Atividade de Programação: Serviço de Processamento de Arquivos com gRPC

Disciplina: Computação Distribuída

Profs. Alcides Teixeira Barboza Júnior e Mário O. Menezes - 2025

Objetivo Geral:

- Compreender e aplicar os conceitos básicos de gRPC no desenvolvimento de sistemas distribuídos.
- Implementar um serviço gRPC para processamento de arquivos, explorando diferentes tipos de RPC (unário e streaming cliente-servidor).
- Utilizar C++ para o servidor gRPC e C++ e Python para clientes gRPC.
- Integrar ferramentas de linha de comando externas (Ghostscript e ImageMagick) em um serviço gRPC.
- Implementar a solução utilizando containers (Docker ou LXD)
- Implementar logging para rastreamento e depuração de serviços distribuídos.

Objetivos Específicos:

- Definir e implementar serviços gRPC utilizando Protocol Buffers.
- Criar um servidor gRPC em C++ que ofereça serviços de processamento de arquivos.
- Desenvolver clientes gRPC em C++ e Python que consumam os serviços do servidor.
- Utilizar streaming cliente-servidor para o envio eficiente de arquivos para processamento.
- Implementar funções para compressão de PDF, conversão de PDF para TXT, conversão de formato de imagem e redimensionamento de imagem.
- Implementar o isolamento dos serviços utilizando containers
- Registrar logs de todas as operações realizadas no servidor, incluindo informações de requisição e status de conclusão.
- Tratar erros e retornar status adequados para os clientes gRPC.

Pré-requisitos:

- Conhecimentos básicos de programação em C++ e Python.
- Familiaridade com a linha de comando e sistemas Unix-like (para instalação e uso de Ghostscript e ImageMagick).
- Noções básicas de computação distribuída e RPC.

Entrega:

- Código fonte completo do servidor gRPC (C++).
- Código fonte completo dos clientes gRPC (C++ e Python).
- Arquivo protobuf (.proto) definindo os serviços e mensagens gRPC.
- Arquivo de log gerado pelo servidor durante os testes.
- Relatório sucinto descrevendo a implementação, os desafios encontrados e as decisões de design tomadas, e os prints das telas de execução.

Serviços a serem Implementados:

O servidor gRPC deverá oferecer os seguintes serviços de processamento de arquivos:

1. **Compressão de PDF (CompressPDF):**
    - **Entrada:** Arquivo PDF (streaming cliente-servidor).
    - **Saída:** Arquivo PDF comprimido (streaming servidor-cliente) ou status de sucesso/falha (unário).
    - **Processamento:** Utilizar o comando gs (Ghostscript) para comprimir o arquivo PDF recebido.
    - **Comando gs Exemplo:** gs -sDEVICE=pdfwrite -dCompatibilityLevel=1.4 -dPDFSETTINGS=/ebook -dNOPAUSE -dQUIET -dBATCH -sOutputFile=output.pdf input.pdf
2. **Conversão de PDF para TXT (ConvertToTXT):**
    - **Entrada:** Arquivo PDF (streaming cliente-servidor).
    - **Saída:** Conteúdo do arquivo TXT (streaming servidor-cliente) ou status de sucesso/falha (unário).
    - **Processamento:** Utilizar o comando pdftotext (parte do pacote Poppler, frequentemente instalado com Ghostscript ou separadamente) para converter o PDF para texto.
    - **Comando pdftotext Exemplo:** pdftotext input.pdf output.txt
3. **Conversão de Formato de Imagem (ConvertImageFormat):**
    - **Entrada:** Arquivo de imagem (streaming cliente-servidor), formato de saída desejado (string).
    - **Saída:** Arquivo de imagem convertido (streaming servidor-cliente) ou status de sucesso/falha (unário).
    - **Processamento:** Utilizar o comando convert (ImageMagick) para converter o formato da imagem.
    - **Comando convert Exemplo:** convert input.jpg output.png (onde .png é o formato de saída desejado).
4. **Redimensionamento de Imagem (ResizeImage):**
    - **Entrada:** Arquivo de imagem (streaming cliente-servidor), largura e altura desejadas (inteiros).
    - **Saída:** Arquivo de imagem redimensionado (streaming servidor-cliente) ou status de sucesso/falha (unário).
    - **Processamento:** Utilizar o comando convert (ImageMagick) para redimensionar a imagem.
    - **Comando convert Exemplo:** convert input.jpg -resize 800x600 output.jpg (onde 800x600 são as dimensões desejadas).

Log de Operações:

- O servidor deve registrar todas as requisições de serviço em um arquivo de log (ex: server.log).
- Cada entrada de log deve incluir:
    - Timestamp da requisição.
    - Nome do serviço requisitado.
    - Nome do arquivo de entrada (se aplicável).
    - Status da operação (sucesso, falha, mensagem de erro em caso de falha).

Passos para Implementação:

1. **Definição do Protobuf (.proto):**

- Defina as mensagens para requisição e resposta de cada serviço.
- Utilize streaming para envio e recebimento de arquivos.
- Inclua campos para parâmetros específicos de cada serviço (formato de saída, dimensões, etc.).
- Defina o serviço FileProcessorService com os métodos correspondentes a cada serviço (CompressPDF, ConvertToTXT, ConvertImageFormat, ResizeImage).

2. **Implementação do Servidor gRPC (C++):**
   - Crie um servidor gRPC em C++ utilizando a biblioteca gRPC.
   - Implemente a classe de serviço FileProcessorServiceImpl que herda da classe base gerada pelo protobuf.
   - Implemente cada método de serviço (CompressPDF, ConvertToTXT, ConvertImageFormat, ResizeImage).
     - Receba o arquivo como um stream de bytes do cliente.
     - Salve o arquivo recebido temporariamente no servidor.
     - Execute o comando de linha de comando apropriado (gs, pdftotext, convert) para processar o arquivo.
     - Capture a saída e o código de retorno do comando.
     - Registre a operação no arquivo de log, incluindo status e mensagens de erro se houver.
     - Retorne o arquivo processado como um stream de bytes para o cliente ou um status de sucesso/falha.
     - Limpe os arquivos temporários.
   -
   - Configure o servidor para ouvir em uma porta específica.
3. **Implementação dos Clientes gRPC (C++ e Python):**
   - Crie clientes gRPC em C++ e Python utilizando as bibliotecas gRPC correspondentes.
   - Implemente funções em cada cliente para interagir com cada serviço do servidor.
     - Peça ao usuário para selecionar o arquivo de entrada e fornecer os parâmetros necessários (formato, dimensões, etc.).
     - Abra o arquivo de entrada e envie-o como um stream de bytes para o servidor utilizando o streaming cliente-servidor do gRPC.
     - Receba a resposta do servidor (stream de bytes ou status).
     - Se receber um stream de bytes (arquivo processado), salve-o no disco.
     - Exiba mensagens de sucesso ou erro para o usuário.
   -
4. **Testes e Geração de Log:**
   - Execute o servidor gRPC.
   - Execute os clientes gRPC (C++ e Python) para testar cada serviço com diferentes arquivos de entrada e parâmetros.
   - Verifique o arquivo de log do servidor para garantir que todas as operações foram registradas corretamente.
   - Teste cenários de erro (arquivo não encontrado, formato inválido, etc.) e verifique se o servidor e os clientes lidam com os erros adequadamente.

Exemplo de Protobuf (.proto):

```
syntax = "proto3";

package file_processor;

message FileChunk {
  bytes content = 1;
}

message FileRequest {
  string file_name = 1;
  stream FileChunk file_content = 2;
  oneof parameters {
    CompressPDFRequest compress_pdf_params = 3;
    ConvertToTXTRequest convert_to_txt_params = 4;
    ConvertImageFormatRequest convert_image_format_params = 5;
    ResizeImageRequest resize_image_params = 6;
  }
}

message CompressPDFRequest {}

message ConvertToTXTRequest {}

message ConvertImageFormatRequest {
  string output_format = 1;
}

message ResizeImageRequest {
  int32 width = 1;
  int32 height = 2;
}

message FileResponse {
  string file_name = 1;
  stream FileChunk file_content = 2;
  string status_message = 3;
  bool success = 4;
}

service FileProcessorService {
  rpc CompressPDF(FileRequest) returns (FileResponse);
  rpc ConvertToTXT(FileRequest) returns (FileResponse);
  rpc ConvertImageFormat(FileRequest) returns (FileResponse);
  rpc ResizeImage(FileRequest) returns (FileResponse);
}
```

Trechos de Código de Exemplo (C++ Servidor - CompressPDF):

```
#include <fstream>
#include <iostream>
#include <memory>
#include <sstream>
#include <string>
#include <chrono>
#include <ctime>

#include <grpcpp/grpcpp.h>
#include "file_processor.grpc.pb.h" // Arquivo gerado pelo protobuf

using grpc::Server;
using grpc::ServerBuilder;
using grpc::ServerContext;
using grpc::Status;
using grpc::ServerReaderWriter;
using file_processor::FileProcessorService;
using file_processor::FileRequest;
using file_processor::FileResponse;
using file_processor::FileChunk;

class FileProcessorServiceImpl final : public FileProcessorService::Service {
public:
    Status CompressPDF(ServerContext* context, const FileRequest* request,
FileResponse* response) override {
        std::string input_file_path = "/tmp/input_" + request->file_name(); //
Arquivo temporário
        std::string output_file_path = "/tmp/output_" + request->file_name();
        std::ofstream input_file_stream(input_file_path, std::ios::binary);
        if (!input_file_stream) {
            LogError("CompressPDF", request->file_name(), "Falha ao criar
arquivo temporário de entrada.");
            response->set_success(false);
            response->set_status_message("Erro no servidor ao criar arquivo
temporário.");
            return Status::INTERNAL;
        }

        // Receber stream do cliente e salvar no arquivo temporário
        grpc::ClientReaderWriter<FileChunk, FileChunk>* stream =
grpc::ServerContext::FromServerContext(*context).ReaderWriter();
        FileChunk chunk;
        while (stream->Read(&chunk)) {
            input_file_stream.write(chunk.content().c_str(),
chunk.content().size());
        }
```

```
        input_file_stream.close();

        // Executar comando gs
        std::string command = "gs -sDEVICE=pdfwrite -dCompatibilityLevel=1.4
-dPDFSETTINGS=/ebook -dNOPAUSE -dQUIET -dBATCH -sOutputFile=" + output_file_path
+ " " + input_file_path;
        int gs_result = std::system(command.c_str());

        if (gs_result == 0) {
            LogError("CompressPDF", request->file_name(), "Compressão PDF
bem-sucedida.");
            response->set_success(true);
            response->set_file_name("compressed_" + request->file_name());

            std::ifstream output_file_stream(output_file_path,
std::ios::binary);
            if (output_file_stream) {
                while (output_file_stream.peek() != EOF) {
                    FileChunk response_chunk;
                    char buffer[1024];
                    output_file_stream.read(buffer, sizeof(buffer));
                    response_chunk.set_content(buffer,
output_file_stream.gcount());
                    stream->Write(response_chunk); // Enviar stream para o
cliente
                }
                output_file_stream.close();
            } else {
                LogError("CompressPDF", request->file_name(), "Falha ao abrir
arquivo comprimido para envio.");
                response->set_success(false);
                response->set_status_message("Erro no servidor ao abrir arquivo
comprimido.");
                return Status::INTERNAL;
            }
             stream->WritesDone();
             stream->Finish();

        } else {
            LogError("CompressPDF", request->file_name(), "Falha na compressão
PDF. Código de retorno: " + std::to_string(gs_result));
            response->set_success(false);
            response->set_status_message("Falha ao comprimir PDF.");
            return Status::INTERNAL;
        }

        std::remove(input_file_path.c_str()); // Limpar arquivos temporários
        std::remove(output_file_path.c_str());
```

```
        return Status::OK;
    }

private:
    void LogError(const std::string& service_name, const std::string& file_name,
const std::string& message) {
        auto now = std::chrono::system_clock::now();
        std::time_t now_c = std::chrono::system_clock::to_time_t(now);
        std::tm now_tm;
        localtime_r(&now_c, &now_tm);
        char timestamp[26];
        std::strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S",
&now_tm);

        std::ofstream log_file("server.log", std::ios::app);
        if (log_file.is_open()) {
            log_file << "[" << timestamp << "] ERROR - Service: " <<
service_name << ", File: " << file_name << ", Message: " << message <<
std::endl;
            log_file.close();
        } else {
            std::cerr << "Falha ao abrir arquivo de log!" << std::endl;
        }
        std::cerr << "[" << timestamp << "] ERROR - Service: " << service_name
<< ", File: " << file_name << ", Message: " << message << std::endl; // Log para
console também
    }
    // ... (Função Log para sucesso - LogSuccess, similar a LogError, mas para
logs de sucesso)
    void LogSuccess(const std::string& service_name, const std::string&
file_name, const std::string& message) {
         auto now = std::chrono::system_clock::now();
        std::time_t now_c = std::chrono::system_clock::to_time_t(now);
        std::tm now_tm;
        localtime_r(&now_c, &now_tm);
        char timestamp[26];
        std::strftime(timestamp, sizeof(timestamp), "%Y-%m-%d %H:%M:%S",
&now_tm);

        std::ofstream log_file("server.log", std::ios::app);
        if (log_file.is_open()) {
            log_file << "[" << timestamp << "] SUCCESS - Service: " <<
service_name << ", File: " << file_name << ", Message: " << message <<
std::endl;
            log_file.close();
        } else {
            std::cerr << "Falha ao abrir arquivo de log!" << std::endl;
        }
```

```
        std::cout << "[" << timestamp << "] SUCCESS - Service: " << service_name
<< ", File: " << file_name << ", Message: " << message << std::endl; // Log para
console também
    }

};

void RunServer() {
    std::string server_address("0.0.0.0:50051");
    FileProcessorServiceImpl service;

    ServerBuilder builder;
    builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
    builder.RegisterService(&service);
    std::unique_ptr<Server> server(builder.BuildAndStart());
    std::cout << "Servidor gRPC ouvindo em " << server_address << std::endl;
    server->Wait();
}

int main() {
    RunServer();
    return 0;
}
```

Trecho de Código de Exemplo (Python Cliente - CompressPDF):

```
import grpc
import file_processor_pb2
import file_processor_pb2_grpc

def compress_pdf(stub, input_file_path, output_file_path):
    def file_iterator():
        with open(input_file_path, 'rb') as f:
            while True:
                chunk = f.read(1024)
                if not chunk:
                    break
                yield file_processor_pb2.FileChunk(content=chunk)

    request = file_processor_pb2.FileRequest(
        file_name=input_file_path.split('/')[-1], # Usa o nome do arquivo como
nome no request
        file_content=file_iterator(),
        compress_pdf_params=file_processor_pb2.CompressPDFRequest()
    )

    response_stream = stub.CompressPDF(request)
```

```
    try:
        with open(output_file_path, 'wb') as output_file:
            for chunk in response_stream:
                output_file.write(chunk.content)
        print(f"PDF comprimido e salvo em: {output_file_path}")
    except grpc.RpcError as e:
        print(f"Erro ao comprimir PDF: {e.details()}")
    except Exception as e:
        print(f"Erro ao salvar arquivo comprimido: {e}")


def run_client():
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = file_processor_pb2_grpc.FileProcessorServiceStub(channel)
        input_pdf = "input.pdf" # Substitua pelo caminho do seu arquivo PDF de
teste
        output_pdf = "compressed_output.pdf"
        compress_pdf(stub, input_pdf, output_pdf)

if __name__ == '__main__':
    run_client()
```

Observações Importantes:

- **Instalação de Dependências:** Os alunos precisarão instalar as bibliotecas gRPC para C++ e Python, o compilador protobuf (protoc), Ghostscript e ImageMagick em seus ambientes de desenvolvimento.
- **Geração de Código gRPC:** Após definir o arquivo .proto, os alunos precisarão gerar o código gRPC em C++ e Python utilizando o compilador protoc com os plugins apropriados.
- **Tratamento de Erros:** A atividade deve enfatizar a importância do tratamento de erros tanto no servidor quanto nos clientes. Os exemplos fornecidos incluem um tratamento básico de erros, mas podem ser expandidos para cenários mais complexos.
- **Segurança:** Em um cenário real, seria crucial considerar a segurança, especialmente ao executar comandos de sistema e lidar com arquivos enviados por clientes. Para fins didáticos, a segurança pode ser abordada de forma mais superficial, mas é importante mencionar as preocupações.
- **Melhorias e Desafios Adicionais:**
    - Adicionar autenticação e autorização para os serviços.
    - Implementar tratamento de erros mais robusto e detalhado.
    - Utilizar um sistema de filas (ex: RabbitMQ, Redis) para enfileirar as requisições de processamento e aumentar a escalabilidade do servidor.
    - Implementar um cliente web utilizando gRPC-Web para consumir os serviços no navegador.
    - Criar uma interface gráfica para os clientes.
- **Utilização de Containers**
    - Essa é uma etapa extra neste projeto e deverá ser desenvolvida por cada grupo.
    - Todas as decisões de projeto para esta etapa deverão estar completamente documentadas, com bom detalhamento.

Solução Completa (Conceitual):

A "solução completa" para esta atividade seria um projeto bem organizado com os seguintes componentes:

- **Diretório Raiz do Projeto:**
  - proto/: Contém o arquivo file_processor.proto.
  - server_cpp/: Contém o código fonte do servidor C++ (server.cc, file_processor_service_impl.cc, CMakeLists.txt, etc.).
  - client_cpp/: Contém o código fonte do cliente C++ (client.cc, CMakeLists.txt, etc.).
  - client_python/: Contém o código fonte do cliente Python (client.py, requirements.txt, etc.).
  - scripts/: Scripts úteis para compilação, execução e testes (opcional).
  - README.md: Instruções para compilar, executar e testar a atividade.
- **Arquivos Chave:**
  - proto/file_processor.proto: Definição dos serviços e mensagens gRPC.
  - server_cpp/server.cc: Ponto de entrada do servidor gRPC em C++.
  - server_cpp/file_processor_service_impl.cc: Implementação da classe FileProcessorServiceImpl.
  - client_cpp/client.cc: Cliente gRPC em C++.
  - client_python/client.py: Cliente gRPC em Python.
  - server.log: Arquivo de log gerado pelo servidor.

Fluxo de Execução Típico:

1. **Compilar Protobuf:** Utilizar protoc para gerar código gRPC em C++ e Python a partir do arquivo file_processor.proto.
2. **Compilar Servidor C++:** Utilizar CMake (ou outro sistema de build) para compilar o servidor C++ e gerar o executável do servidor.
3. **Compilar Cliente C++:** Utilizar CMake para compilar o cliente C++ e gerar o executável do cliente.
4. **Instalar Dependências Python:** Utilizar pip install -r requirements.txt (se houver) para instalar as dependências do cliente Python.
5. **Executar Servidor:** Iniciar o executável do servidor C++.
6. **Executar Clientes:** Executar os clientes C++ e Python, fornecendo os arquivos de entrada e os parâmetros desejados.
7. **Verificar Resultados:** Verificar os arquivos de saída gerados pelos clientes e o arquivo de log do servidor para confirmar o correto funcionamento do sistema.