UNIVERSIDADE FEDERAL

CURSO DE SISTEMAS DISTRIBUÍDOS

TRABALHO 3 - WEB SERVICES E API REST

Aluno: Francisco Ulisses

Disciplina: Sistemas Distribuídos

Data: Janeiro de 2026

---

1. INTRODUÇÃO

Este relatório documenta a implementação do Trabalho 3 da disciplina de Sistemas Distribuídos, que consiste na reimplementação de um serviço remoto utilizando Web Services ou API REST, conforme especificado no material didático. O trabalho visa demonstrar a aplicação prática de conceitos de comunicação distribuída utilizando protocolos de alto nível, diferenciando-se das implementações anteriores baseadas em sockets TCP e RMI manual.

2. OBJETIVO

O objetivo principal deste trabalho é desenvolver um sistema distribuído de gerenciamento de telefonia utilizando arquitetura REST (Representational State Transfer) sobre protocolo HTTP, garantindo a interoperabilidade entre diferentes linguagens de programação e demonstrando a comunicação cliente-servidor em ambiente heterogêneo.

3. RESUMO DO TRABALHO

O sistema implementado consiste em uma API REST para gerenciamento de serviços de telefonia, permitindo operações de cadastro de clientes, gerenciamento de linhas telefônicas, registro de chamadas, geração de faturas e consulta de estatísticas. A arquitetura adotada segue o padrão cliente-servidor, onde o servidor é implementado em Java utilizando a biblioteca HttpServer nativa, e os clientes são implementados em duas linguagens distintas: Python e JavaScript (Node.js).

O sistema atende aos requisitos especificados no trabalho, a saber:

a) Reimplementação do serviço remoto do trabalho anterior via Web Service ou API REST;

b) Não utilização de sockets de baixo nível ou RMI;

c) Implementação de clientes em pelo menos duas linguagens diferentes da linguagem do servidor;

d) Suporte à interação simultânea entre múltiplos clientes, permitindo demonstração de comunicação entre estudantes da dupla.

4. ARQUITETURA DO SISTEMA

4.1. Servidor

O servidor foi desenvolvido em linguagem Java, utilizando a classe HttpServer do pacote com.sun.net.httpserver. Esta escolha tecnológica permite a implementação de um servidor HTTP completo

sem necessidade de manipulação direta de sockets, atendendo aos requisitos do trabalho.

O servidor opera na porta 8080 e implementa os seguintes endpoints REST:

- GET /api/health: Verificação do status do servidor
- POST /api/clientes: Cadastro de novo cliente
- GET /api/clientes: Listagem de todos os clientes
- GET /api/clientes/{cpf}: Consulta de cliente específico
- DELETE /api/clientes/{cpf}: Remoção de cliente
- POST /api/linhas: Adição de linha telefônica
- DELETE /api/linhas: Remoção de linha telefônica
- POST /api/chamadas: Registro de chamada telefônica
- POST /api/faturas: Geração de fatura para cliente
- GET /api/faturas/{cpf}: Listagem de faturas do cliente
- GET /api/estatisticas: Consulta de estatísticas do sistema

A serialização e deserialização de dados é realizada utilizando o formato JSON através da biblioteca Google Gson versão 2.10.1.

4.2. Clientes

Foram implementados dois clientes em linguagens distintas:

Cliente Python: Utiliza a biblioteca requests para comunicação HTTP. Implementa interface interativa via linha de comando, permitindo ao usuário executar todas as operações disponíveis na API.

Cliente Node.js: Utiliza a biblioteca axios para comunicação HTTP assíncrona. Também implementa interface interativa via linha de comando com funcionalidades equivalentes ao cliente Python.

Ambos os clientes se comunicam com o servidor utilizando o protocolo HTTP na porta 8080, enviando e recebendo dados no formato JSON.

5. REQUISITOS DO SISTEMA

5.1. Requisitos para o Servidor

Para executar o servidor, são necessários os seguintes componentes:

- Java Development Kit (JDK) versão 8 ou superior
- Sistema operacional Windows, Linux ou macOS
- Porta 8080 disponível no sistema
- Biblioteca Gson 2.10.1 (download automático via script de compilação)

5.2. Requisitos para o Cliente Python

Para executar o cliente Python, são necessários:

- Python versão 3.6 ou superior
- Biblioteca requests (instalação via pip install requests)
- Acesso à rede local ou remota onde o servidor está executando

5.3. Requisitos para o Cliente Node.js

Para executar o cliente Node.js, são necessários:

- Node.js versão 14 ou superior
- Gerenciador de pacotes npm (incluído na instalação do Node.js)
- Biblioteca axios (instalação via npm install)
- Acesso à rede local ou remota onde o servidor está executando

6. INSTALAÇÃO E CONFIGURAÇÃO

6.1. Instalação do Java Development Kit

Para sistemas Windows:

1. Acessar o site oficial da Oracle ou AdoptOpenJDK
2. Baixar o instalador apropriado para o sistema operacional
3. Executar o instalador e seguir as instruções
4. Configurar a variável de ambiente JAVA_HOME
5. Adicionar o diretório bin do Java ao PATH do sistema
6. Verificar a instalação executando: java -version

6.2. Instalação do Python

Para sistemas Windows:

1. Acessar o site oficial python.org
2. Baixar o instalador da versão mais recente
3. Executar o instalador, marcando a opção "Add Python to PATH"
4. Completar a instalação
5. Verificar a instalação executando: python --version
6. Instalar a biblioteca requests executando: pip install requests

6.3. Instalação do Node.js

Para sistemas Windows:

1. Acessar o site oficial nodejs.org
2. Baixar o instalador LTS (Long Term Support)
3. Executar o instalador e seguir as instruções padrão
4. Verificar a instalação executando: node --version
5. Verificar o npm executando: npm --version
6. EXECUÇÃO DO SISTEMA

7.1. Compilação e Execução do Servidor

Etapa 1: Compilar o servidor Navegue até o diretório raiz do projeto tb3 e execute o script de compilação:

```
compilar-servidor.bat
```

Este script realizará automaticamente as seguintes operações:

- Criação do diretório de classes compiladas
- Download da biblioteca Gson caso não exista
- Compilação de todos os arquivos Java
- Configuração do classpath

Etapa 2: Iniciar o servidor Após compilação bem-sucedida, execute o script de inicialização:

```
iniciar-servidor.bat
```

O servidor será iniciado e exibirá mensagens indicando:

- Inicialização do HttpServer
- Porta de escuta (8080)
- Endpoints registrados
- Status de prontidão para receber requisições

O servidor permanecerá em execução até ser manualmente encerrado através do comando Ctrl+C ou fechamento da janela do terminal.

7.2. Execução do Cliente Python

Etapa 1: Instalar dependências (primeira execução) Navegue até o diretório do cliente Python:

```
cd cliente-python
pip install -r requirements.txt
```

Etapa 2: Executar o cliente A partir do diretório raiz tb3, execute:

```
iniciar-cliente-python.bat
```

Alternativamente, a partir do diretório cliente-python:

```
python cliente.py
```

O cliente apresentará um menu interativo com as seguintes opções:

1. Adicionar Cliente
2. Listar Clientes
3. Consultar Cliente
4. Remover Cliente

5. Adicionar Linha
6. Remover Linha
7. Registrar Chamada
8. Gerar Fatura
9. Listar Faturas
10. Ver Estatísticas
11. Sair

7.3. Execução do Cliente Node.js

Etapa 1: Instalar dependências (primeira execução) Navegue até o diretório do cliente Node.js:

```
cd cliente-nodejs
npm install
```

Etapa 2: Executar o cliente A partir do diretório raiz tb3, execute:

```
iniciar-cliente-nodejs.bat
```

Alternativamente, a partir do diretório cliente-nodejs:

```
node cliente.js
```

O cliente apresentará menu interativo idêntico ao cliente Python, garantindo funcionalidade equivalente em ambas as implementações.

8. PROCEDIMENTOS DE TESTE

8.1. Teste Básico de Conectividade

Antes de iniciar os testes funcionais, é recomendável verificar se o servidor está respondendo corretamente. Isto pode ser feito através de requisição HTTP ao endpoint de health check:

Utilizando navegador web: http://localhost:8080/api/health

Utilizando PowerShell: Invoke-WebRequest -Uri "http://localhost:8080/api/health"

Resposta esperada: Status: 200 OK Conteúdo: {"service":"Sistema de Telefonia API","version":"1.0","status":"OK"}

8.2. Teste Funcional Sequencial

Este teste valida o fluxo completo de operações do sistema:

Passo 1: Adicionar Cliente

- Executar cliente Python ou Node.js

- Selecionar opção 1 (Adicionar Cliente)
- Informar dados: nome, CPF, telefone e email
- Verificar mensagem de sucesso

Passo 2: Consultar Cliente Cadastrado

- Selecionar opção 3 (Consultar Cliente)
- Informar o CPF cadastrado no passo anterior
- Verificar exibição dos dados corretos

Passo 3: Adicionar Linha Telefônica

- Selecionar opção 5 (Adicionar Linha)
- Informar CPF do cliente e número da linha
- Verificar mensagem de sucesso

Passo 4: Registrar Chamada

- Selecionar opção 7 (Registrar Chamada)
- Informar número de origem (linha cadastrada), destino e duração
- Verificar mensagem de confirmação

Passo 5: Gerar Fatura

- Selecionar opção 8 (Gerar Fatura)
- Informar CPF do cliente
- Verificar exibição da fatura formatada com detalhes da chamada

Passo 6: Consultar Estatísticas

- Selecionar opção 10 (Ver Estatísticas)
- Verificar totalização correta de clientes, linhas e chamadas

8.3. Teste de Integração Multi-Cliente

Este teste demonstra a capacidade de múltiplos clientes interagirem simultaneamente com o servidor, compartilhando estado:

Passo 1: Iniciar Servidor

- Executar script iniciar-servidor.bat

Passo 2: Iniciar Cliente Python

- Em novo terminal, executar iniciar-cliente-python.bat
- Adicionar cliente "Cliente A" através do menu interativo

Passo 3: Iniciar Cliente Node.js

- Em terceiro terminal, executar iniciar-cliente-nodejs.bat
- Listar clientes (opção 2)
- Verificar que "Cliente A" aparece na listagem

Passo 4: Validar Compartilhamento de Estado

- No cliente Node.js, adicionar "Cliente B"
- No cliente Python, listar clientes
- Verificar que ambos "Cliente A" e "Cliente B" são exibidos
- Em ambos clientes, consultar estatísticas
- Verificar que os valores são idênticos em ambas as consultas

Este teste confirma que:

- Múltiplos clientes podem conectar simultaneamente
- O estado é compartilhado entre todos os clientes
- As operações são sincronizadas corretamente
- Clientes em linguagens diferentes interoperam perfeitamente

8.4. Teste de Persistência de Dados

Durante a execução do servidor, todos os dados são mantidos em memória. Para validar a consistência:

Passo 1: Realizar múltiplas operações

- Adicionar vários clientes
- Adicionar múltiplas linhas
- Registrar diversas chamadas

Passo 2: Consultar estatísticas periodicamente

- Verificar que os totalizadores aumentam corretamente
- Validar cálculo da média de duração das chamadas

Passo 3: Gerar faturas para diferentes clientes

- Verificar que cada fatura contém apenas as chamadas do respectivo cliente
- Validar cálculo correto dos valores

8.5. Teste de Tratamento de Erros

Para validar o tratamento adequado de situações excepcionais:

Teste 1: Cliente inexistente

- Tentar consultar cliente com CPF não cadastrado
- Verificar mensagem de erro apropriada

Teste 2: Linha duplicada

- Tentar adicionar linha com número já existente para o mesmo cliente
- Verificar tratamento adequado

Teste 3: Chamada com origem inválida

- Tentar registrar chamada com número de origem não cadastrado
- Verificar mensagem de erro

Teste 4: Fatura sem chamadas

- Gerar fatura para cliente sem chamadas registradas
- Verificar que fatura é gerada com valor zero

9. ESTRUTURA DE DIRETÓRIOS

O projeto está organizado da seguinte forma:

tb3/ servidor-java/ ServidorAPI.java ServicoTelefonia.java Modelos.java servidor/ (diretório de classes compiladas) cliente-python/ cliente.py requirements.txt cliente-nodejs/ cliente.js package.json node_modules/ (criado após npm install) lib/ gson-2.10.1.jar compilar-servidor.bat iniciar-servidor.bat iniciar-cliente-python.bat iniciar-cliente-nodejs.bat README.md

10. CONSIDERAÇÕES TÉCNICAS

10.1. Protocolo de Comunicação

A comunicação entre clientes e servidor utiliza o protocolo HTTP/1.1 com os seguintes métodos:

- GET: Para consultas e listagens (operações idempotentes)
- POST: Para criação de recursos e operações que modificam estado
- DELETE: Para remoção de recursos

Todos os payloads utilizam formato JSON com Content-Type application/json.

10.2. Códigos de Status HTTP

O servidor retorna códigos de status HTTP padronizados:

- 200 OK: Operação de consulta bem-sucedida
- 201 Created: Recurso criado com sucesso
- 400 Bad Request: Requisição malformada
- 404 Not Found: Recurso não encontrado
- 405 Method Not Allowed: Método HTTP não suportado para o endpoint
- 500 Internal Server Error: Erro interno do servidor

10.3. Concorrência

O servidor utiliza ExecutorService com pool de threads (tamanho 10) para processar requisições concorrentes, garantindo que múltiplos clientes possam ser atendidos simultaneamente sem bloqueios.

10.4. Serialização

A serialização e deserialização de objetos Java para JSON é realizada pela biblioteca Gson, que oferece conversão automática e suporte a tipos complexos, simplificando o processo de comunicação.

11. LIMITAÇÕES E TRABALHOS FUTUROS

11.1. Limitações Conhecidas

- Os dados são armazenados em memória, sendo perdidos quando o servidor é encerrado
- Não há autenticação ou autorização implementada

- O servidor aceita requisições apenas na interface local (localhost)
- Não há limite de taxa (rate limiting) para requisições

11.2. Possíveis Melhorias

- Implementação de persistência em banco de dados
- Adição de autenticação via tokens JWT
- Suporte a HTTPS para comunicação segura
- Implementação de paginação para listagens grandes
- Adição de validação mais robusta de dados de entrada
- Implementação de logs estruturados para auditoria

12. CONCLUSÃO

O trabalho implementado demonstra com sucesso a aplicação de conceitos de sistemas distribuídos utilizando arquitetura REST sobre protocolo HTTP. A solução atende integralmente aos requisitos especificados, incluindo a implementação de servidor em Java e clientes em duas linguagens distintas (Python e JavaScript), além de demonstrar a capacidade de múltiplos clientes interagirem simultaneamente com dados compartilhados.

A escolha de tecnologias modernas e amplamente utilizadas na indústria (REST API, JSON, bibliotecas HTTP de alto nível) resulta em um sistema mais simples de implementar e manter quando comparado às abordagens de baixo nível baseadas em sockets ou RMI manual, evidenciando a evolução das tecnologias de sistemas distribuídos em direção a abstrações de mais alto nível.

Os testes realizados confirmam o funcionamento correto de todas as funcionalidades implementadas, bem como a interoperabilidade entre clientes em diferentes linguagens, validando os objetivos pedagógicos do trabalho.

13. REFERÊNCIAS

Oracle Corporation. Java Platform, Standard Edition Documentation. Disponível em: https://docs.oracle.com/javase/

Python Software Foundation. Python Documentation. Disponível em: https://docs.python.org/

Node.js Foundation. Node.js Documentation. Disponível em: https://nodejs.org/docs/

Google. Gson User Guide. Disponível em: https://github.com/google/gson

Axios. Axios Documentation. Disponível em: https://axios-http.com/docs/

Requests. Requests Documentation. Disponível em: https://requests.readthedocs.io/

FIM DO RELATÓRIO