Implementação RPC

Computação Escalável

FGV EMAp

Ana Júlia Amaro Pereira Rocha
Maria Eduarda Mesquita Magalhães
Mariana Fernandes Rocha
Paula Eduarda de Lima

Ciência de Dados e Inteligência Artificial
5º Período

Rio de Janeiro, 2025

Contents

1 Introdução

Este projeto tem como objetivo integrar um sistema de comunicação entre processos distribuídos na implementação de pipeline já confeccionada pelo grupo, utilizando o paradigma de Chamada de Procedimento Remoto (RPC - Remote Procedure Call). Através dessa abordagem, é possível invocar funções em servidores remotos como se fossem locais, abstraindo detalhes de rede e facilitando a integração entre diferentes componentes de um sistema distribuído. A implementação adotada busca demonstrar a eficiência e escalabilidade do modelo RPC no contexto de uma pipeline de processamento de dados simulados.

2 Cliente

O cliente desenvolvido para esta aplicação tem como função principal simular o envio de dados a um servidor ETL por meio do protocolo gRPC. Ele foi implementado em Python e utiliza as interfaces geradas a partir do arquivo .proto, o qual define a estrutura das mensagens e os serviços disponíveis.

O envio dos dados é realizado por meio da chamada ao método remoto EnviarDados, que recebe uma mensagem do tipo DadosRequest. Esta mensagem contém três campos: o nome da origem dos dados (como "oms", "hospital" ou "secretaria"), o nome do arquivo simulado, e uma lista de registros. Cada registro é representado por uma mensagem do tipo Linha, que encapsula, por meio de um campo oneof, exatamente um dos três formatos específicos: LinhaOMS, LinhaHospital ou LinhaSecretaria.

Os dados são gerados aleatoriamente por funções auxiliares que simulam diferentes fontes de dados. A função gerar_dados_oms cria registros com informações populacionais e sanitárias de cada ilha, incluindo número de óbitos, recuperados, vacinados e data da coleta. A função gerar_dados_hospital simula internações hospitalares, com dados como idade, sexo, sintomas e localização por CEP. Por fim, gerar_dados_secretaria produz registros com informações agregadas por região, incluindo escolaridade, população, diagnóstico e status de vacinação.

Cada tipo de dado é enviado separadamente ao servidor. O cliente imprime a resposta de confirmação para cada envio, permitindo verificar se o servidor recebeu e processou corretamente os dados. Essa abordagem permite validar o fluxo completo de comunicação e integrar a simulação de dados à pipeline de processamento distribuído.

3 Servidor

O servidor gRPC foi desenvolvido em Python com o objetivo de receber dados simulados provenientes de diferentes origens — OMS, hospitais e secretarias de saúde — e acionar uma etapa posterior de processamento. Para isso, utiliza a biblioteca grpc e implementa o serviço ETLService, conforme definido no arquivo de definição .proto.

A principal classe do servidor, PipelineServicer, herda de ETLServiceServicer e implementa o método EnviarDados. Esse método é invocado remotamente pelos clientes e recebe uma requisição do tipo DadosRequest, contendo a origem dos dados, um nome de arquivo representativo e uma lista de registros. Cada registro, recebido como uma instância do tipo Linha, é convertido para um dicionário Python (formato JSON), dependendo do campo oneof ativo (linha_oms, linha_hospital ou linha_secretaria).

Os dados recebidos são então serializados e salvos em arquivos temporários nomeados de acordo com a origem (temp_oms.json, temp_hospital.json, etc.). O servidor utiliza um dicionário compartilhado protegido por um lock (arquivos_recebidos) para armazenar o caminho dos arquivos recebidos até que os três tipos esperados estejam presentes.

Uma vez que todos os três conjuntos de dados tenham sido recebidos com sucesso, o servidor dispara a execução de um programa externo (programa.exe) por meio do módulo subprocess. Este programa é responsável por processar os dados agregados em uma pipeline de transformação mais robusta. Após a execução, o servidor limpa os arquivos temporários armazenados e está pronto para uma nova rodada de ingestão de dados.

O servidor é inicializado com uma ThreadPoolExecutor, permitindo o atendimento simultâneo de múltiplas requisições. Ele escuta na porta 50051 e permanece em execução contínua, aguardando chamadas dos clientes. Essa estrutura garante desacoplamento entre o envio de dados e o processamento, além de permitir escalabilidade futura, com eventuais adaptações para ambientes distribuídos ou balanceamento de carga.

4 Vantagens e Limitações

4.1 Vantagens

- **Eficiência de comunicação:** A utilização de gRPC com HTTP/2 e serialização binária via Protocol Buffers proporciona uma comunicação eficiente, com menor latência e uso reduzido de largura de banda, especialmente vantajoso para o envio de grandes volumes de dados.

- **Validação automática de tipos:** A definição rigorosa dos dados no arquivo `.proto` garante integridade e validação automática, reduzindo erros comuns de integração entre sistemas cliente e servidor.

- **Tratamento especializado por origem:** O uso do campo `oneof` permite distinguir facilmente os tipos de dados (OMS, hospital, secretaria), possibilitando lógica personalizada para cada caso no servidor.

- **Execução condicional da pipeline:** A estratégia de aguardar o recebimento dos três tipos de dados antes de executar a pipeline garante consistência e evita processamento parcial, assegurando que os dados estejam completos.

- **Concorrência segura:** O uso de um `lock` para controle de acesso ao dicionário de arquivos compartilhado garante segurança em ambientes com múltiplas conexões simultâneas.
- **Simplicidade de implementação:** A escrita de arquivos temporários em formato JSON é simples, de fácil debug e compatível com diversas linguagens e ferramentas externas.

4.2 Limitações

- **Dependência dos três conjuntos de dados:** A execução da pipeline depende do recebimento completo dos dados de OMS, hospital e secretaria. A ausência de qualquer um deles impede o processamento, o que pode levar à ociosidade.
- **Execução síncrona da pipeline:** A execução da pipeline bloqueia a thread responsável até sua finalização, o que pode afetar a escalabilidade e o desempenho em cenários de alta concorrência.
- **Escalabilidade limitada:** A arquitetura atual está centralizada em um único servidor, dificultando a distribuição de carga ou a tolerância a falhas, o que limita sua aplicação em ambientes produtivos de larga escala.
- **Falta de persistência intermediária:** Os dados não são armazenados em repositórios duráveis (como bancos de dados), o que pode comprometer a recuperação em caso de falha no servidor ou reinicialização.

5 Execução do sistema gRPC

A seguir, detalhamos os passos necessários para compilar, configurar e executar o sistema de comunicação gRPC desenvolvido no projeto, que permite o envio e processamento distribuído de dados simulados.

5.1 Compilação dos arquivos `.proto`

O primeiro passo consiste na geração dos stubs a partir do arquivo `etl.proto`, que define as mensagens e serviços utilizados na comunicação. A compilação deve ser feita para Python (cliente e servidor) utilizando o `protoc` com os plugins apropriados.

```
# Compilação para Python
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. etl.proto
```

5.2 Execução do servidor

Após compilar os arquivos, o servidor gRPC pode ser executado diretamente, pois escuta conexões na porta 50051 (para rodar localmente) e processa as requisições recebidas.

```
python server.py
```

O servidor aguarda conexões dos clientes e salva temporariamente os dados recebidos em arquivos locais (`temp_oms.json`, `temp_hospital.json`, `temp_secretaria.json`). Quando os três tipos de arquivos forem recebidos, ele executa o programa principal da pipeline, utilizando esses arquivos como entrada.

5.3 Execução do cliente

O cliente simula o envio de dados aleatórios para os três tipos de origem (OMS, hospital, secretaria), conectando-se ao servidor gRPC para transmitir os dados. Basta executar:

```
python client.py
```

Durante a execução, o cliente mostra a resposta do servidor para cada envio, permitindo o monitoramento do processo. Os dados enviados são gerados dinamicamente com diferentes tamanhos e conteúdos, simulando arquivos de entrada reais.

5.4 Observações

Caso o programa principal (`programa.exe`) não esteja compilado na raiz do projeto rode `make` no terminal para gerar os executáveis da pipeline.

6 Resultados

Nesta seção temos os tempos de latência com diferentes configurações de clientes. Um ponto de ressalva é que nem todo cliente consegue rodar e finalizar a pipeline, já que é preciso dados das 3 naturezas, hospital, secretaria e OMS, e a pipeline só é finalizada quando tem pelo menos um tipo de cada dado.

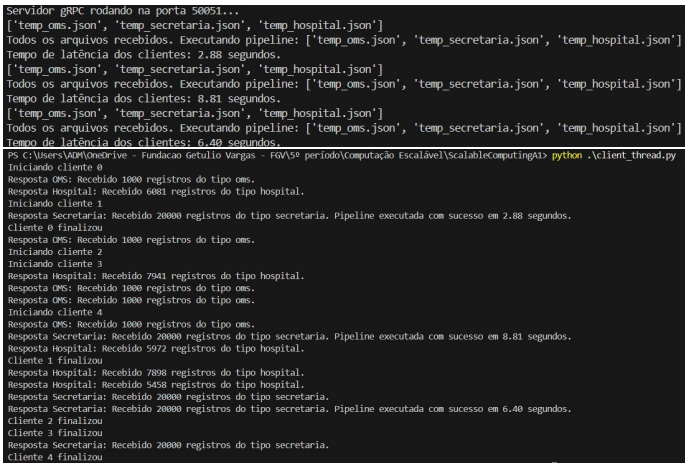

Figure 1: Server e 5 clientes

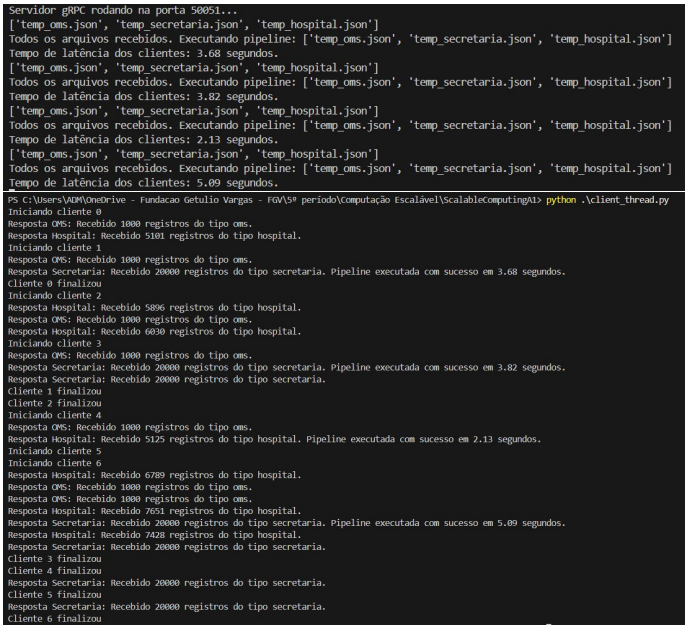

Figure 2: Server e 7 clientes

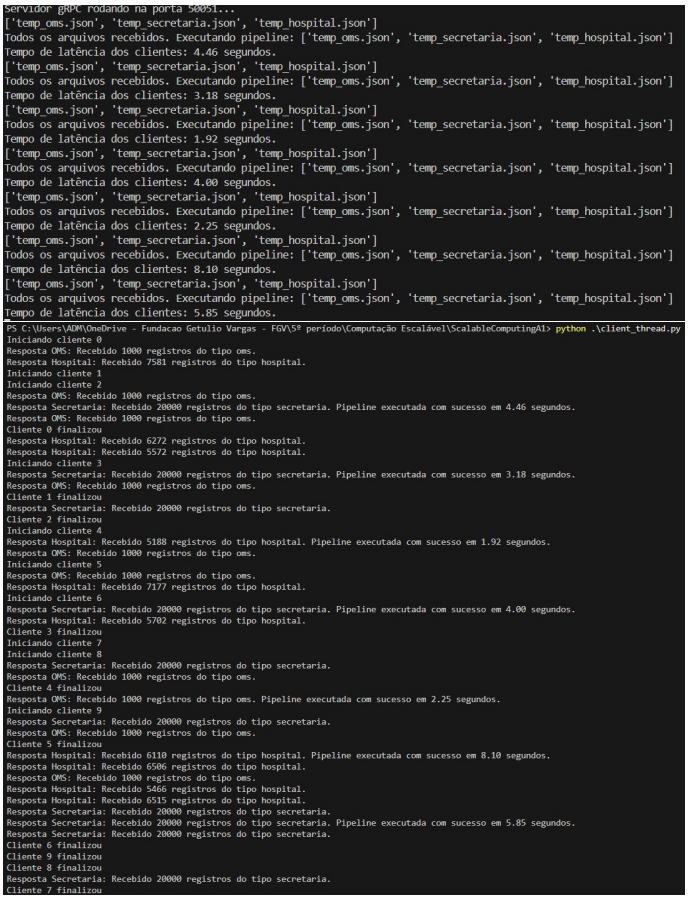

Figure 3: Server e 10 clientes