UNIVERSIDADE FEDERAL DO CEARÁ - CAMPUS QUIXADÁ
DISCIPLINA: SISTEMAS DISTRIBUÍDOS
PROFESSOR: Rafael Braga
DISCENTES: Pablo Brandão Passos (539730) e Pedro Wilson Coelho Parreira (541491)
CURSO: Engenharia da Computação **|** **SEMESTRE:** 8°

Trabalho 3 (EJB-servidor-java-WS-API): RELATÓRIO TÉCNICO DE IMPLEMENTAÇÃO: SISTEMA DISTRIBUÍDO PARA GESTÃO DE PEDIDOS DE UM RESTAURANTE

Nota: 10
Nível de dificuldade: Difícil

QUIXADÁ - CE
14 DE JANEIRO DE 2026

RESUMO

Este documento descreve o desenvolvimento do terceiro trabalho prático da disciplina de Sistemas Distribuídos. O projeto consiste na implementação de um sistema de gerenciamento de restaurante baseado na arquitetura REST (Representational State Transfer). O objetivo central foi substituir a comunicação fortemente acoplada (Sockets/RMI) por um modelo desacoplado, utilizando HTTP como protocolo de transporte e JSON para serialização de dados. O sistema integra um backend em Java (Spark Framework), frontends em Python (Streamlit) e clientes de automação em Node.js, demonstrando interoperabilidade, concorrência via Threads e transparência de localização.

SUMÁRIO

1. INTRODUÇÃO

A evolução dos sistemas computacionais modernos caminha invariavelmente para a descentralização. A capacidade de integrar diferentes plataformas, linguagens e dispositivos em uma única rede coesa é o pilar da computação distribuída atual.

Este relatório documenta o desenvolvimento do Trabalho 3, cujo escopo é a criação de um sistema de gerenciamento de pedidos para um restaurante. Diferente das abordagens anteriores baseadas em Sockets puros ou RMI (Remote Method Invocation), este projeto adota uma arquitetura orientada a serviços (SOA) implementada através de uma **API REST (Representational State Transfer)**.

O sistema simula um ambiente real onde clientes realizam pedidos via interfaces diversas (Web e Mobile), e o servidor central gerencia o ciclo de vida desses pedidos (recebimento, preparo na cozinha e entrega), lidando com desafios inerentes à disciplina, como concorrência, latência e heterogeneidade de tecnologias.

2. OBJETIVOS

O objetivo principal deste trabalho é demonstrar a aplicação prática dos conceitos de Sistemas Distribuídos, especificamente:

- **Interoperabilidade:** Capacidade de sistemas escritos em linguagens diferentes (Java, Python e Node.js) trocarem informações de forma transparente.
- **Transparência de Localização:** O sistema deve permitir acesso de qualquer dispositivo na rede local, abstraindo a localização física do servidor.
- **Concorrência e Assincronismo:** O servidor deve ser capaz de receber novos pedidos enquanto processa (cozinha) pedidos anteriores, sem bloquear a thread principal.
- **Padronização:** Utilização do protocolo HTTP e formato JSON como meio universal de comunicação.

3. ARQUITETURA DO SISTEMA E TECNOLOGIAS UTILIZADAS

O sistema foi projetado sob a arquitetura **Cliente-Servidor**, utilizando o estilo arquitetural **REST**. A comunicação é realizada via protocolo HTTP, onde os recursos (Pedidos, Cardápio) são manipulados através dos verbos padrão (GET, POST).

3.1 DIAGRAMA DE COMPONENTES

A solução é composta por três camadas lógicas distintas:

1. **Backend (Servidor Central):**
   - Desenvolvido em **Java 17**.
   - Responsável pela lógica de negócios, manutenção do estado dos pedidos em memória e controle de concorrência.
   - Atua como a "Single Source of Truth" (Fonte Única da Verdade).

2. **Frontend (Interfaces de Usuário):**
   - **Interface Web (Streamlit):** Aplicação em Python que serve tanto como cardápio digital para o cliente quanto como dashboard administrativo.
   - **Clientes CLI (Scripts):** Scripts de automação em **Node.js** e **Python** para testes de carga e demonstração de interoperabilidade.
3. **Middleware / Camada de Rede:**
   - Uso de **JSON (*JavaScript Object Notation*)** como payload de dados.
   - Exposição na rede local (Bind IP *0.0.0.0*), permitindo acesso via dispositivos móveis.

3.2 STACK TECNOLÓGICO

A escolha das tecnologias visou maximizar a heterogeneidade para provar a flexibilidade da arquitetura REST:

- **Java (Maven, Spark Framework, GSON):** Escolhido para o backend devido à sua robustez no tratamento de Threads e tipagem forte. O framework Spark foi utilizado pela sua leveza na criação de microservices.
- **Python (Streamlit, Requests):** Escolhido para o frontend pela capacidade de prototipagem rápida de interfaces de dados e facilidade de requisições HTTP.
- **Node.js (Axios):** Utilizado para demonstrar que uma stack baseada em JavaScript pode consumir a mesma API que o Python, sem alterações no servidor.
- **Batch Scripting (.bat):** Utilizado para orquestração e automação do ambiente de execução no Windows.

4. IMPLEMENTAÇÃO TÉCNICA

4.1 SERVIDOR JAVA E O MODELO DE CONCORRÊNCIA

O coração do sistema é o servidor Java. Para atender ao requisito de processamento não-bloqueante, foi implementado um modelo de **Worker Threads**.

Quando uma requisição *POST /pedido* chega:

1. O servidor valida o JSON e cria um objeto Pedido.
2. O pedido recebe o status *RECEBIDO*.
3. Uma nova **Thread** é instanciada (simulando a Cozinha).
4. A resposta HTTP é enviada imediatamente ao cliente (Status 201), enquanto a Thread continua rodando em background.

```
// Pseudocódigo da lógica de concorrência
post("/pedido", (req, res) -> {
    Pedido p = gson.fromJson(req.body(), Pedido.class);
    Repo.adicionar(p);

    new Thread(() -> {
        Thread.sleep(5000); // Simula cozimento
        p.setStatus("PRONTO");
    }).start();

    return "Pedido Recebido";
});
```

Isso garante que o servidor nunca trave, mesmo se 50 pedidos chegarem simultaneamente.

4.2 INTEROPERABILIDADE E PROTOCOLO JSON

A comunicação entre as camadas é estritamente realizada via JSON. Isso elimina a dependência de plataforma.

Exemplo de Payload de envio:

```
{
  "cliente": "Pedro",
  "item": "Hamburguer",
  "quantidade": 2
}
```

Tanto o script em Python quanto o script em Node.js geram exatamente a mesma estrutura de JSON, tornando-os indistinguíveis para o servidor Java.

4.3 INTERFACE REATIVA COM POLLING

Como o protocolo HTTP é *stateless* (sem estado) e o servidor não envia "push notifications" nativamente nesta arquitetura simples, os clientes (Streamlit) utilizam a técnica de **Polling**. A interface consulta o endpoint *GET /pedidos* a cada 2 ou 5 segundos para atualizar o status na tela do usuário (de "*PREPARANDO*" para "*PRONTO*").

5. AUTOMAÇÃO E ORQUESTRAÇÃO (SCRIPTING)

Um dos desafios em durante o trabalho foi a complexidade de iniciar múltiplos processos (servidor, cliente web, cliente admin). Para solucionar isso, foi desenvolvido um orquestrador via script **Batch** (*init.bat*). Inicialmente, executar o projeto exigia abrir 4 terminais diferentes e digitar comandos longos.

5.1 EVOLUÇÃO DO SCRIPT *init.bat*

O script passou por várias iterações para lidar com limitações do Windows:

1. **Compilação Automática:** O script entra na pasta do servidor e executa *mvn clean package* antes de iniciar, garantindo que o código esteja sempre atualizado.

2. **Execução em Background:** Utilizou-se o comando *start /min* para iniciar os processos sem poluir a área de trabalho do usuário.

3. **Headless Mode:** Adicionou-se a flag *--server.headless=true* nos comandos do Streamlit para evitar que o navegador abrisse múltiplas abas descontroladamente.

4. **Menu Interativo:** Implementou-se um loop com *set /p* para capturar input do usuário, permitindo lançar scripts de teste (Node/Python) sob demanda sem fechar o servidor principal.

5. **Correção de Sintaxe:** Problemas com caracteres reservados (|, >) foram resolvidos usando caracteres de escape (^|), permitindo uma interface de terminal rica e informativa.

6. **Network Discovery:** Identifica automaticamente o IP da máquina na rede local e exibe um QR Code ou URL para acesso via outro dispositivo que não seja local (porém na mesma rede).

6. DESAFIOS E TESTES

6.1 DESAFIOS TÉCNICOS RESOLVIDOS

1. **Bloqueio de Threads:** Inicialmente, o servidor processava o pedido sequencialmente. Se um prato demorasse 10 segundos, ninguém mais conseguia pedir.
   Solução: Implementação de Threads assíncronas.

2. **Visibilidade na Rede:** O servidor rodava em *localhost*, impedindo acesso pelo celular.
   Solução: Alteração do binding para *0.0.0.0*, expondo o serviço para a LAN.

3. **Confusão de Dependências:** O Git estava rastreando arquivos compilados *(.class, target/*).
   Solução: Criação rigorosa de um arquivo *.gitignore* na raiz e nas subpastas.

6.2 EXPERIÊNCIA MOBILE

Graças à configuração do servidor para escutar no endereço *0.0.0.0* (todas as interfaces de rede), foi possível demonstrar a **Transparência de Localização**.

- O servidor roda no Notebook do aluno.
- O cliente acessa via navegador do Smartphone conectado ao Wi-Fi.
- A experiência é fluida e indistinguível de uma aplicação rodando nativamente no celular.

7. RESULTADOS OBTIDOS

Os testes realizados demonstraram o sucesso da implementação:

1. **Robustez:** O servidor manteve-se estável mesmo com múltiplos scripts de teste (Python e Node) enviando requisições em *loop*.

2. **Integração:** O Dashboard Administrativo refletiu em tempo real os pedidos feitos via celular e via terminal, provando que todos consomem a mesma base de dados.

3. **Concorrência:** Foi verificado que pedidos longos (que demoram a cozinhar) não impedem que novos pedidos sejam aceitos, validando o uso de Threads.

8. CONCLUSÃO

O desenvolvimento deste trabalho permitiu consolidar os conhecimentos teóricos sobre sistemas distribuídos. A transição de Sockets para API REST demonstrou as vantagens da abstração HTTP: facilidade de depuração, padronização de mensagens e, principalmente, a capacidade de integrar tecnologias heterogêneas, ou seja, a migração de uma arquitetura proprietária e acoplada (utilizada nos trabalhos anteriores) para uma arquitetura baseada em REST e JSON, e isso demonstrou claras vantagens:

- **Facilidade de Integração:** Foi trivial conectar um cliente Node.js a um backend Java.

- **Escalabilidade:** O modelo *stateless* do HTTP facilita (teoricamente) a replicação do servidor, embora neste trabalho tenhamos usado memória local.

- **Experiência de Usuário:** A separação entre Frontend e Backend permitiu criar interfaces ricas sem impactar a lógica de negócio.

O uso de Java para o processamento robusto no backend, aliado à agilidade do Python e JavaScript no frontend, provou ser uma arquitetura moderna e eficiente. Além disso, a automação do ambiente de execução garantiu que a complexidade da arquitetura distribuída ficasse transparente para o usuário final, resultando em um sistema funcional, resiliente e de fácil utilização.

O projeto foi organizado para separar claramente as responsabilidades. A estrutura final, com códigos, arquivos, instruções e documentação está presente no repositório GitHub abaixo:

LINK:

https://github.com/PabloBr4ndao/SD_Trabalho_3_EJB-servidor-java-WS-API