Engenharia de Telecomunicações
STD29006 – Sistemas Distribuídos

Prof. Emerson Ribeiro de Mello
mello@ifsc.edu.br

Laboratório 3.3: gRPC com Python 3.8

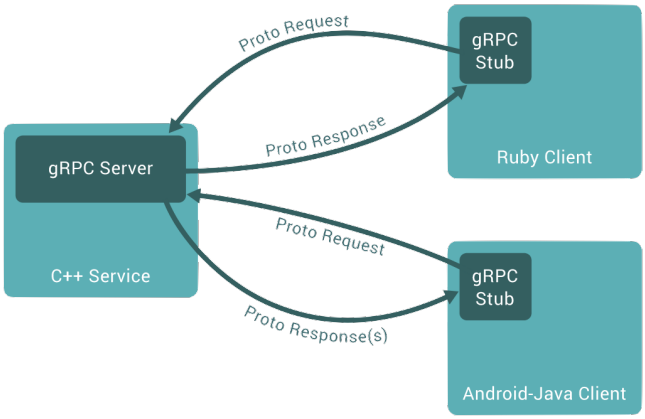
26/11/2021

Nesse laboratório será desenvolvida uma agenda de contatos, apresentada no tutorial oficial do Protocol Buffers. A aplicação cliente pode adicionar ou buscar por um contato que está armazenado na aplicação servidora. Para simplificar, o servidor irá armazenar os contatos em uma estrutura em memória e não faremos persistência em disco.

O gRPC (https://grpc.io/) consiste de um *framework* para desenvolvimento aplicações distribuídas usando chamada de procedimento remoto e faz uso, por padrão, do Protocol Buffers para serializar os dados trocados.

O gRPC oferece suporte para as seguintes linguagens de programação: C#, C++, Dart, Go, Java, Kotlin/JVM, Node, Objective-C, PHP, Python e Ruby. Sendo assim, com o gRPC é possível um cliente escrito em Ruby consiga invocar um procedimento de um código servidor, escrito em C++ e que está em execução em um computador diferente daquele onde o cliente está sendo executado.

Figura 1: Exemplo com gRPC. Fonte: https://grpc.io/



Na Figura 2 é apresentado o fluxo para desenvolver aplicações com gRPC. O primeiro passo consiste em escrever e compartilhar o arquivo `.proto` com os responsáveis por implementar o cliente e o servidor. Cada um desses deve compilar o arquivo `.proto` com o compilador do gRPC, o que resultará em um conjunto de arquivos (módulos Python). Esse arquivos deverão ser importados no código do cliente e no código do servidor.

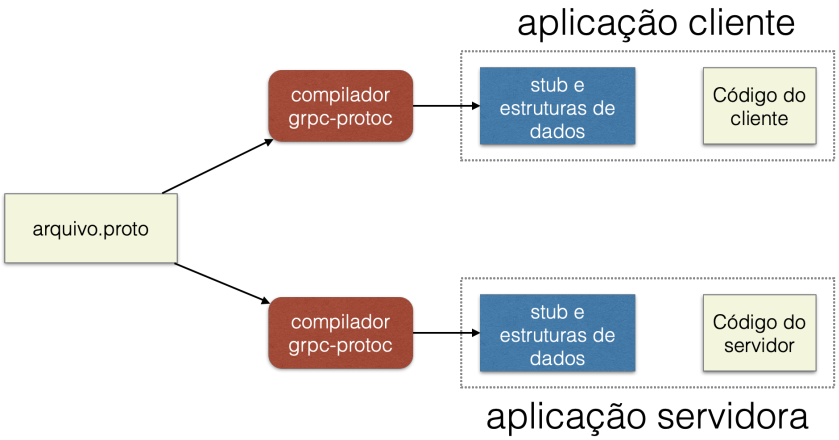
1 Preparando ambiente para desenvolvimento

Para garantir a uniformidade do ambiente, faremos uso de um contêiner Docker baseado no Ubuntu 20.04 LTS que deverá montar como volume o subdiretório `app` em `/app`. Assim, toda alteração feita no subdiretório `app` da máquina hospedeira ficará visível dentro do contêiner.

```
mkdir -p lab-03-grpc/app/protos

cd lab-03-grpc
```

Figura 2: Passos para desenvolver uma aplicação cliente ou servidora usando gRPC



Crie um arquivo chamado `Dockefile` no diretório `lab-03-grpc` e coloque o conteúdo apresentado abaixo.

```
FROM ubuntu:20.04

RUN apt-get update && apt-get install -y iproute2 iputils-ping\
    python3 python3-pip protobuf-compiler \
    && rm -rf /var/lib/apt/lists/*

RUN python3 -m pip install grpcio grpcio-tools
WORKDIR /app
```

```
# Para compilar a imagem execute o comando
docker build -t std/grpc .

# Para criar uma rede para os contêineres
docker network create --driver bridge rede-std
```

2 Criando a interface do serviço

O servidor irá expor dois métodos: `adicionar` – que recebe uma mensagem do tipo Pessoa e retorna uma mensagem do tipo Resposta; `buscar` – que recebe e retorna uma mensagem do tipo Pessoa. Dentro do diretório `lab-03-grpc/app/protos` crie um arquivo com o nome `agenda.proto` e dentro dele coloque o conteúdo apresentado na listagem abaixo.

```
syntax = "proto3";

package agenda;
import "google/protobuf/timestamp.proto";

option java_package = "engtelecom.std.agenda";
option java_outer_classname = "AgendaProtos";

// Definição das estruturas de dados
message Pessoa {
  int32  id    = 1;
  string nome  = 2;
  string email = 3;

  enum TipoTelefone {
    CELULAR     = 0;
```

```
    RESIDENCIAL = 1;
    TRABALHO    = 2;
  }

  message NumeroTelefone {
    string     numero = 1;
    TipoTelefone tipo = 2;
  }

  repeated NumeroTelefone telefones = 4;

  google.protobuf.Timestamp last_updated = 5;
}

message Resposta{
    string resultado = 1;
}

// Definição da interface do serviço (métodos que poderão ser invocados)
service Agenda {
    rpc adicionar (Pessoa) returns (Resposta) {}
    rpc buscar(Pessoa) returns (Pessoa){}
}
```

O mesmo arquivo `agenda.proto` deve ser compartilhado com desenvolvedores da aplicação servidora e da aplicação cliente. Nesse laboratório você será o desenvolvedor de ambas as aplicações. No caso, fará uso de um contêiner docker para executar o servidor e de um outro contêiner para executar o cliente. Como esses contêineres irão compartilhar o mesmo volume (diretório `app` da máquina hospedeira), então só precisará compilar o arquivo `agenda.proto` uma única vez.

Listagem 1: Executando contêiner servidor

```
# Inicie o contêiner que ficará como servidor
docker run -v `pwd`/app:/app -it --rm --network rede-std --name servidor std/grpc

# Dentro da shell do contêiner, execute o compilador protoc para gerar os módulos python
python3 -m grpc_tools.protoc -I./protos --python_out=.  --grpc_python_out=. ./protos/agenda.proto
```

Após a compilação serão gerados os arquivos `agenda_pb2.py` e `agenda_pb2_grpc.py`. Esses arquivos serão importados posteriormente quando você for escrever o código do cliente e do servidor.

3 Desenvolvendo códigos das aplicações cliente e servidora

Dentro do diretório `app` crie o arquivo com o nome `agenda_cliente.py` e coloque dentro dele o código apresentado na Listagem 2.

Listagem 2: Código da aplicação cliente

```
import logging
import sys

import grpc
import agenda_pb2
import agenda_pb2_grpc

if __name__ == "__main__":
    logging.basicConfig(format='%(asctime)s -> %(message)s',stream=sys.stdout, level=logging.DEBUG
    )
```

```
    joao = agenda_pb2.Pessoa() # criando uma pessoa
    joao.id = 1
    joao.nome = 'Joao'
    joao.email = 'joao@email.com'
    telefone = joao.telefones.add()
    telefone.numero = '(48) 3381-2800'
    telefone.tipo = agenda_pb2.Pessoa.TRABALHO

    # conectando na porta 50051 da máquina que tem o nome 'servidor'
    with grpc.insecure_channel('servidor:50051') as channel:
        stub = agenda_pb2_grpc.AgendaStub(channel)

        # adicionando um contato
        resposta = stub.adicionar(joao)
        logging.debug(f'Resposta: {resposta.resultado}')

        # buscando pelo ID de um contato
        resposta = stub.buscar(agenda_pb2.Pessoa(id=1))

        if resposta.id != -1:
            logging.debug(f'id: {resposta.id}, nome: {resposta.nome}, telefones: {resposta.
    telefones}')
        else:
            logging.debug(f'contato não encontrado')
```

Dentro do diretório `app` crie o arquivo com o nome `agenda_servidor.py` e coloque dentro dele o código apresentado na Listagem 3.

Listagem 3: Código da aplicação servidora

```
import logging
import sys
from concurrent import futures

import grpc

import agenda_pb2
import agenda_pb2_grpc


class Agenda(agenda_pb2_grpc.AgendaServicer):
    def __init__(self) -> None:
        # banco de dados será uma lista em memória
        self.database = []

    def duplicado(self,id) -> agenda_pb2.Pessoa:
        for pessoa in self.database:
            if pessoa.id == id:
                return pessoa
        return agenda_pb2.Pessoa(id=-1)

    # método que poderá ser invocado pelo cliente
    def adicionar(self, request, context):
        # Verificando se id já está no banco
        pessoa = self.duplicado(request.id)
        if pessoa.id == -1:
            self.database.append(request)
            mensagem = f'contato ({request.id}, {request.nome}) adicionado com sucesso'
        else:
            mensagem = 'id já existe no banco de dados'

        logging.debug(mensagem)
        return agenda_pb2.Resposta(resultado=mensagem)

    # método que poderá ser invocado pelo cliente
    def buscar(self, request, context):
        return self.duplicado(request.id)


if __name__ == "__main__":
    logging.basicConfig(format='%(asctime)s -> %(message)s',stream=sys.stdout, level=logging.DEBUG
    )

    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))

    agenda_pb2_grpc.add_AgendaServicer_to_server(Agenda(), server)

    server.add_insecure_port('[::]:50051')
    server.start()
    server.wait_for_termination()
```

4 Executando as aplicações

Você ainda deve estar com a shell aberta em um contêiner para o servidor que subiu quando executou o código apresentado na Listagem 1. Sendo assim, abra um outro terminal para subir um contêiner que atuará como a máquina cliente. Estando no diretório `lab-03-grpc`, execute o código abaixo:

```
docker run -v `pwd`/app:/app -it --rm  --network rede-std --name cliente std/grpc
```

Feito isso, você estará com dois contêineres em execução, um com o nome `servidor` e o outro com o nome `cliente` e ambos estarão na mesma rede docker chamada `rede-std`. Sendo assim, basta agora executar a aplicação servidora e depois executar a aplicação cliente e ver as mensagem que são impressas na tela.

```
# Dentro do contêiner do servidor
python3 agenda_servidor.py

# Dentro do contêiner do cliente
python3 agenda_cliente.py
```

5 Usando o Visual Studio Code para desenvolver com Docker

O VSCode permite usar contêiner docker como o ambiente de desenvolvimento. Assim, você consegue ter as facilidades do VSCode (como fazer uso do depurador) para escrever seu código em Python (que é o caso desse laboratório) e usufrui das ferramentas instaladas dentro do contêiner.

Para tal, instale a extensão Remote - Containers no VSCode, deixe o contêiner em execução e depois na paleta de comandos escolha *Remote-Containers: Attach to a running container*. Veja mais detalhes em https://code.visualstudio.com/docs/remote/attach-container.

6 Prática recomendada

Caso ainda não tenha familiaridade com a linguagem Python, veja pequenos exemplos que estão disponíveis em https://github.com/std29006/oo-java-e-python. E a documentação oficinal disponível em https://docs.python.org/pt-br/3/tutorial/.

Referências

- https://grpc.io/docs/what-is-grpc/core-concepts/
- https://grpc.io/docs/what-is-grpc/introduction/
- https://grpc.io/docs/protoc-installation/
- https://grpc.io/docs/languages/python/quickstart/
- https://grpc.io/docs/languages/python/basics/
- https://developers.google.com/protocol-buffers/docs/pythontutorial