Relatório trabalho Remote Procedure Call (RPC):

Transmissão de dados via RPC para pipeline ETL

25 de Maio de 2025

Integrantes:

Daniel de Miranda Almeida

João Felipe Vilas Boas

Lívia Verly

Luís Felipe de Abreu Marciano

Paulo César Gomes Rodrigues

Visão Geral e Objetivos

Esse trabalho é complementar do trabalho feito no repositório: https://github.com/Vilasz/A1_comp_esc. O commit que marca o estado final do trabalho anterior está marcado com a tag “Versao_Final_A1”. Para observar as diferenças entre a versão entregue na A1 e a versão desse trabalho acesse o link https://github.com/Vilasz/A1_comp_esc/compare/Versao_Final_A1...main.

No trabalho anterior a comunicação entre o pipeline ETL e o simulador, que era uma das fontes de dados do ETL, era feita por meio de sistemas de arquivo e, portanto, era limitada a uma única máquina. Este trabalho propõe aprimorar a comunicação entre o simulador de fontes de dados e o ETL, substituindo o modelo atual por um mecanismo de RPC (Remote Procedure Call). Com essa mudança, será possível que instâncias do simulador, executando em máquinas remotas e conectadas via rede, enviem eventos diretamente ao ETL de forma eficiente e escalável.

Mudanças ETL pipeline A1

De estrutura, tentamos aprimorar através da re-implementação da nossa estrutura de dataframe utilizada para load e tratamento dos dados provenientes das bases. Nessa abordagem tentamos fornecer uma estrutura de dados tabular, implementando seus métodos de manipulação vetorizados. Além disso re-modularizamos nossa estrutura de ETL de forma a ter uma divisão modular mais precisa dos handlers tratadores e extratores. Demais mudanças em relação a implementação de comunicação com RPC serão discutidas na próxima seção.

Comunicação com RPC

Nossa implementação demonstra um sistema completo de transmissão de dados utilizando gRPC (Google Remote Procedure Call) em Python, especificamente projetado para simular o envio de dados de pedidos de um sistema de e-commerce. A arquitetura segue o padrão cliente-servidor, onde múltiplos clientes

podem enviar dados simultaneamente para um servidor central que persiste essas informações em um banco de dados SQLite, a fim de que o pipeline ETL faça leitura do banco de dados. Vamos passar mais detalhadamente pela estrutura a fim de compreender melhor as escolhas de projeto.

O arquivo demo.proto define o contrato de comunicação utilizando Protocol Buffers (protobuf), que é a linguagem de definição de interface do gRPC. A estrutura principal é composta por três tipos de mensagens:

- **PedidoMessage**: Esta é a mensagem principal que representa um único pedido no sistema de e-commerce, um datapoint para o nosso ETL. Ela encapsula todas as informações necessárias de um pedido de e-commerce e metadados de controle (timestamp de envio). A escolha de usar strings para datas e timestamps oferece flexibilidade na serialização, embora sacrifique um pouco a eficiência de armazenamento.
- **ListaPedidos**: Esta mensagem representa um lote de pedidos, permitindo o envio de múltiplos pedidos em uma única operação (múltiplos datapoints). Contém um identificador único (id), o identificador do cliente que está enviando (id_client) e uma lista repetida de PedidoMessage. Esta estrutura é fundamental para operações em lote, que são mais eficientes para transferência de grandes volumes de dados.

- **Ack**: Uma mensagem simples de confirmação que contém apenas uma string de mensagem. Serve como resposta padrão do servidor para confirmar o recebimento e processamento das mensagens. Essa resposta não, no nosso código, não tem função para os clientes.

O serviço GRPCDemo define quatro métodos RPC que implementam diferentes padrões de comunicação:

- **SimpleSendData**: Implementa o padrão unário simples (request-response), onde o cliente envia um único pedido e recebe uma confirmação. É o padrão mais básico e direto de comunicação.
- **StreamData**: Implementa streaming do cliente para o servidor, permitindo que o cliente envie uma sequência contínua de pedidos individuais através de um único canal de comunicação. O servidor processa cada pedido conforme recebe e retorna uma única confirmação ao final.

- **EnviarPedidosEmLote**: Utiliza o padrão unário para envio de lotes, onde o cliente envia uma única mensagem contendo múltiplos pedidos, e o servidor processa o lote completo antes de retornar uma confirmação.
- **StreamPedidosEmLote**: Combina streaming com processamento em lote, permitindo que o cliente envie uma sequência contínua de lotes de pedidos. Este é o padrão mais sofisticado, oferecendo alta throughput para sistemas de alto volume.

Indo para o servidor.py, utilizamos SQLite como solução de persistência, criando uma tabela pedidos com esquema completo que mapeia todos os campos da mensagem protobuf. A escolha do SQLite é pragmática para este exemplo, oferecendo facilidade de configuração e adequação para demonstrações, mas em produção real seria necessário considerar soluções mais robustas.

O gerenciamento de conexões de banco foi implementado de forma que cada operação utiliza sua própria conexão para evitar problemas de concorrência. O parâmetro check_same_thread=False na conexão principal permite compartilhamento seguro entre threads, enquanto operações individuais criam conexões locais para garantir isolamento.

O servidor utiliza ThreadPoolExecutor para gerenciar múltiplas conexões simultâneas de forma mais eficaz, com configuração ajustável do número de workers. Cada thread processa requisições independentemente, com isolamento garantido por conexões de banco de dados separadas. Mas mesmo com uma única thread o servidor é capaz de lidar com múltiplos clientes. Para abrir um servidor com 6 threads, por exemplo, basta rodar dentro da pasta ./gGPC/ o comando: python3 server.py 6, no terminal.

Já o client.py implementa um sistema de geração de dados sintéticos que simula pedidos realistas de um e-commerce. Os dados incluem listas de produtos tecnológicos brasileiros, centros logísticos das principais cidades do país, canais de venda diversos, e um sistema de preços.

A função generate_pedido_data() é a que efetivamente gera dados, utilizando a biblioteca Faker para gerar dados brasileiros realistas (cidades, estados) e implementando lógica de negócio como pesos diferenciados para quantidades

(favorecendo pedidos menores), cálculos de valor total automatizados, e geração de timestamps precisos.

O sistema utiliza multiprocessing para simular múltiplos clientes simultâneos, criando processos separados para cada cliente virtual. Esta abordagem oferece isolamento real entre clientes e permite teste realista de concorrência. Os clientes são capazes de utilizar qualquer um dos métodos listados acima como padrão de comunicação. Para gerar clientes ligados ao servidor enviando mensagens por meio do padrões de comunicação **EnviarPedidosEmLote** com 5 clientes, lotes com 20 datapoints e 3 lotes, por exemplo, basta rodar dentro da pasta ./gGPC/ o comando: python3 client.py --n_clients 5 --size_batch 20 --n_msgs 3.

O que será escolhido para realmente transmitir os dados para que o nosso pipeline leia foi o **EnviarPedidosEmLote.** Optamos por enviar dados em lotes múltiplas vezes, pois com a outra abordagem de streaming, tanto streaming de datapoints ou de lotes, observamos que, por conta da estrutura da função com um yield, um servidor com x threads conseguia ouvir somente de x clientes. Isso se dá porque o servidor fica preso no streaming de somente um cliente em cada thread. Com a abordagem do **EnviarPedidosEmLote** isso não acontece.

Resultados

Com esse trabalho conseguimos implementar uma lógica de comunicação utilizando RPC. O servidor está escrevendo no banco de dados os dados que são necessários para o nosso pipeline rodar. Entretanto utilizamos grande parte do nosso tempo para fazer com que o pipeline demonstrasse uma melhora relevante no tempo de processamento com mais workers. Por conta disso, não tivemos tempo hábil para fazer com que o pipeline fosse capaz de ler o banco de dados corretamente, visto que ele não poderia ler pedidos repetidos. Por conta disso, também não realizamos testes de carga, aumentando as instâncias de clientes de 0 a 20, a fim de observar o tempo médio da emissão do evento até o fim do processamento no pipeline.