 Estácio	Universidade Estácio Campus Belém Curso de Desenvolvimento Full Stack Relatório da Missão Prática 5 - Mundo 3
Disciplina:	RPG0018 - Por Que Não Paralelizar?
Nome:	Cleyton Isamu Muto
Turma:	2023.1

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

1. Título da Prática: “1º Procedimento | Criando o Servidor e Cliente de Teste”

2. Objetivo da Prática

- Criar servidores Java com base em Sockets.
- Criar clientes síncronos para servidores com base em Sockets.
- Criar clientes assíncronos para servidores com base em Sockets.
- Utilizar Threads para implementação de processos paralelos.
- 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.

3. Códigos solicitados: estão disponíveis no repositório e lá também compactados em .tar.gz <https://github.com/cleytonmuto/mundo-3-missao-pratica-5>

4. Resultados da execução dos códigos

As estruturas do projeto servidor e do projeto cliente são mostradas nas figuras a seguir, assim como observações pertinentes das implementações solicitadas.

Neste 1º procedimento, a aplicação cliente é denominada “CadastroCliente” e a aplicação servidora correspondente é denominada “CadastroServer”.

No 2º procedimento, a aplicação cliente é denominada “CadastroClienteV2” e a aplicação servidora correspondente é denominada “CadastroServerV2”.

No diretório-raiz do repositório, constam os scripts de criação (create_db.sql) e inserção de dados de teste (insert_db.sql) do banco de dados SQL Server.

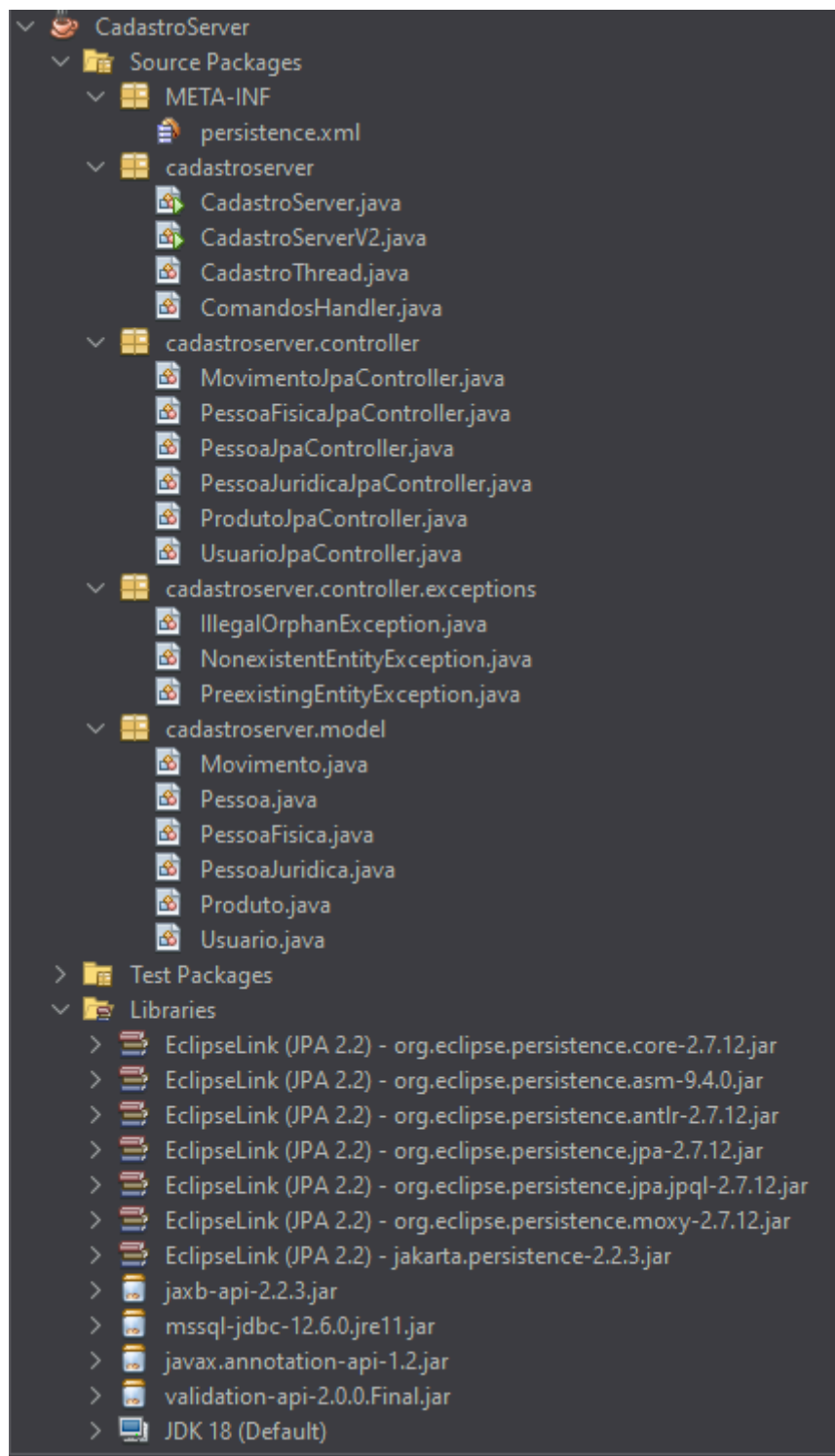


Figura 1. Estrutura das classes e bibliotecas do projeto CadastroServer.

```

public class CadastroServer {

    private final int PORT = 4321;

    public CadastroServer() {

    }

    private void run() {
        try (ServerSocket serverSocket = new ServerSocket(port: PORT)) {
            System.out.println("===== SERVIDOR CONECTADO - PORTA " + PORT + " =====");
            // Inicializa controladores
            EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnitName: "CadastroServerPU");
            ProdutoJpaController produtoController = new ProdutoJpaController(emf);
            MovimentoJpaController movimentoController = new MovimentoJpaController(emf);
            PessoaJpaController pessoaController = new PessoaJpaController(emf);
            UsuarioJpaController usuarioController = new UsuarioJpaController(emf);
            while (true) {
                System.out.println(x: "Aguardando conexao de cliente ...");
                Socket socket = serverSocket.accept();
                System.out.println(x: "Cliente conectado.");
                ClientHandler clientHandler = new ClientHandler(socket, produtoController,
                    movimentoController, pessoaController, usuarioController);
                Thread thread = new Thread(target: clientHandler);
                thread.start();
            }
        } catch (IOException e) {
            Logger.getLogger(name: CadastroServer.class.getName()).log(level: Level.SEVERE, msg: null, thrown: e);
        }
    }

    public static void main(String[] args) {
        new CadastroServer().run();
    }
}

```

Figura 2. Classe CadastroServer com objeto clientHandler encapsulado na Thread.

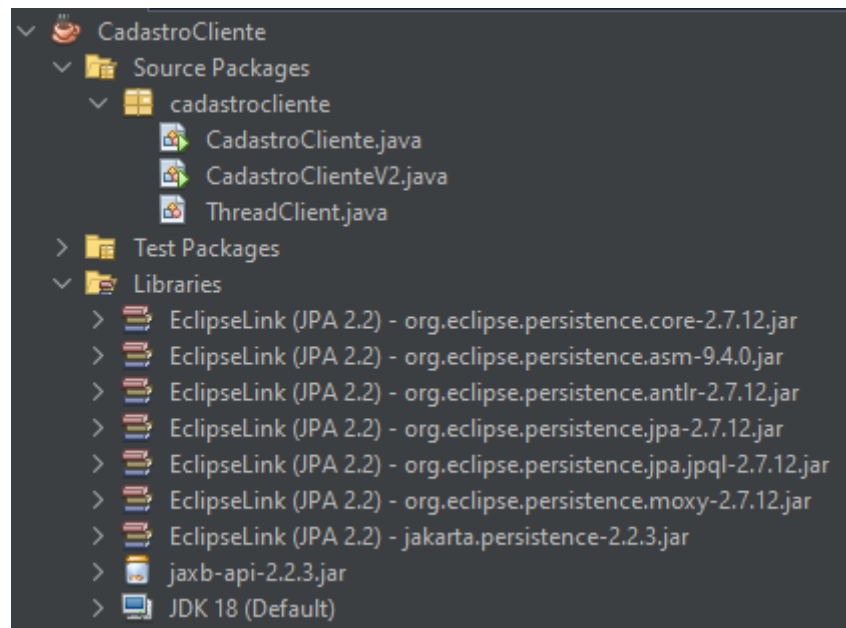


Figura 3. Estrutura das classes e bibliotecas do projeto CadastroCliente.

```

public class CadastroCliente {

    private final String SERVER_ADDRESS = "localhost";
    private final int SERVER_PORT = 4321;

    public CadastroCliente() {

    }

    private void run() {
        try {
            Socket socket = new Socket(host: SERVER_ADDRESS, port: SERVER_PORT);
            BufferedReader in = new BufferedReader(new InputStreamReader(in: socket.getInputStream()));
            PrintWriter out = new PrintWriter(out: socket.getOutputStream(), autoFlush: true);
            BufferedReader consoleIn = new BufferedReader(new InputStreamReader(in: System.in)) {
                System.out.println(x: "Conectado ao servidor CadastroServer.");
                System.out.print(s: "Digite seu nome de usuario: ");
                String username = consoleIn.readLine();
                System.out.print(s: "Digite sua senha: ");
                String password = consoleIn.readLine();
                out.println(x: username);
                out.println(x: password);
                String response = in.readLine();
                System.out.println(x: response);
                if (response.equals(anObject: "Autenticacao bem-sucedida. Aguardando comandos...")) {
                    boolean exitChoice = false;
                    while (!exitChoice) {
                        System.out.print(s: "Digite 'L' para listar produtos ou 'S' para sair: ");
                        String command = consoleIn.readLine().toUpperCase();
                        out.println(x: command);
                        switch (command) {
                            case "S" → exitChoice = true;
                            case "L" → receiveAndDisplayProductList(in);
                            default → System.out.println(x: "Opcao invalida!");
                        }
                    }
                }
            }
        } catch (IOException e) {
            Logger.getLogger(name: CadastroCliente.class.getName()).log(level: Level.SEVERE, msg: null, thrown: e);
        }
    }
}

```

Figura 4. Classe CadastroCliente com Socket.

```

run:
Conectado ao servidor CadastroServer.
Digite seu nome de usuario: op1
Digite sua senha: op1
Autenticacao bem-sucedida. Aguardando comandos...
Digite 'L' para listar produtos ou 'S' para sair: L
Conjunto de produtos disponiveis:
Banana
Laranja
Manga
Digite 'L' para listar produtos ou 'S' para sair: S
BUILD SUCCESSFUL (total time: 12 seconds)

```

Figura 5. CadastroCliente em execução.

5. Análise e Conclusão

(a) Como funcionam as classes Socket e ServerSocket?

A classe Socket é usada para criar um ponto de conexão (socket) para comunicação entre dois dispositivos em rede. Quando o cliente deseja estabelecer uma conexão com o servidor, um Socket é criado para se conectar a um ServerSocket que está em modo de escuta, acessível através de um endereço IP e porta específicos.

Por outro lado, a classe ServerSocket é usada no lado do servidor para escutar e aceitar conexões de clientes. Quando o ServerSocket aceita uma conexão, ele retorna um novo objeto Socket que é usado para a comunicação com o cliente específico. Assim, essa combinação de Socket e ServerSocket permite a troca de dados entre clientes e servidores em uma rede.

(b) Qual a importância das portas para a conexão com servidores?

As portas de comunicação em rede são pontos de conexão que permitem a comunicação entre clientes, servidores e demais dispositivos em uma rede. Cada porta possui um valor associado a um processo ou serviço, o que permite que o tráfego de rede seja direcionado para o processo ou serviço apropriado. Por exemplo, navegadores utilizam normalmente, além da URL ou endereço IP, a porta padrão 80 (http) do servidor, ou 443 (https), para comunicação web.

Serviços de rede comuns seguem uma numeração padrão de portas do servidor [1], associados ao tipo de protocolo da aplicação, por exemplo, porta 21 é usada para transferência de arquivos FTP, porta 25 para envio de emails por SMTP, porta 110 para receber emails por POP3, porta 80 para HTTP, porta 443 para HTTPS, entre vários outros serviços.

É importante ressaltar que esses valores de portas referem-se ao lado servidor, pois o valor de porta utilizada pelo lado cliente é distribuído por disponibilidade, ou seja, não são aleatórios, mas possuem valores acima de 1024, já que portas abaixo desse valor são padronizadas. Para visualizar as portas utilizadas em um computador, é possível utilizar no terminal o comando “netstat -a”.

(c) Para que servem as classes de entrada e saída InputStream e OutputStream, e por que os objetos transmitidos devem ser serializáveis?

Em Java, as classes InputStream e OutputStream são utilizadas para manipular a entrada e saída de objetos, permitindo a leitura e escrita de objetos em fluxos de dados, denominados “*streams*”. Essas classes fazem parte do pacote java.io e são essenciais para a serialização e desserialização de objetos.

A classe OutputStream é usada para escrever objetos em uma *stream* de saída. Ela permite que objetos sejam convertidos em um formato que pode ser armazenado ou transmitido, geralmente para um arquivo ou para a rede.

A seguir, exemplo de uso da classe OutputStream:

```

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class WriteObject {
    public static void main(String[] args) {
        try (FileOutputStream fileOut = new FileOutputStream("object.dat");
            ObjectOutputStream out = new ObjectOutputStream(fileOut)) {
            MyClass object = new MyClass("example", 123);
            out.writeObject(object);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Figura 6. Exemplo de uso da classe ObjectOutputStream.

A classe ObjectInputStream é usada para ler objetos de uma *stream* de entrada. Ela converte os dados de volta em objetos, o que permite que objetos previamente serializados sejam reconstruídos em seu estado original.

```

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class ReadObject {
    public static void main(String[] args) {
        try (FileInputStream fileIn = new FileInputStream("object.dat");
            ObjectInputStream in = new ObjectInputStream(fileIn)) {
            MyClass object = (MyClass) in.readObject();
            System.out.println(object);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

Figura 7. Exemplo de uso da classe ObjectInputStream.

Contudo, para que um objeto possa ser escrito em um ObjectOutputStream e lido de volta através de um ObjectInputStream, a sua classe de referência deve implementar a interface Serializable. A serialização é o processo de converter um objeto em uma sequência de bytes, que pode ser armazenada ou transmitida. A desserialização é o processo inverso, o qual converte a sequência de bytes de volta em um objeto.

```

import java.io.Serializable;

public class MyClass implements Serializable {
    private static final long serialVersionUID = 1L;

    private String name;
    private int value;

    public MyClass(String name, int value) {
        this.name = name;
        this.value = value;
    }
}

```

```
@Override
public String toString() {
    return "MyClass{name='" + name + "', value=" + value + "}";
}
}
```

Figura 8. Exemplo de uso de classe serializável MyClass.

Dentre as razões para a necessidade de serialização, é possível citar [1]:

- *persistência*: armazenar o estado de um objeto em um arquivo para ser recuperado posteriormente.
- *comunicação*: transmitir objetos entre diferentes máquinas ou processos através de redes, como em RMI (*Remote Method Invocation*).
- *caching*: armazenar objetos em caches para acesso rápido.
- *deep copy*: criar uma cópia de um objeto, serializar e desserializar o objeto.

Observações:

- *serialVersionUID*: é uma versão de controle utilizada durante a desserialização para garantir que um objeto lido corresponda a uma versão compatível da classe. Se não for especificado, o Java gera um automaticamente.
- *campos transitórios*: Campos marcados com a palavra-chave “*transient*” não são serializados. Eles são ignorados durante o processo de serialização.
- *implements Serializable*: é essencial para permitir que os objetos sejam corretamente serializados e desserializados, garantindo a integridade e a compatibilidade dos dados durante a transmissão ou armazenamento.

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

O isolamento do acesso ao banco de dados é garantido pois a JPA, como parte da especificação Java EE, é projetada para separar claramente as operações de lógica de negócios da camada de persistência de dados.

As classes de entidade (@Entity) JPA representam a estrutura dos dados e são usadas tanto no cliente quanto no servidor, mas a gestão real do acesso ao banco de dados é realizada exclusivamente no lado do servidor. Isso significa que as operações de banco de dados, tais como inserções, consultas, atualizações, exclusões (*CRUD* - *create, read, update, delete*), são executadas por meio de sessões e transações gerenciadas pelo servidor, geralmente através de um provedor de persistência.

Portanto, mesmo que o cliente use as mesmas classes de entidade, ele não tem acesso direto ao banco de dados, o que garante o isolamento e a segurança dos dados.

1. Título da Prática: “2º Procedimento | Servidor Completo e Cliente Assíncrono”

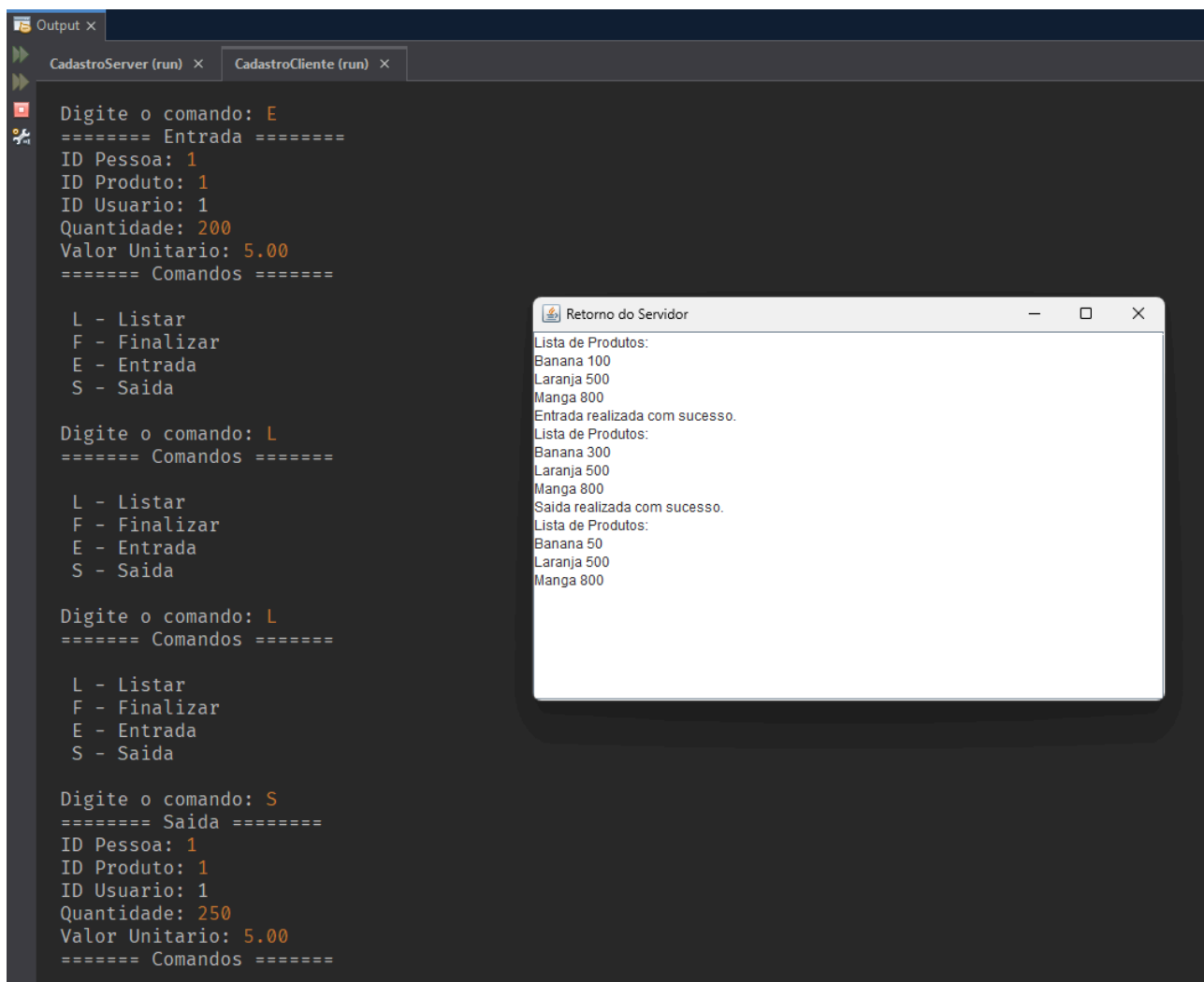
2. Objetivo da Prática

- Criar servidores Java com base em Sockets.
- Criar clientes síncronos para servidores com base em Sockets.
- Criar clientes assíncronos para servidores com base em Sockets.
- Utilizar Threads para implementação de processos paralelos.
- 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.

3. Códigos solicitados: estão disponíveis no repositório e lá também compactados em .tar.gz

<https://github.com/cleytonmuto/mundo-3-missao-pratica-5>

4. Resultados da execução dos códigos



```
Output x
CadastroServer (run) x CadastroCliente (run) x

Digite o comando: E
===== Entrada =====
ID Pessoa: 1
ID Produto: 1
ID Usuario: 1
Quantidade: 200
Valor Unitario: 5.00
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: L
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: L
===== Comandos =====

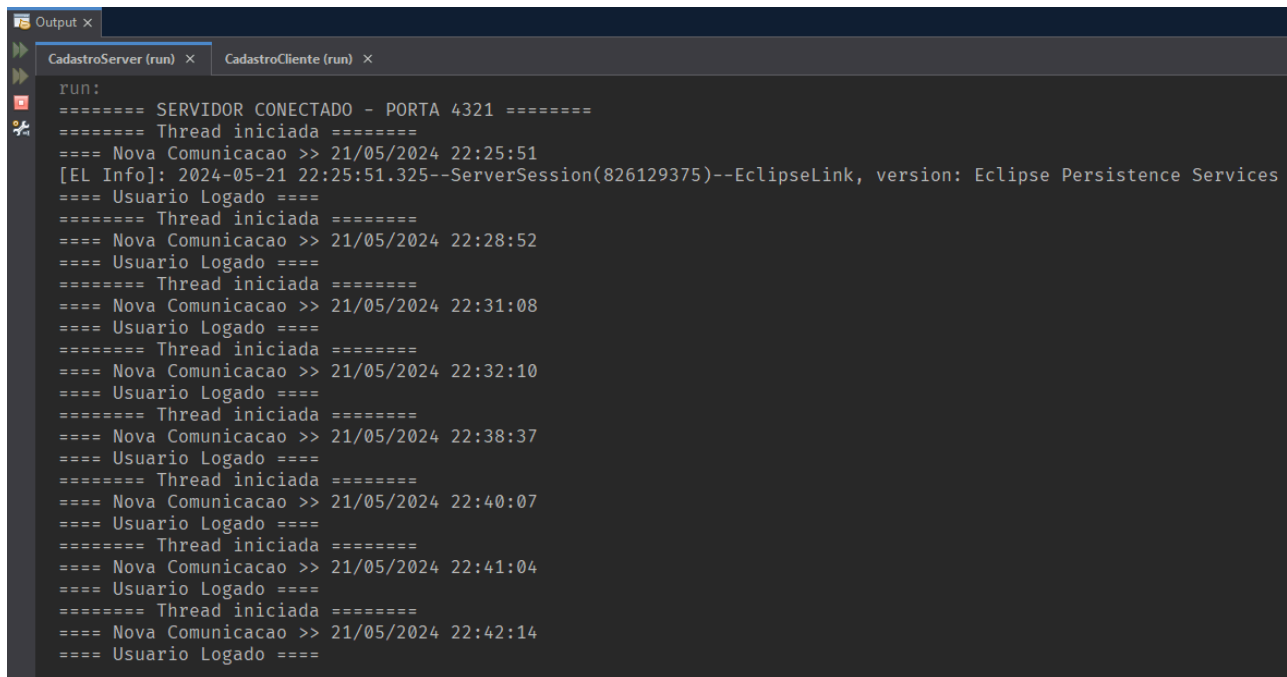
L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: S
===== Saida =====
ID Pessoa: 1
ID Produto: 1
ID Usuario: 1
Quantidade: 250
Valor Unitario: 5.00
===== Comandos =====
```

Retorno do Servidor

Lista de Produtos:
Banana 100
Laranja 500
Manga 800
Entrada realizada com sucesso.
Lista de Produtos:
Banana 300
Laranja 500
Manga 800
Saida realizada com sucesso.
Lista de Produtos:
Banana 50
Laranja 500
Manga 800

Figura 9. Classe CadastroClienteV2 em execução, com exemplos de movimentações (E/S).



```
run:
===== SERVIDOR CONECTADO - PORTA 4321 =====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:25:51
[EL Info]: 2024-05-21 22:25:51.325--ServerSession(826129375)--EclipseLink, version: Eclipse Persistence Services
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:28:52
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:31:08
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:32:10
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:38:37
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:40:07
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:41:04
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:42:14
==== Usuario Logado =====
```

Figura 10. Classe CadastroServerV2 em execução.

5. Análise e Conclusão

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

Quando um servidor recebe múltiplas solicitações de clientes, pode utilizar Threads para processar cada solicitação simultaneamente, de modo a evitar assim um bloqueio, enquanto aguarda, por exemplo, por uma operação de longa duração em uma única solicitação.

Cada solicitação é processada em uma Thread separada, o que permite que o servidor continue a receber e responder outras solicitações. Essa abordagem melhora a eficiência e a capacidade de resposta do servidor, pois as Threads operam de forma independente e paralela, isto garante que o processamento de uma solicitação não seja interrompido ou atrasado pelas outras.

Em resumo, o uso de Threads no servidor possibilita o tratamento assíncrono das respostas, o que melhora o desempenho geral do sistema em ambientes de rede.

(b) Para que serve o método `invokeLater`, da classe `SwingUtilities`?

O método `invokeLater` da classe `SwingUtilities` é utilizado na programação de interfaces gráficas em Java, especificamente no Swing framework.

Este método é importante para garantir que as alterações na interface do usuário sejam feitas de maneira segura no contexto da Event Dispatch Thread (EDT), que é a thread responsável por gerenciar eventos e atualizações de interface gráfica em Swing.

Quando se deseja executar um bloco de código que altera a interface do usuário, como atualizar um componente gráfico ou responder a eventos de usuário, `invokeLater` é utilizado para enfileirar esse bloco de código na EDT. Isto assegura que as mudanças na interface sejam realizadas de maneira sequencial e segura, assim evita problemas de concorrência e inconsistências na interface gráfica.

(c) Como os objetos são enviados e recebidos pelo Socket Java?

Em Java, os objetos são enviados e recebidos através de sockets com uso de fluxos (*streams*), especificamente as classes `ObjectOutputStream` e `ObjectInputStream`.

Para enviar um objeto, primeiro serializa-se o objeto, ou seja, o objetivo é convertido em uma sequência de bytes, com uso de `ObjectOutputStream`; em seguida, é enviado através de um `Socket`.

Do lado receptor, `ObjectInputStream` é usado para ler a sequência de bytes do `Socket` e desserializá-lo (reconstruir o objeto original). É crucial que o objeto a ser enviado implemente a interface `Serializable`, o que permite essa serialização e desserialização.

Essa abordagem permite a transmissão de objetos complexos e suas propriedades através da rede de forma eficiente, com a manutenção da integridade e do estado dos objetos.

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

Na programação de rede com Sockets em Java, o comportamento síncrono e assíncrono possui implicações significativas no bloqueio do processamento.

As operações síncronas bloqueiam a execução do programa até que a operação de rede seja concluída; por exemplo, um método `read()` em um `Socket` síncrono pausará a execução até que os dados sejam recebidos. Isso pode simplificar a lógica de programação, mas pode levar à ineficiência, pois o thread que executa a operação fica inativo enquanto espera.

Por outro lado, um comportamento assíncrono, frequentemente implementado através de callbacks ou promises, permite que o programa continue a executar outras tarefas enquanto aguarda a conclusão da operação de rede. Isso aumenta a eficiência e a capacidade de resposta do aplicativo, pois os recursos de processamento podem ser utilizados para outras tarefas durante as operações de espera, mas isto torna a lógica do programa mais complexa, devido ao gerenciamento das operações não sequenciais.

Referências

[1] **“What is serialization in Java”**

Disponível em <https://www.shiksha.com/online-courses/articles/understanding-serialization-in-java>

Acesso em 19 de maio de 2024.