Servidores e clientes baseados em Socket, com uso de Threads tanto no lado

cliente quanto no lado servidor, acessando o banco de dados via JPA.

Eros Santos de Vasconcelos/ 202307120545

Polo Iputinga

RPG0018 - Por que não paralelizar– 9001 – 3° semestre

Objetivo da Prática

O objetivo desta prática é desenvolver e implementar um sistema de comunicação entre um servidor e clientes utilizando Sockets e Threads em Java, com o uso de persistência de dados por meio de JPA. A prática visa aplicar conceitos de programação paralela e comunicação de redes, reforçando a utilização de classes Java para manipulação de conexões e fluxos de dados.

1º Procedimento | Criando o Servidor e Cliente de Teste

Códigos Solicitados:

1. Código da Classe Servidor (ServidorSocket):

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import controller.*;

public class ServidorSocket {

public static void main(String[] args) {
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("CadastroServerPU");

EntityManager em = emf.createEntityManager();

ProdutoJpaController produtoCtrl = new ProdutoJpaController();
UsuarioJpaController usuarioCtrl = new UsuarioJpaController();

try (ServerSocket serverSocket = new ServerSocket(4321)) {
System.out.println("Servidor aguardando conexÃµes na porta 4321...");

while (true) {
Socket socket = serverSocket.accept();
System.out.println("Novo cliente conectado: " + socket.getInetAddress());

CadastroThread cadastroThread = new CadastroThread(produtoCtrl, usuarioCtrl,
socket);

cadastroThread.start();
}
} catch (IOException e) {
e.printStackTrace();
} finally {
em.close();
emf.close();
}
}
}
```

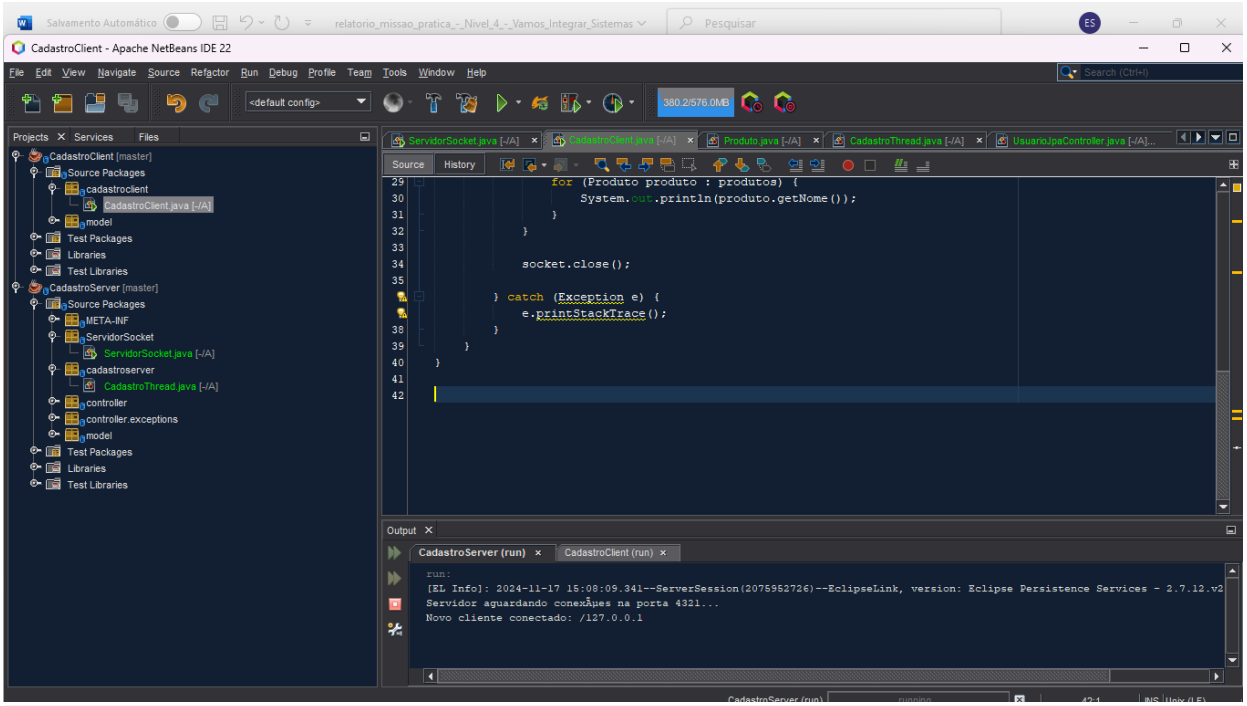
2. Código da Classe CadastroThread:

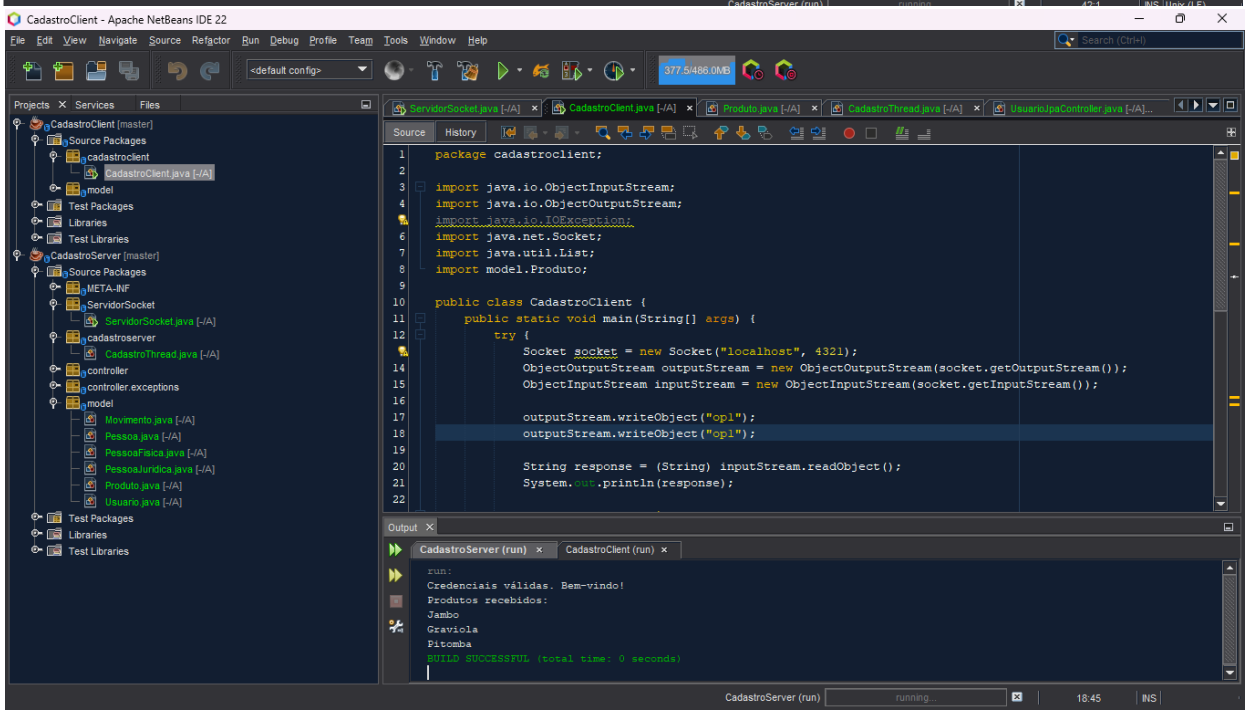
```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.util.List;
import model.*;
import controller.*;

public class CadastroThread extends Thread {
private ProdutoJpaController ctrl;
private UsuarioJpaController ctrlUsu;
private Socket s1;
private ObjectOutputStream out;
private ObjectInputStream in;

public CadastroThread(ProdutoJpaController ctrl, UsuarioJpaController ctrlUsu,
Socket s1) {
this.ctrl = ctrl;
this.ctrlUsu = ctrlUsu;
this.s1 = s1;
}

@Override
public void run() {
try {
ObjectInputStream inputStream = new ObjectInputStream(s1.getInputStream());
ObjectOutputStream outputStream = new ObjectOutputStream(s1.getOutputStream());

String login = (String) inputStream.readObject();
String senha = (String) inputStream.readObject();

Usuario usuario = ctrlUsu.findUsuario(login, senha);
if (usuario == null) {
outputStream.writeObject("Credenciais inválidas");
s1.close();
return;
}

outputStream.writeObject("Credenciais válidas. Bem-vindo!");

String comando = (String) inputStream.readObject();

if ("L".equals(comando)) {
List<Produto> produtos = ctrl.findProdutoEntities();
outputStream.writeObject(produtos);
}

} catch (Exception e) {
e.printStackTrace();
} finally {
try {
s1.close();
} catch (IOException e) {
e.printStackTrace();
}
}
}
}
```

3. Código da Classe Cliente (CadastroClient):

```
package cadastroclient;

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

public class CadastroClient {
public static void main(String[] args) {
try {
Socket socket = new Socket("localhost", 4321);
ObjectOutputStream outputStream = new
ObjectOutputStream(socket.getOutputStream());
ObjectInputStream inputStream = new ObjectInputStream(socket.getInputStream());

outputStream.writeObject("op1");
outputStream.writeObject("op1");

String response = (String) inputStream.readObject();
System.out.println(response);

if ("Credenciais válidas. Bem-vindo!".equals(response)) {
outputStream.writeObject("L");

List<Produto> produtos = (List<Produto>) inputStream.readObject();
System.out.println("Produtos recebidos:");

for (Produto produto : produtos) {
System.out.println(produto.getNome());
}
}

socket.close();

} catch (Exception e) {
e.printStackTrace();
}
}
}
```

Resultados da execução:





Análise e Conclusão:

1. **Como funcionam as classes Socket e ServerSocket?**

A classe ServerSocket permite que um servidor escute conexões de clientes em uma porta específica, enquanto a classe Socket permite que um cliente se conecte a um servidor. Juntas, elas facilitam a criação de canais de comunicação bidirecional em uma rede.

2. **Qual a importância das portas para a conexão com servidores?**

As portas são utilizadas para identificar processos específicos em execução em um servidor, permitindo que múltiplas conexões ocorram simultaneamente em uma única máquina.

3. **Como o NetBeans viabiliza a melhoria de produtividade ao lidar com as tecnologias JPA e EJB? Para que servem as classes de entrada e saída ObjectInputStream e ObjectOutputStream, e por que os objetos transmitidos devem ser serializáveis?**

As classes ObjectInputStream e ObjectOutputStream permitem a leitura e escrita de objetos Java em fluxos de dados. Os objetos devem ser serializáveis para que possam ser convertidos em uma sequência de bytes e transmitidos pela rede.

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

O isolamento foi garantido porque a manipulação e persistência dos dados foram feitas no lado do servidor, enquanto o cliente apenas consumiu os dados de forma controlada. Isso mantém a lógica de negócios centralizada e segura.

Objetivo da Prática

O objetivo desta prática é aprender a implementar e utilizar **Threads** no contexto de comunicação cliente-servidor utilizando **Sockets** no Java. A prática abrange o uso de threads para tratamento assíncrono de respostas enviadas por um servidor, manipulação de interfaces gráficas utilizando a biblioteca Swing, e a implementação de um sistema cliente-servidor funcional, onde o cliente envia e recebe informações do servidor por meio de **Sockets**.

2º Procedimento | Servidor Completo e Cliente Assíncrono

Códigos Solicitados:

1. **Código do Cliente (CadastroClientV2.java):**

```
package cadastroclient;

import java.net.Socket;
import java.io.*;
import javax.swing.*;
import view.SaidaFrame;

public class CadastroClientV2 {
```

```
public static void main(String[] args) {
try (Socket socket = new Socket("localhost", 4321);
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

System.out.println("Conectado ao servidor.");

SaidaFrame janela = new SaidaFrame(null);
janela.setVisible(true);
JTextArea textArea = janela.texto;

ThreadClient thread = new ThreadClient(in, textArea);
thread.start();

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
String comando;

while (true) {
System.out.print("Escolha um comando: L - Listar, X - Finalizar, E - Entrada, S - Saída: ");
comando = reader.readLine();
out.writeObject(comando);
out.flush();

if ("E".equals(comando)) {
System.out.print("ID da pessoa: ");
int idPessoa = Integer.parseInt(reader.readLine());
System.out.print("ID do produto: ");
int idProduto = Integer.parseInt(reader.readLine());
System.out.print("Quantidade: ");
int quantidade = Integer.parseInt(reader.readLine());
System.out.print("Valor Unitário: ");
double valorUnitario = Double.parseDouble(reader.readLine());

out.writeInt(idPessoa);
out.writeInt(idProduto);
out.writeInt(quantidade);
out.writeDouble(valorUnitario);
out.flush();
} else if ("X".equals(comando)) {
System.out.println("Finalizando...");

thread.interrupt();

janela.fechar();
break;
} else if (!"L".equals(comando)) {
System.out.println("Comando inválido.");
}
}

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

} catch (IOException e) {
e.printStackTrace();
}
}
}
```

2. Código do Thread Cliente (ThreadClient.java):

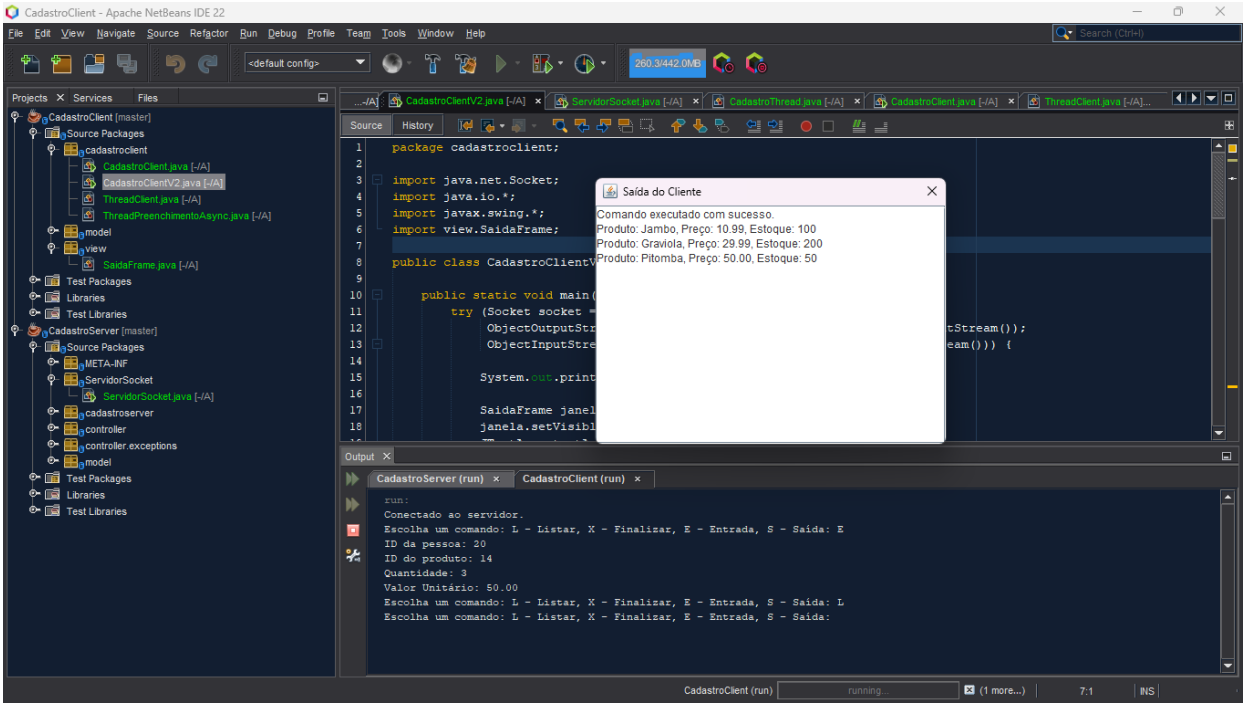
```
package cadastroclient;

import java.io.*;
import javax.swing.*;
import java.util.List;
import model.Produto;

public class ThreadClient extends Thread {
private ObjectInputStream input;
private JTextArea textArea;

public ThreadClient(ObjectInputStream in, JTextArea textArea) {
this.input = in;
this.textArea = textArea;
}

@Override
public void run() {
try {
while (!Thread.currentThread().isInterrupted()) {
Object obj = input.readObject();

if (obj instanceof String) {
String message = (String) obj;
SwingUtilities.invokeLater(() -> textArea.append(message + "\n"));
} else if (obj instanceof List<?>) {
List<Produto> produtos = (List<Produto>) obj;
for (Produto produto : produtos) {
String message = "Produto: " + produto.getNome() + ", Preço: " +
produto.getPrecoVenda() + ", Estoque: " + produto.getQuantidade();
SwingUtilities.invokeLater(() -> textArea.append(message + "\n"));
}
}
}
} catch (IOException | ClassNotFoundException e) {
if (!Thread.currentThread().isInterrupted()) {
e.printStackTrace();
}
} finally {
try {
input.close();
} catch (IOException e) {
e.printStackTrace();
}
}
}
}
```

3. Código da janela para apresentação das mensagens (SaidaFrame.java):

```
package view;
```

```
import javax.swing.*;
import java.awt.*;

public class SaidaFrame extends JDialog {
public JTextArea texto;

public SaidaFrame(Frame owner) {
super(owner, "Saída do Cliente", false);
setBounds(100, 100, 400, 300);

texto = new JTextArea();
texto.setEditable(false);

add(new JScrollPane(texto), BorderLayout.CENTER);

setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);

setLocationRelativeTo(null);
setAlwaysOnTop(true);
setVisible(true);

}

public void fechar() {
this.dispose();
}
}
```

Resultado da execução:



Análise e Conclusão:

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

As Threads são fundamentais para permitir o tratamento assíncrono das respostas do servidor, pois permitem que o programa continue a execução de outras tarefas enquanto aguarda por uma resposta. No contexto de um cliente-servidor, o cliente pode enviar uma solicitação e, ao mesmo tempo, continuar recebendo respostas do servidor sem bloquear a interface de usuário. A utilização de threads garante que a interface gráfica continue responsiva, mesmo durante o processamento de grandes volumes de dados ou enquanto espera por respostas do servidor.A implementação da **ThreadClient** permite que o cliente aguarde as mensagens do servidor em um loop contínuo, processando as respostas e atualizando a interface gráfica em tempo real.

2. **Para que serve o método invokeLater, da classe SwingUtilities?**

No Java, os **Sockets** permitem que os dados sejam enviados e recebidos por meio de streams de entrada e saída. Para enviar e receber objetos, utilizamos **ObjectOutputStream** e **ObjectInputStream**, respectivamente.

- **ObjectOutputStream**: Serializa o objeto e envia pela rede.
- **ObjectInputStream**: Deserializa os objetos recebidos pela rede.

No código do cliente, o ObjectOutputStream é utilizado para enviar comandos e dados ao servidor, enquanto o ObjectInputStream é utilizado para receber as respostas do servidor, que podem ser objetos como String ou listas de Produto.

.

3. **Qual a diferença entre um redirecionamento simples e o uso do método forward, a partir do RequestDispatcher**?

   O redirecionamento (usando response.sendRedirect) faz com que o cliente receba uma nova requisição e possa mudar a URL no navegador. O método forward do RequestDispatcher, por outro lado, mantém a URL original e permite que o servidor encaminhe a requisição para outro recurso no servidor (como outro servlet ou JSP) sem que o cliente perceba.

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

- **Comportamento Síncrono**:
  - No comportamento síncrono, o cliente envia uma solicitação e fica bloqueado aguardando a resposta do servidor antes de continuar com outras operações.
  - O processo de recepção de dados bloqueia a execução do programa, o que pode levar a uma interface gráfica congelada ou sem resposta enquanto o cliente espera por dados.

    - Exemplos de operação síncrona são os métodos que utilizam readObject() sem threads adicionais, onde a execução do programa para até que a resposta seja recebida.

- **Comportamento Assíncrono**:

    - No comportamento assíncrono, o cliente pode continuar executando outras tarefas enquanto aguarda as respostas do servidor.

    - A utilização de **Threads** permite que o programa não bloqueie o fluxo principal, garantindo que o processamento continue enquanto a thread secundária lida com a recepção de dados.

    - O uso de SwingUtilities.invokeLater() permite que a interface gráfica seja atualizada de maneira eficiente e sem bloqueios, tornando o comportamento mais dinâmico e responsivo.

Em resumo, o comportamento assíncrono melhora a performance e a experiência do usuário ao permitir que o programa continue processando outras operações enquanto aguarda a comunicação com o servidor, sem bloquear a execução de outras tarefas ou a interface gráfica.

Conclusão

A utilização de **Threads** e **Sockets** em Java para comunicação cliente-servidor permite a criação de aplicativos mais eficientes e responsivos. O uso de comportamentos assíncronos (como threads para processar as respostas do servidor) é fundamental para garantir que a interface gráfica do cliente permaneça responsiva. Além disso, o método SwingUtilities.invokeLater() é uma ferramenta importante para garantir que as atualizações da interface gráfica ocorram de forma segura. Ao implementar esses conceitos, conseguimos um sistema que realiza a comunicação com o servidor de forma eficiente, sem prejudicar a experiência do usuário.