RPG0018 - Por que não paralelizar

Jéssica Maria de Carvalho Matrícula: 202209187939

POLO JARDIM SÃO BERNARDO - SÃO PAULO - SP

Nível 5: Por que não paralelizar – 9003 – 3°

Endereço do Repositório GIT: https://github.com/Jessicac30/missao-5

Objetivo da Prática

1. Criar servidores Java com base em Sockets.
2. Criar clientes síncronos para servidores com base em Sockets.
3. Criar clientes assíncronos para servidores com base em Sockets.
4. Utilizar Threads para implementação de processos paralelos.
5. No final do exercício, o aluno terá criado um servidor Java baseado em Socket, com acesso ao banco de dados via JPA, além de utilizar os recursos nativos do Java para implementação de clientes síncronos e assíncronos. As Threads serão usadas tanto no servidor, para viabilizar múltiplos clientes paralelos, quanto no cliente, para implementar a resposta assíncrona.

1º Procedimento | Criando o Servidor e Cliente de Teste

CadastroClient.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Main.java to
edit this template
 */
package cadastroclient;

import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.PrintStream;
import java.net.Socket;
import java.util.List;
import java.util.Scanner;
import model.Produto;

/**
 *
 * @author jess
 */
public class CadastroClient {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)throws ClassNotFoundException,
IOException {
        Socket socket = new Socket("localhost", 4321);
        ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream in = new
ObjectInputStream(socket.getInputStream());

       // Login, passando usuário "op1"
        out.writeObject("op1");

        // Senha para o login usando "op1"
        out.writeObject("op1");

        // Lê resultado do login:
        System.out.println((String)in.readObject());
```

```
        // Lista produtos:
        out.writeObject("L");

        List<Produto> produtos = (List<Produto>) in.readObject();
        for (Produto produto : produtos) {
            System.out.println(produto.getNome());
        }

        out.close();
        in.close();
        socket.close();
    }

}
```

CadastroServer.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Main.java to
edit this template
 */
package cadastroserver;
import controller.ProdutoJpaController;
import controller.UsuarioJpaController;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;


/**
 *
 * @author jess
 */
public class CadastroServer {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws IOException{
```

```
        ServerSocket serverSocket = new ServerSocket(4321);
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("CadastroServerPU");
        ProdutoJpaController ctrl = new ProdutoJpaController(emf);
        UsuarioJpaController ctrlUsu = new UsuarioJpaController(emf);
        //Socket socket = serverSocket.accept();
        //System.out.println("Cliente Conectou");

        while (true) {
            // Aguarda um cliente se conectar e aceita a conexão (chamada
bloqueante)
            Socket clienteSocket = serverSocket.accept();
            System.out.println("Cliente conectado: " +
clienteSocket.getInetAddress());

            // CadastroThread V1:
            // CadastroThread thread = new CadastroThread(ctrl, ctrlUsu,
clienteSocket);

            // CadastroThread V2:
            CadastroThread thread = new CadastroThread(ctrl, ctrlUsu,
clienteSocket);

            thread.start();
            System.out.println("Aguardando nova conexão...");
        }

        }
    }
```

2º Procedimento | Servidor Completo e Cliente Assíncrono

CadastroClientv2.java

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Main.java to
edit this template
 */
package cadastroclient;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.PrintStream;
import java.net.Socket;
import java.util.List;
import java.util.Scanner;
import model.Produto;

/**
 *
 * @author jess
 */


public class CadastroClientv2 {

    private static ObjectOutputStream socketOut;
    private static ObjectInputStream socketIn;
    private static ThreadClient threadClient;


    /**
     * @param args the command line arguments
     */


    public static void main(String[] args)throws ClassNotFoundException,
IOException {
        Socket socket = new Socket("localhost", 4321);
        socketOut = new ObjectOutputStream(socket.getOutputStream());
        socketIn = new ObjectInputStream(socket.getInputStream());
```

```
        // Encapsula a leitura do teclado em um BufferedReader
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

        // Instancia a janela SaidaFrame para apresentação de mensagens
        SaidaFrame saidaFrame = new SaidaFrame();
        saidaFrame.setVisible(true);

        // Instancia a Thread para preenchimento assíncrono com a
passagem do canal de entrada do Socket
        threadClient = new ThreadClient(socketIn, saidaFrame.texto);
        threadClient.start();

        // Login, passando usuário "op1"
        socketOut.writeObject("op1");

        // Senha para o login usando "op1"
        socketOut.writeObject("op1");

        // Exibe Menu:
        Character commando = ' ';
        try {
            while (!commando.equals('X')) {
                System.out.println("Escolha uma opção:");
                System.out.println("L - Listar | X - Finalizar | E -
Entrada | S - Saída");

                // Lê a opção do teclado usando o reader e converte para
Character:
                commando = reader.readLine().charAt(0);

                processaComando(reader, commando);
            }
        } catch(Exception e) {
            e.printStackTrace();
        } finally {
            saidaFrame.dispose();
            socketOut.close();
            socketIn.close();
            socket.close();
            reader.close();
        }
    }

    static void processaComando(BufferedReader reader, Character
commando) throws IOException {
        // Define comando a ser enviado ao servidor:
        socketOut.writeChar(commando);
```

```
        socketOut.flush();

        switch (commando) {
            case 'L':
                // Comando é apenas enviado para o servidor.
                break;
            case 'S':
            case 'E':
                // Confirma envio do comando ao servidor:
                socketOut.flush();

                // Lê os dados do teclado:
                System.out.println("Digite o Id da pessoa:");
                int idPessoa = Integer.parseInt(reader.readLine());
                System.out.println("Digite o Id do produto:");
                int idProduto = Integer.parseInt(reader.readLine());
                System.out.println("Digite a quantidade:");
                int quantidade = Integer.parseInt(reader.readLine());
                System.out.println("Digite o valor unitário:");
                long valorUnitario = Long.parseLong(reader.readLine());

                // Envia os dados para o servidor:
                socketOut.writeInt(idPessoa);
                socketOut.flush();
                socketOut.writeInt(idProduto);
                socketOut.flush();
                socketOut.writeInt(quantidade);
                socketOut.flush();
                socketOut.writeLong(valorUnitario);
                socketOut.flush();
                break;
            case 'X':
                threadClient.cancela(); // Cancela a ThreadClient já que
o cliente está desconectando.
                break;
            default:
                System.out.println("Opção inválida!");
        }
    }

}
```

SaidaFrame.java

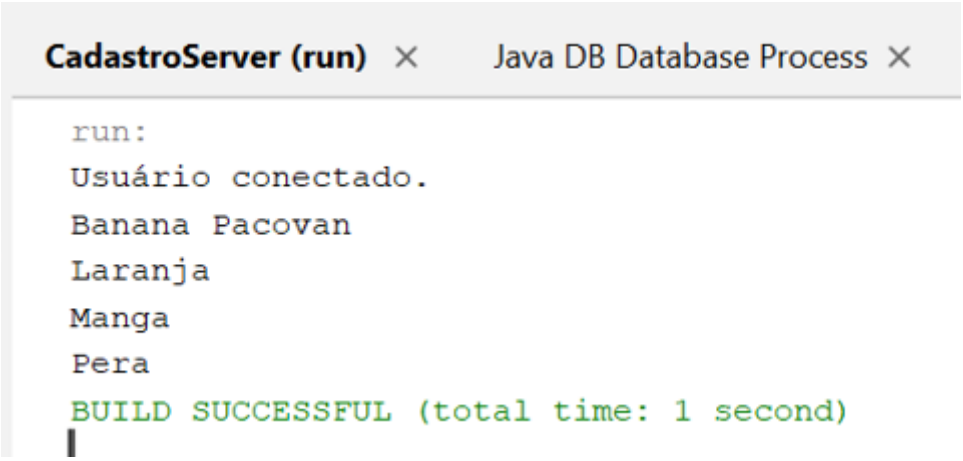
```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to
edit this template
 */
package cadastroclient;
import javax.swing.*;
/**
 *
 * @author jess
 */
public class SaidaFrame extends JDialog {
    public JTextArea texto;

    public SaidaFrame() {
        // Define as dimensões da janela
        setBounds(100, 100, 400, 300);

        // Define o status modal como false
        setModal(false);

        // Acrescenta o componente JTextArea na janela
        texto = new JTextArea(25, 40);
        texto.setEditable(false); // Bloqueia edição do campo de texto

        // Adiciona componente para rolagem
        JScrollPane scroll = new JScrollPane(texto);
        scroll.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTA
L_SCROLLBAR_NEVER); // Bloqueia rolagem horizontal
        add(scroll);
    }
}
```

TheadClient.java

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to
edit this template
 */
package cadastroclient;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.net.SocketException;
import java.util.List;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;
import model.Produto;

/**
 *
 * @author jess
 */
public class ThreadClient extends Thread {
    private ObjectInputStream entrada;
    private JTextArea textArea;
    private Boolean cancelada;

    public ThreadClient(ObjectInputStream entrada, JTextArea textArea) {
        this.entrada = entrada;
        this.textArea = textArea;
        this.cancelada = false;
    }

    @Override
    public void run() {
        while (!cancelada) {
            try {
                Object resposta = entrada.readObject();
                SwingUtilities.invokeLater(() -> {
                    processaResposta(resposta);
                });
            } catch (IOException | ClassNotFoundException e) {
                if (!cancelada) {
                    System.err.println(e);
                }
            }
        }
    }

    public void cancela() {
```

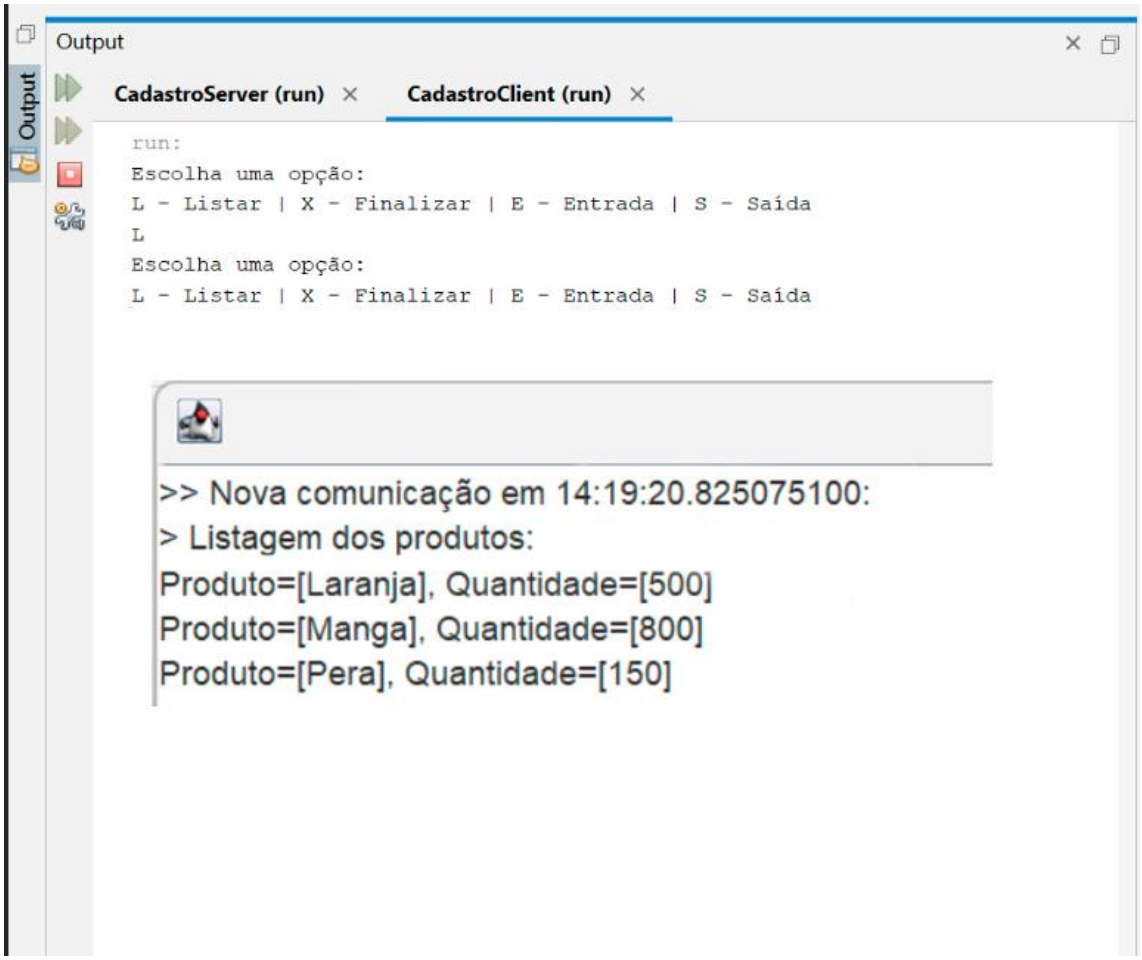
```
        cancelada = true;
    }

    private void processaResposta(Object resposta) {
        // Adiciona nova mensagem ao textArea contendo o horário atual:
        textArea.append(">> Nova comunicação em " +
java.time.LocalTime.now() + ":\n");

        if (resposta instanceof String) {
            textArea.append((String) resposta + "\n");
        } else if (resposta instanceof List<?>) {
            textArea.append("> Listagem dos produtos:\n");
            List<Produto> lista = (List<Produto>) resposta;
            for (Produto item : lista) {
                textArea.append("Produto=[" + item.getNome() + "],
Quantidade=["+ item.getQuantidade() + "]\n");
            }
        }
        textArea.append("\n");
        textArea.setCaretPosition(textArea.getDocument().getLength());
    }
}
```

Resultados


CadastroServer (run)
Java DB Database Process
run:
Usuário conectado.
Banana Pacovan
Laranja
Manga
Pera
BUILD SUCCESSFUL (total time: 1 second)

CadastroServer (run) × CadastroClient (run) ×

```
Escolha uma opção:
L - Listar | X - Finalizar | E - Entrada | S - Saída
L
Escolha uma opção:
L - Listar | X - Finalizar | E - Entrada | S - Saída
E
Digite o Id da pessoa:
1
Digite o Id do produto:
1
Digite a quantidade:
10
Digite o valor unitário:
5
Escolha uma opção:
L - Listar | X - Finalizar | E - Entrada | S - Saída
S
Digite o Id da pessoa:
1
Digite o Id do produto:
1
Digite a quantidade:
20
Digite o valor unitário:
5
Escolha uma opção:
L - Listar | X - Finalizar | E - Entrada | S - Saída
X
BUILD SUCCESSFUL (total time: 3 minutes 1 second)
```


Output
CadastroServer (run)
CadastroClient (run)
run:
Escolha uma opção:
L - Listar | X - Finalizar | E - Entrada | S - Saída
L
Escolha uma opção:
L - Listar | X - Finalizar | E - Entrada | S - Saída
>> Nova comunicação em 14:19:20.825075100:
> Listagem dos produtos:
Produto=[Laranja], Quantidade=[500]
Produto=[Manga], Quantidade=[800]
Produto=[Pera], Quantidade=[150]

Análise e Conclusão:

Como funcionam as classes Socket e ServerSocket?

- **Socket**: Representa um ponto de conexão (socket) para comunicação entre o cliente e o servidor. Quando um cliente quer se conectar a um servidor, ele cria um objeto **Socket**, especificando o endereço IP e a porta do servidor. Após a conexão ser estabelecida, o **Socket** é usado para enviar e receber dados.
- **ServerSocket**: É usado pelo servidor para ouvir conexões de entrada em uma porta específica. Quando um servidor inicia, ele cria um objeto **ServerSocket** e o associa a uma porta. O servidor então espera (ou "ouve") as solicitações de conexão dos clientes. Quando uma solicitação é recebida, o **ServerSocket** aceita a conexão e cria um **Socket** para se comunicar com o cliente.

Qual a importância das portas para a conexão com servidores?

As portas são essenciais para a comunicação em redes, funcionando como pontos de extremidade em um host. Cada serviço em um servidor é identificado por uma porta única, permitindo que vários serviços (como HTTP, FTP) funcionem simultaneamente em um único servidor. Quando um cliente se conecta a um servidor, ele deve especificar a porta correta para estabelecer a conexão com o serviço desejado.

Para que servem as classes de entrada e saída ObjectInputStream e ObjectOutputStream, e por que os objetos transmitidos devem ser serializáveis?

- Estas classes são usadas para ler e escrever objetos em streams. Elas são úteis para enviar e receber objetos através de uma conexão de rede.
- Os objetos transmitidos devem ser serializáveis, o que significa que eles devem implementar a interface Serializable. Isso é necessário porque a serialização converte um objeto em uma sequência de bytes que pode ser transmitida por uma rede e depois reconstruída no outro lado.

Por que, mesmo utilizando as classes de entidades JPA no cliente, foi possível garantir o isolamento do acesso ao banco de dados?

- Mesmo usando classes de entidades JPA no cliente, o isolamento do acesso ao banco de dados é mantido porque essas entidades são apenas representações dos dados. Elas não realizam operações de banco de dados diretamente. Em vez disso, elas são usadas em conjunto com um gerenciador de entidades (EntityManager) no lado do servidor, que gerencia o acesso ao banco de dados.
- Esse modelo permite que a lógica de negócios e o acesso ao banco de dados sejam mantidos no servidor, enquanto o cliente lida apenas com objetos de transferência de dados (DTOs), garantindo a segurança e a integridade dos dados.

Como as Threads podem ser utilizadas para o tratamento assíncrono das respostas enviadas pelo servidor?

As Threads podem ser utilizadas para o tratamento assíncrono das respostas enviadas pelo servidor de várias maneiras. Uma abordagem comum é criar uma Thread separada para lidar com cada solicitação de cliente. Isso permite que o servidor atenda a múltiplos clientes de forma simultânea, garantindo que as respostas sejam tratadas de maneira assíncrona. Cada Thread fica responsável por processar uma solicitação específica, permitindo que o servidor continue a receber e responder a outras solicitações enquanto uma Thread lida com o processamento de uma resposta. Isso é particularmente útil em sistemas onde é importante manter a capacidade de resposta e a eficiência, especialmente em ambientes de servidor web ou aplicativos de rede, onde várias solicitações podem chegar ao servidor simultaneamente. No entanto, é importante implementar mecanismos de sincronização e gerenciamento de recursos compartilhados adequadamente para evitar problemas de concorrência.

Para que serve o método invokeLater, da classe SwingUtilities?

O método invokeLater da classe SwingUtilities em Java é usado para agendar a execução de um pedaço de código (normalmente um Runnable) para ser executado de forma assíncrona na thread de despacho de eventos (Event Dispatch Thread - EDT) do Swing. A EDT é a thread principal responsável pela atualização da interface gráfica de um aplicativo Swing, e ela é sensível ao desempenho e à responsividade da interface do usuário.

Ao utilizar o invokeLater, você pode garantir que as operações que afetam a interface gráfica sejam realizadas na EDT, evitando assim problemas de concorrência e garantindo a consistência da interface do usuário. Isso é especialmente importante quando você precisa realizar atualizações na GUI a partir de outras threads, como threads de processamento em segundo plano, para evitar bloqueios e travamentos na interface do usuário.

Como os objetos são enviados e recebidos pelo Socket Java?

Em Java, para enviar e receber objetos por meio de sockets, você precisa seguir algumas etapas. Primeiro, certifique-se de que a classe do objeto que você deseja transmitir implemente a interface **Serializable**. Em seguida, abra um socket para a comunicação, usando **Socket** no lado do cliente e **ServerSocket** no lado do servidor. Obtenha fluxos de entrada e saída a partir do socket e utilize um **ObjectOutputStream** para enviar objetos e um **ObjectInputStream** para recebê-los. Basta chamar os métodos **writeObject(objeto)** para enviar e **readObject()** para receber. Não se esqueça de fechar os fluxos e o socket após a comunicação. Essa abordagem permite que objetos sejam serializados (convertidos em bytes) e transmitidos pela rede, tornando a comunicação entre processos ou máquinas possível em Java.

Compare a utilização de comportamento assíncrono ou síncrono nos clientes com Socket Java, ressaltando as características relacionadas ao bloqueio do processamento.

O comportamento síncrono em clientes com Socket Java bloqueia o processamento durante operações de leitura e escrita no socket, tornando a aplicação menos responsiva, enquanto o comportamento assíncrono permite que o processamento continue sem bloqueio, tornando a aplicação mais responsiva e eficiente. No comportamento síncrono, o programa espera até que a operação de E/S seja concluída, enquanto no assíncrono, a aplicação pode continuar executando outras tarefas em paralelo, o que é especialmente útil em situações de latência de rede. No entanto, a implementação de comportamento assíncrono pode ser mais complexa, exigindo o uso de técnicas adicionais, como threads ou bibliotecas específicas para gerenciar operações assíncronas. Portanto, a escolha entre síncrono e assíncrono depende das necessidades de desempenho e responsividade da aplicação, bem como da complexidade de implementação desejada.