Engenharia de Telecomunicações
STD29006 – Sistemas Distribuídos

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

Laboratório 3: Aplicação com Java e Python usando gRPC

27/09/2023

Nesse laboratório será desenvolvida uma agenda de contatos, apresentada no tutorial oficial do Protocol Buffers. O servidor provê dois métodos que poderão ser invocados remotamente pelos clientes: adicionar pessoa na agenda; e buscar por pessoa na agenda. Para simplificar, o servidor irá armazenar os contatos em uma estrutura em memória.

1 Introdução

O *Protocol Buffers* é adequado em cenários onde se deseja serializar dados estruturados em um formato que seja neutro à plataforma e à linguagem de programação. Segundo a documentação oficial, trata-se de uma estrutura como o JSON, porém menor, mais rápida e gera ligações nativas para diferentes linguagens de programação. A extensão padrão dos arquivos é `.proto` e dentro deles são feitas as declarações de *messages* e *services*. Na Listagem 1 é apresentada a declaração de uma mensagem com o *Protocol Buffers* versão **proto3**.

Listagem 1: Arquivo `.proto`

```
syntax = "proto3";

message Pessoa {
    optional int32 id = 1; // cada campo precisa de um identificador único
    optional string nome = 2;
    optional string email = 3;
}
```

Ao compilar um arquivo `.proto` com o compilador **protoc**[1] são gerados códigos na linguagem de programação escolhida (i.e. C++, Java, Python). Veja detalhes sobre quais arquivos serão gerados em https://developers.google.com/protocol-buffers/docs/proto3. Na Listagem 2 é apresentado um exemplo de como compilar um arquivo `.proto` para gerar saída para Java. Se desejar gerar uma saída para Python, basta usar o parâmetro `-python_out`. No exemplo assume-se que existem duas variáveis de ambiente na shell, a `$SRC_DIR` e `$DST_DIR` para indicar onde estão os arquivos com o código fonte e onde deverão ser armazenados os arquivos resultantes da compilação, respectivamente.

Listagem 2: Como compilar arquivo `.proto`

```
protoc -I=$SRC_DIR --java_out=$DST_DIR $SRC_DIR/arquivo.proto
```

Neste laboratório usaremos o *Protocol Buffers* em conjunto com o gRPC, sendo assim não usaremos diretamente o compilador `protoc`, mas sim o compilador do gRPC.

O gRPC (https://grpc.io/) consiste de um *framework* para desenvolvimento aplicações distribuídas usando chamada de procedimento remoto e faz uso do *Protocol Buffers* tanto como linguagem de

[1] Tem binários compilados para Windows, Linux e macOS em https://github.com/protocolbuffers/protobuf/releases. No Linux também é possível instalar o pacote `protobuf-compiler` por meio do apt-get

definição de interface (*Interface Definition Language* – IDL), quanto formato intercambiável para troca de mensagens.

O desenvolvimento de aplicações com gRPC segue o mesmo conceito presente desde o ONC RPC. Ou seja, deve-se definir um serviço, especificando quais métodos poderão ser invocados remotamente, bem como sua lista de parâmetros e o tipo de retorno. A definição desta interface do serviço é feita usando o *Protocol Buffers*. No lado do servidor é provida a implementação interface do serviço e este servidor ficará em execução para atender os pedidos dos clientes. No lado do cliente tem-se um *stub* (gerado pelo compilador gRPC) que permitirá ao cliente invocar os métodos remotos da mesma maneira que faria para invocar métodos locais.

Na Listagem 3 é apresentada a definição de um serviço, chamado `Greeter`, que provê um método chamado `SayHello`, o qual possui um parâmetro do tipo `HelloRequest` e retorna uma mensagem do tipo `HelloReply`.

Listagem 3: Definindo serviço gRPC

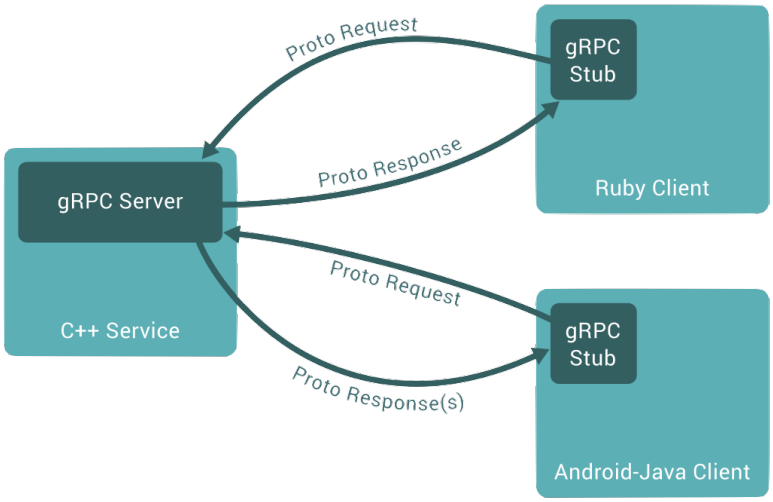
```
// Definição do serviço chamado Greeter
service Greeter {
    // Definição do método que poderá ser invocado pelos clientes
    rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// Definição da mensagem que deverá ser enviada no pedido
message HelloRequest {
    string name = 1;
}

// Definição da mensagem que será enviada como resposta
message HelloReply {
    string message = 1;
}
```

O gRPC oferece suporte para diveras linguagens de programação (i.e C#, C++, Dart, Go, Java, Kotlin/JVM, Node, Objective-C, PHP, Python e Ruby) e assim é possível ter um cliente escrito em Java que consiga invocar facilmente um procedimento remoto em um servidor que foi escrito em C++ e que está em execução em um computador diferente daquele onde o cliente está sendo executado. Veja um exemplo na Figura 1.

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

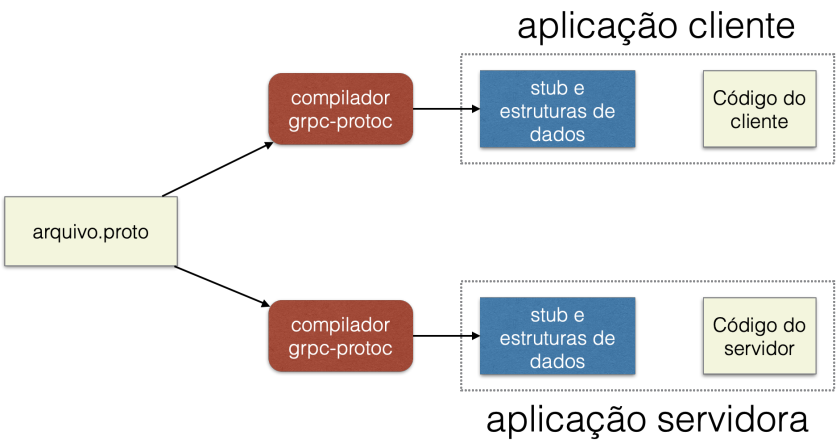


Os passos para para desenvolver aplicações com gRPC (veja Figura 2) são:

1. Escrever o arquivo `.proto` com a interface do serviço;

2. Compartilhar o arquivo `.proto` com os responsáveis pelo desenvolvimento da aplicação servidora e da aplicação cliente;
3. Cada responsável deverá compilar o arquivo `.proto`, indicando qual linguagem de programação será usada (i.e. Java, Python);
4. Importar os arquivos resultantes da compilação no projeto.

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



2 Definindo a interface do serviço agenda de contatos

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

Crie um arquivo com o nome `agenda.proto` e dentro dele coloque o conteúdo apresentado na Listagem 4.

Listagem 4: Arquivo `agenda.proto`

```
syntax = "proto3";

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

// Para indicar que a compilação deverá gerar múltiplos arquivos .java
option java_multiple_files = true;
// Para indicar o nome do pacote das classes Java que serão geradas
option java_package = "engtelecom.std.agenda";
// Para indicar o nome da classe Java que será gerada na compilação
option java_outer_classname = "AgendaProtos";

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

```
  // Enumeration para indicar os valores permitidos para indicar o tipo de telefone
  enum TipoTelefone {
    CELULAR     = 0;
    RESIDENCIAL = 1;
    TRABALHO    = 2;
  }

  // Todo telefone deve ter um número e um rótulo (tipo do telefone)
  message NumeroTelefone {
    string     numero = 1;
    TipoTelefone tipo = 2;
  }

  // para indicar que um contato poderá ter vários telefones
  repeated NumeroTelefone telefones = 4;

  // para registrar quando o contato foi criado
  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){}
}
```

Este arquivo `agenda.proto` deve ser compartilhado com desenvolvedores da aplicação servidora e da aplicação cliente. Neste laboratório, serão apresentados os passos e códigos para desenvolver cliente e servidor em Java e em Python.

3 Desenvolvimento na linguagem Java – servidor e cliente

Faremos dois projetos **gradle**, um para o servidor e um para o cliente. Na Listagem 5 são apresentados os passos para criar um diretório para cada projeto e dentro de cada diretório será criado um projeto com Gradle versão 7.X. O pré-requisito aqui é ter o JDK 17 LTS.

Listagem 5: Criando estrutura de diretórios e projeto gradle para servidor e cliente

```
mkdir -p lab03-grpc/java/servidor lab03-grpc/java/cliente

cd lab03-grpc/java/servidor

gradle init --project-name "ServidorAgendaGrpc" --package "engtelecom.std" --dsl groovy --type
    java-application --test-framework "junit-jupiter"

cd ../cliente

gradle init --project-name "ClienteAgendaGrpc" --package "engtelecom.std" --type java-application
    --dsl groovy --test-framework "junit-jupiter"
```

3.1 Alterações no arquivo `build.gradle`

Em https://github.com/google/protobuf-gradle-plugin tem um plugin para o Gradle que permite compilar os arquivos `.proto` e adicionar os arquivos Java resultantes no *classpath* do projeto, de forma que possam ser usados no código fonte em Java do projeto que você desenvolver. Na Listagem 6 é apresentado o arquivo `build.gradle` que deverá ser usado em ambos projetos (cliente e servidor).

- Os arquivo `agenda.proto` deverá ser armazenado dentro do subdiretório **src/main/proto** (você precisará criar manualmente este diretório, tanto no projeto do servidor quanto no projeto do cliente);
- Ao executar a tarefa `gradle build`, o *plugin protobuf* irá compilar automaticamente os arquivos `.proto` e os arquivos resultantes desta compilação serão armazenados em *build/generated/source/proto/main/grpc* e *build/generated/source/proto/main/java*;
- No arquivo `build.gradle` é possível indicar para que as IDEs, por meio da seção `sourceSets`, fiquem cientes das classes geradas nestes diretórios.

Listagem 6: Arquivo `build.gradle` a ser usado pelo servidor e cliente

```
plugins {
    // Apply the application plugin to add support for building a CLI application in Java.
    id 'application'
    id 'java'
    // plugin protoc
    id "com.google.protobuf" version "0.9.4"
}

repositories {
    mavenCentral()
}

def grpcVersion = '1.58.0'
def protobufVersion = '3.24.3'
def protocVersion = protobufVersion

dependencies {
    // https://mvnrepository.com/artifact/io.grpc/grpc-protobuf
    implementation "io.grpc:grpc-protobuf:${grpcVersion}"
    // https://mvnrepository.com/artifact/io.grpc/grpc-stub
    implementation "io.grpc:grpc-stub:${grpcVersion}"
    // https://mvnrepository.com/artifact/org.apache.tomcat/annotations-api
    compileOnly "org.apache.tomcat:annotations-api:6.0.53"
    // https://mvnrepository.com/artifact/io.grpc/grpc-netty-shaded
    runtimeOnly "io.grpc:grpc-netty-shaded:${grpcVersion}"

    // https://mvnrepository.com/artifact/com.google.protobuf/protobuf-java-util
    implementation "com.google.protobuf:protobuf-java-util:${protobufVersion}"

    // https://mvnrepository.com/artifact/io.grpc/grpc-testing
    testImplementation "io.grpc:grpc-testing:${grpcVersion}"

    testImplementation 'org.junit.jupiter:junit-jupiter:5.9.2'
    // This dependency is used by the application.
    implementation 'com.google.guava:guava:31.1-jre'
}
```

```
// Configuração do plugin protobuf
protobuf {
    // Baixar o protoc deste repositório
    protoc { artifact = "com.google.protobuf:protoc:${protocVersion}" }
    // Compilador gRPC para java
    plugins { grpc { artifact = "io.grpc:protoc-gen-grpc-java:${grpcVersion}" } }
    // Configuração da tarefa para o compilador gRPC (saída padrão é Java)
    generateProtoTasks {
        all()*.plugins { grpc {} }
    }
}

// Para indicar para as IDEs onde estão as classes geradas a partir do arquivo .proto
sourceSets {
    main {
        java {
            srcDirs 'build/generated/source/proto/main/grpc'
            srcDirs 'build/generated/source/proto/main/java'
        }
    }
}

application {
    // Define the main class for the application.
    mainClass = 'engtelecom.std.App'
}

tasks.named('test') {
    // Use JUnit Platform for unit tests.
    useJUnitPlatform()
}
```

3.2 Implementando o servidor

Iremos trabalhar agora no projeto gradle do servidor, criado no diretório `lab03-grpc/java/servidor`. Com o projeto gradle criado, com as alterações aplicadas no arquivo `build.gradle`, para ficar igual ao apresentado na Listagem 6, e com o arquivo `agenda.proto` (veja Listagem 4) dentro do subdiretório *src/main/proto*, chegou a hora de codificar o servidor. Teremos que fazer os seguintes passos (na sequência apresentada):

1. Compilar o arquivo `agenda.proto` com o gRPC/protoc para gerar as classes Java. Para isto, basta executar a tarefa gradle chamada `generateProto`

   ```
   ./gradlew generateProto
   ```

2. Codificar a classe Java que fará a implementação da interface do serviço descrita no arquivo `agenda.proto`. No caso, esta classe terá o nome `AgendaImpl` e irá estender a classe `AgendaGrpc.AgendaImplBase`, que foi gerada pelo compilador gRPC (veja Listagem 7);

   - Esta classe deverá prover implementação para os métodos herdados (`adicionar` e `buscar`)

3. Codificar a classe que colocará em execução o *daemon* do processo servidor e registrará o serviço da AgendaImpl para que possa ser consumido pelos clientes (veja Listagem 8).

Listagem 7: Arquivo `AgendaImpl.java` na aplicação servidora

```java
package engtelecom.std;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Logger;

import engtelecom.std.agenda.Pessoa;
import engtelecom.std.agenda.Resposta;
import io.grpc.stub.StreamObserver;
import engtelecom.std.agenda.AgendaGrpc.AgendaImplBase;

public class AgendaImpl extends AgendaImplBase {

    // Serviço de log para registrar as mensagens de depuração, informação, erro, etc.
    private static final Logger logger = Logger.getLogger(AgendaImpl.class.getName());

    // Banco de dados para armazenar todos os contatos
    private Map<Integer, Pessoa> agenda;

    public AgendaImpl() {
        this.agenda = new HashMap<>();
    }

    @Override
    public void adicionar(Pessoa request, StreamObserver<Resposta> responseObserver) {
        String mensagem = "id do contato já existe no banco de dados";
        if (!this.agenda.containsKey(request.getId())) {
            this.agenda.put(request.getId(), request);
            mensagem = "Contato com o id " + request.getId() + " foi adicionado com sucesso";
        }
        logger.info(mensagem);
        // Padrão de projeto Builder. Veja mais em https://java-design-patterns.com/patterns/builder/
        Resposta resposta = Resposta.newBuilder().setResultado(mensagem).build();
        responseObserver.onNext(resposta);
        responseObserver.onCompleted();
    }

    @Override
    public void buscar(Pessoa request, StreamObserver<Pessoa> responseObserver) {
        Pessoa resposta = this.agenda.get(request.getId());
        logger.info("Buscar... " + resposta);
        responseObserver.onNext(resposta);
        responseObserver.onCompleted();
    }
}
```

Listagem 8: Arquivo `App.java` na aplicação servidora

```java
package engtelecom.std;

import io.grpc.Server;
import io.grpc.ServerBuilder;

import java.util.logging.Logger;

public class App {

    // Serviço de log para registrar as mensagens de depuração, informação, erro, etc.
    private static final Logger logger = Logger.getLogger(App.class.getName());

    public static void main(String[] args) throws Exception {

        logger.info("Iniciando o servidor");

        // Padrão de projeto Builder. Veja mais em https://java-design-patterns.com/patterns/builder/
        // Iniciando o servidor com a implementação da AgendaImpl
        Server servidor = ServerBuilder.forPort(50051)
                .addService(new AgendaImpl())
                .build()
                .start();
```

```
        // Para finalizar o servidor quando a JVM for finalizada
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.err.println("servidor gRPC sendo desligado pois a JVM está sendo desligada");
            servidor.shutdown();
            System.err.println("Servidor parado com sucesso");
        }));

        // Para Finalizar o servidor
        servidor.awaitTermination();
    }
}
```

3.3 Implementando o cliente

Iremos trabalhar agora no projeto gradle do cliente, criado no diretório `lab03-grpc/java/cliente`. Com o projeto gradle criado, com as alterações aplicadas no arquivo `build.gradle`, para ficar igual ao apresentado na Listagem 6, e com o arquivo `agenda.proto` (veja Listagem 4) dentro do subdiretório *src/main/proto*, chegou a hora de codificar o cliente. Teremos que fazer os seguintes passos (na sequência apresentada):

1. Compilar o arquivo `agenda.proto` com o gRPC/protoc para gerar as classes Java. Para isto, basta executar a tarefa gradle chamada `generateProto`

   ```
   ./gradlew generateProto
   ```

2. Codificar a classe App que fará requisições ao serviço `Agenda` (veja Listagem 9).

Listagem 9: Arquivo `App.java` na aplicação cliente

```
package engtelecom.std;

import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;

import engtelecom.std.agenda.AgendaGrpc;
import engtelecom.std.agenda.Pessoa;
import engtelecom.std.agenda.Pessoa.NumeroTelefone;
import engtelecom.std.agenda.Pessoa.TipoTelefone;
import io.grpc.ManagedChannelBuilder;

public class App {

    private static final Logger logger = Logger.getLogger(App.class.getName());

    public static void main(String[] args) throws Exception {

        // Por padrão o gRPC sempre será sobre TLS, como não criamos um certificado digital, forçamos aqui nã
    o usar TLS
        var channel = ManagedChannelBuilder.forTarget("localhost:50051").usePlaintext().build();

        // Criando uma pessoa usando o padrão de projeto Builder
        var juca = Pessoa.newBuilder().setNome("Juca")
        .setEmail("juca@email.com")
        .setId(1)
        .addTelefones(NumeroTelefone.newBuilder().setNumero("48 3381-2800").setTipo(TipoTelefone.TRABALHO).
    build()).build();

        logger.info("Adicionando uma pessoa na agenda de contatos no servidor");
        var agendaBlockingStub = AgendaGrpc.newBlockingStub(channel);
        agendaBlockingStub.adicionar(juca);
        logger.info("Pessoa adicionada");

        logger.info("Buscando por uma pessoa na agenda de contatos");
        var resultado = agendaBlockingStub.buscar(juca);
```

```
        logger.info("Dados da pessoa retornada pelo servidor: " + resultado);

        logger.info("Finalizando...");
        channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
    }
}
```

3.4 Executando as aplicações cliente e servidor

```
# Para executar o servidor, abra um terminal, entre no diretório lab03-grpc/java/servidor e execute
./gradlew --console=plain -q run

# Para executar o cliente, abra outro terminal, entre no diretório lab03-grpc/java/cliente e execute
./gradlew --console=plain -q run
```

4 Desenvolvimento na linguagem Python3 - servidor e cliente

Nesta seção serão apresentados os passos para desenvolver o servidor e cliente para a mesma agenda de contatos que fora desenvolvido em Java na Subseção 3.2. Nesta seção será feito uso do mesmo arquivo `agenda.proto` que é apresentado na Listagem 4.

Em projetos com Python uma boa prática é fazer uso do `Virtualenv`, pois tem-se uma maneira fácil para gerenciar pacotes Python em diferentes projetos. Assim, evita-se instalar pacotes python de forma global no seu sistema operacional. Sendo assim, iremos criar um ambiente virtual python, que será compartilhado pelo aplicativo cliente e aplicativo servidor, e iremos instalar (usando a ferramenta pip) os pacotes `grpcio` e `grpcio-tools`, necessárias para compilar o arquivo `.proto` e para desenvolver as aplicações.

```
mkdir -p lab03-grpc/app-python

cd lab03-grpc/app-python

# Criando o ambiente virtual (só é necessário fazer isto uma única vez, porém este diretório nunca
     deveria ser controlado pelo git)
python3 -m venv venv
# Carregando o ambiente virtual (é necessário fazer isso sempre que abrir um novo terminal)
source venv/bin/activate

# instalando os pacotes grpc (só é necessário fazer isto uma única vez)
python -m pip install grpcio grpcio-tools

# Compilando o arquivo agenda.proto (é necessário que copie o arquivo para dentro do diretório
    lab03-grpc/app-python)
python3 -m grpc_tools.protoc -I. --python_out=.  --grpc_python_out=. agenda.proto
```

Crie o arquivo com o nome `agenda_cliente.py` e coloque dentro dele o código apresentado na Listagem 10.

Listagem 10: Aplicação cliente desenvolvida em Python

```
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('localhost: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')
```

Crie o arquivo com o nome `agenda_servidor.py` e coloque dentro dele o código apresentado na Listagem 11.

Listagem 11: Aplicação servidora desenvolvida em Python

```
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()
```

4.1 Executando as aplicações

Tanto o cliente quanto o servidor desenvolvidos em Python serão capazes de interagir com o cliente e servidor desenvolvidos em Java. Abaixo as instruções para executar o cliente e o servidor (é necessário que tenha carregado o virtualenv).

```
# Abra um terminal, carregue o virtualenv (i.e source venv/bin/activate) e execute
python3 agenda_servidor.py

# Abra outro terminal, carregue o virtualenv (i.e source venv/bin/activate) e execute
python3 agenda_cliente.py
```

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/java/quickstart/
- https://grpc.io/docs/languages/java/basics/
- https://github.com/google/protobuf-gradle-plugin
- https://grpc.io/docs/languages/python/quickstart/
- https://grpc.io/docs/languages/python/basics/
- https://developers.google.com/protocol-buffers/
- https://developers.google.com/protocol-buffers/docs/javatutorial
- https://developers.google.com/protocol-buffers/docs/pythontutorial
- https://developers.google.com/maps-booking/legacy/booking-server-code-samples/gRPC-v2...
- https://github.com/protocolbuffers/protobuf/releases