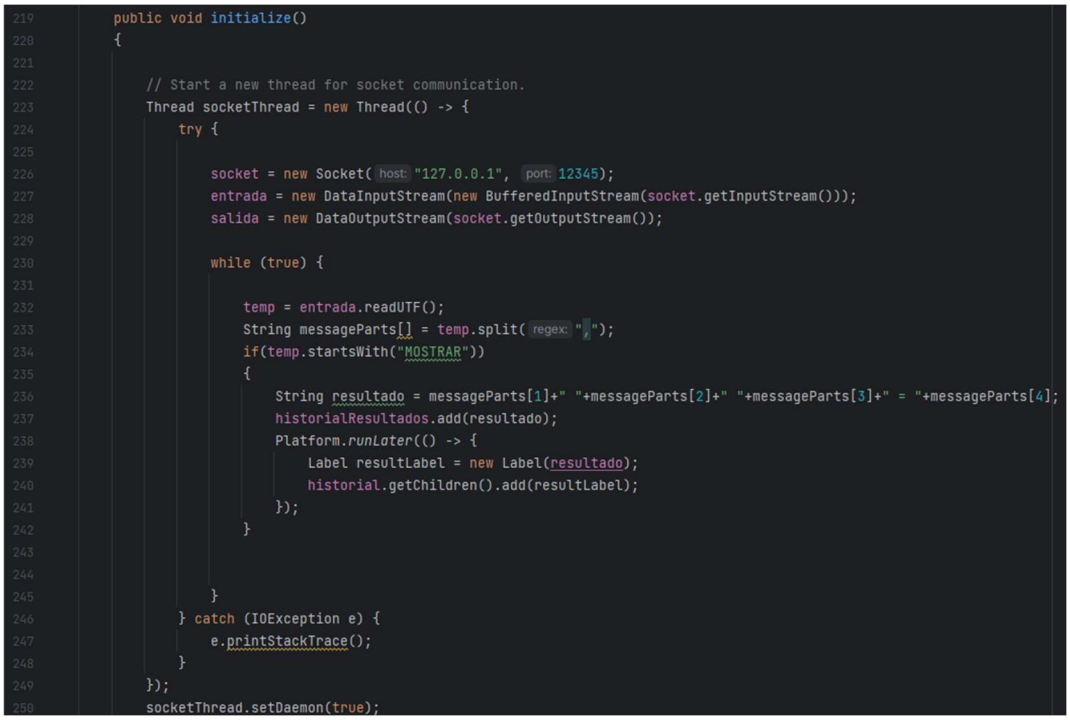
Figura 1: Interfaz gráfica del cliente, es una calculadora sencilla capaz de realizar operaciones y comunicarse con el middelware

Funcionalidad:

Cada vez que un usuario ingresa un número o selecciona una operación, esta entrada se almacena temporalmente en variables. Al presionar el botón de igual, se inicia la comunicación con el middleware. El uso de sockets permite que esta comunicación sea en tiempo real.

```
199     @FXML
200     void buttonIgual() {
201         String paquetePorMandar = "RESOLVER"+","+n1+","+operacion+","+n2;
202
203         n1 = "";
204         n2 = "";
205         operacion = "";
206         pantalla.setText("");
207
208         try
209         {
210             salida.writeUTF(paquetePorMandar);
211         } catch (IOException error)
212         {
213             System.out.println(error);
214         }
215
216     }
```

Figura 2: Funcionalidad del botón igual para mandar la operación



```
public void initialize()
{

    // Start a new thread for socket communication.
    Thread socketThread = new Thread(() -> {
        try {

            socket = new Socket( host: "127.0.0.1",  port: 12345);
            entrada = new DataInputStream(new BufferedInputStream(socket.getInputStream()));
            salida = new DataOutputStream(socket.getOutputStream());

            while (true) {

                temp = entrada.readUTF();
                String messageParts[] = temp.split( regex: ",");
                if(temp.startsWith("MOSTRAR"))
                {
                    String resultado = messageParts[1]+" "+messageParts[2]+" "+messageParts[3]+" = "+messageParts[4];
                    historialResultados.add(resultado);
                    Platform.runLater(() -> {
                        Label resultLabel = new Label(resultado);
                        historial.getChildren().add(resultLabel);
                    });
                }


            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
    socketThread.setDaemon(true);
```

Figura 3: Inicialización de la comunicación entre MOM y cliente utilizando websockets

El principal desafío aquí es garantizar que los datos se envíen de manera coherente y segura al middleware, y que cualquier interrupción o error en la comunicación sea manejado adecuadamente.

- **Middleware o MOM (Message-Oriented Middleware)**

Diseño y arquitectura:

El middleware actúa como puente y buffer. Está diseñado para manejar múltiples conexiones simultáneamente, evitando cuellos de botella y asegurando que los mensajes sean entregados adecuadamente.

Funcionalidad:

Al recibir un mensaje del cliente, el middleware puede hacer una de las siguientes acciones:

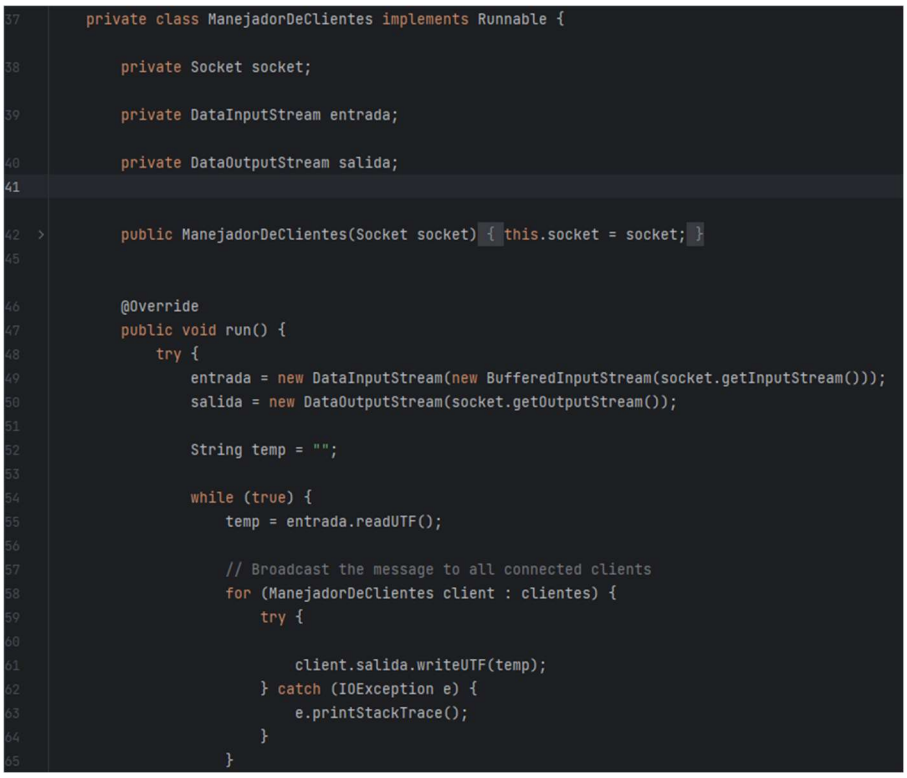
Procesar el mensaje si es algo simple (por ejemplo, una solicitud de ping para comprobar la conexión).

Reenviar el mensaje al servidor si requiere cálculos o procesamiento adicional.

El middleware utiliza programación basada en hilos para manejar múltiples conexiones. Cuando se establece una conexión, se crea un nuevo hilo dedicado a ese cliente en particular.

```
6  public class MOM {
7      ServerSocket servidor;
8      List<ManejadorDeClientes> clientes;
9
10     public MOM(int port) {
11         clientes = new ArrayList<>();
12
13         try {
14             servidor = new ServerSocket(port);
15             System.out.println("Servidor Corriendo");
16             while (true) {
17
18
19                 Socket socket = servidor.accept();
20                 System.out.println("Nuevo Cliente aceptado");
21
22                 ManejadorDeClientes clientHandler = new ManejadorDeClientes(socket);
23                 clientes.add(clientHandler);
24
25                 Thread thread = new Thread(clientHandler);
26                 thread.start();
27             }
28         } catch (IOException i) {
29             System.out.println(i);
30         }
31     }
```

Figura 4: Inicialización de socket para el servidor y creación de hilos y sockets para cada cliente, para poder ser manejados de manera simultánea



```
37    private class ManejadorDeClientes implements Runnable {
38        private Socket socket;
39        private DataInputStream entrada;
40        private DataOutputStream salida;
41
42        public ManejadorDeClientes(Socket socket) { this.socket = socket; }
45
46        @Override
47        public void run() {
48            try {
49                entrada = new DataInputStream(new BufferedInputStream(socket.getInputStream()));
50                salida = new DataOutputStream(socket.getOutputStream());
51
52                String temp = "";
53
54                while (true) {
55                    temp = entrada.readUTF();
56
57                    // Broadcast the message to all connected clients
58                    for (ManejadorDeClientes client : clientes) {
59                        try {
60
61                            client.salida.writeUTF(temp);
62                        } catch (IOException e) {
63                            e.printStackTrace();
64                        }
65                    }
```

Figura 5: Manejo de clientes y comunicación hacia el servidor

- Servidor

Diseño:

El servidor es, en esencia, el motor de cálculo. Está diseñado para ser eficiente y rápido, garantizando tiempos de respuesta mínimos.

Funcionalidad:

Cuando el servidor recibe una solicitud de cálculo del middleware, primero descompone y analiza el mensaje para determinar qué operación debe realizar. Posteriormente, ejecuta la operación y devuelve el resultado al middleware.

Todo el manejo de la operación y la lógica matemática está encapsulada aquí, lo que significa que si en el futuro se quisieran agregar más operaciones o funcionalidades, este sería el lugar para hacerlo.

```
private class Mensajes implements Runnable {
    @Override
    public void run() {
        try {
            DataInputStream entrada = new DataInputStream(socket.getInputStream());

            while (true) {
                String mensaje = entrada.readUTF();
                String[] paquete = mensaje.split( regex: ",");
                int result = 0;
                if(paquete[0].startsWith("RESOLVER"))
                {

                    if(Objects.equals(paquete[2], b: "+"))
                    {
                        result = Integer.parseInt(paquete[1]) + Integer.parseInt(paquete[3]);
                    }

                    if(Objects.equals(paquete[2], b: "-"))
                    {
                        result = Integer.parseInt(paquete[1]) - Integer.parseInt(paquete[3]);
                    }

                    if(Objects.equals(paquete[2], b: "*"))
                    {
                        result = Integer.parseInt(paquete[1])  * Integer.parseInt(paquete[3]);
                    }

                    if(Objects.equals(paquete[2], b: "/"))
                    {
                        result = Integer.parseInt(paquete[1]) / Integer.parseInt(paquete[3]);
                    }
```

Figura 6: Lógica matemática para cada operación

```java
public class Servidor {
    3 usages
    Socket socket;
    1 usage
    DataInputStream entrada;
    2 usages
    DataOutputStream salida;

    no usages
    int resultado;


    1 usage
    public Servidor(String serverName, int serverPort) {
        try {
            socket = new Socket(serverName, serverPort);

            entrada = new DataInputStream(System.in);
            salida = new DataOutputStream(socket.getOutputStream());


            Thread thread = new Thread(new Mensajes());
            thread.start();

        } catch (IOException error) {
            System.out.println(error);
        }
    }

    public static void main(String[] args) { new Servidor( serverName: "127.0.0.1",  serverPort: 12345); }
```

Figura 7: Inicialización del servidor, creando el socket de comunicación, distribución de datos utilizando hilos y definiendo el puerto por el cuál se va a comunicar

Concusión

Cada componente de este proyecto de calculadora distribuida tiene su propio conjunto de desafíos y responsabilidades:

- El cliente se centra en la interacción con el usuario.
- El middleware se encarga de la comunicación y el enrutamiento de mensajes.
- El servidor realiza el cálculo y procesamiento principal.

Hay que asegurar que estos tres componentes trabajen juntos armoniosamente y de manera eficiente es la clave del éxito del proyecto. Además, cada uno de ellos podría escalarse o mejorarse independientemente según las necesidades futuras. Por ejemplo, se podría migrar el servidor a un clúster de servidores para manejar un mayor volumen de cálculos, o se podría mejorar el cliente con una interfaz más avanzada.