Sistemas Distribuídos

Aula Prática - RPC

Prof. Rodrigo Campiolo

UTFPR - Universidade Tecnológica Federal do Paraná

27 de abril de 2022

Objetivos

- Apresentar conceitos básicos para RPC usando gRPC
- Desenvolver uma aplicação usando gRPC.

gRPC: Características

- Código aberto e gratuito.
- Usa protocol buffers para serialização de dados.
- Bibliotecas para diversas linguagens.
- Suporte à autenticação, balanceamento de carga, monitoramento.
- Possibilita conectar dispositivos, aplicações móveis e navegadores Web a serviços de backend.

gRPC: Conceitos

- Definição de serviços
  - Unary RPC.
  - Server Streaming RPC.
  - Client Streaming RPC.
  - Bidirecional Streaming RPC.
- RPC síncrono e assíncrono.
- Deadlines/Timeouts.
- Cancelamento RPC.
- Channels (conexões para o servidor gRPC).

Prática: RPC

Materiais

- Python 3
  - grpcio: pacote para gRPC em Python.
  - grpcio-tools: gerador de código Protocol Buffer para gRPC.
  - protobuf: protocol buffers (formato para representação externa de dados) .
- Java JDK
  - grpc-netty: implementação de transporte baseado no Netty.
  - grpc-protobuf: implementação para gRPC sobre Protocol Buffers para gRPC.
  - grpc-stub: pacote para a camada de *stub*.
- Maven
  - os-maven-plugin
  - protobuf-maven-plugin: gera código-fonte Java a partir do compiladorr Protocol Buffer (protoc).

Prática: RPC

Instalação e Template

- Instalar Python 3 e Java JDK.
- Instalar bibliotecas Python e dependências:
  `pip3 install grpcio grpcio-tools protobuf`
- Obter o projeto Python e o projeto Java (com Maven) no Moodle:
  `Exemplos` → `Exemplo gRPC`

Especificando o serviço

- O serviço e as estruturas de entrada e saída devem ser especificadas no **.proto**. Veja o exemplo básico **Greeter**.

```
1  // versao do Protocol Buffer
2  syntax = "proto3";
3
4  // configuracoes especificas de linguagem
5  option java_multiple_files = true;
6  option java_outer_classname = "HelloWorldProto";
7
8  // especificacao da interface de servico
9  service Greeter {
10   rpc SayHello (HelloRequest) returns (HelloReply) {}
11 }
12
13 // requisicao com o nome do cliente
14 message HelloRequest {
15   string name = 1;
16 }
17
18 // resposta com as saudacoes do servidor
19 message HelloReply {
20   string message = 1;
21 }
```

Gerando o Stub e a RED

- Acessar a pasta do projeto Python e executar:

```
python3 -m grpc_tools.protoc -I.
        --python_out=. --grpc_python_out=.
        helloworld.proto
```

- Abrir o projeto Java em IDE (Netbeans) e compilar[1]. Com o JDK 11, ocorre um erro no arquivo gerado `GreeterGrpc.java`. Deve-se comentar a anotação `@javax.annotation.Generated` ou corrigir a anotação `@javax.annotation.processing.Generated`.[2]

[1] Pode-se compilar com Maven CLI: `mvn compile`

[2] Esse problema pode ter sido corrigido em versões mais atuais do compilador gRPC.

Prática: RPC

Implementando a interface remota

- Herdar da classe gerada que define o serviço.
- Sobrescrever os métodos especificados na interface de serviço.

Prática: RPC

- Em Python:

```
import helloworld_pb2
import helloworld_pb2_grpc

class Greeter(helloworld_pb2_grpc.GreeterServicer):

    def SayHello(self, request, context):
        return helloworld_pb2.HelloReply(message='Hello, %s!' % request.name)
```

Prática: RPC

- Em Java:

```
import io.grpc.stub.StreamObserver;

public class GreeterImpl extends GreeterGrpc.GreeterImplBase {

    @Override
    public void sayHello(HelloRequest request,
                         StreamObserver<HelloReply> responseObserver) {
        System.out.println("Recebido: " + request.getName());
        HelloReply response = HelloReply.newBuilder()
                .setMessage("Olá, recebi sua mensagem.")
                .build();

        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
}
```

Prática: RPC

Implementando o servidor

- Configurar o servidor.
- Configurar o canal de comunicação (tipo de canal, endereço e porta servidor).
- Adicionar o serviço ao servidor.
- Iniciar o servidor.
- Manter o servidor em execução.

Prática: RPC

- Em Python:

```
import grpc
import helloworld_pb2

def server():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    server.add_insecure_port('[::]:7777')
    helloworld_pb2_grpc.add_GreeterServicer_to_server(Greeter(), server)
    server.start()
    print('Servidor iniciado.')
    try:
        while True:
            time.sleep(3600)
    except KeyboardInterrupt:
        server.stop(0)

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

Prática: RPC

- Em Java:

```
import io.grpc.ServerBuilder;
import java.io.IOException;

public class Server {
    public static void main(String[] args) {
        io.grpc.Server server = ServerBuilder
                .forPort(7777)
                .addService(new GreeterImpl())
                .build();

        try {
            server.start();
            System.out.println("Servidor iniciado.");
            server.awaitTermination();
        } catch (IOException | InterruptedException e) {
            System.err.println("Erro: " + e);
        }
    }
}
```

Prática: RPC

Implementando o cliente

- Configurar o canal de comunicação.
- Criar um **stub** para o serviço.
- Realizar a chamada remota.

Prática: RPC

- Em Python:

```
import grpc
import helloworld_pb2
import helloworld_pb2_grpc

def client():
    channel = grpc.insecure_channel('localhost:7777')
    stub = helloworld_pb2_grpc.GreeterStub(channel)

    #chamada remota
    response = stub.SayHello(helloworld_pb2.HelloRequest(name='Zoro'))

    print("Recebido: " + response.message)

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

Prática: RPC

- Em Java:

```
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;

public class Client {
    public static void main(String[] args) {
        ManagedChannel channel = ManagedChannelBuilder
                .forAddress("localhost", 7777)
                .usePlaintext()
                .build();

        GreeterGrpc.GreeterBlockingStub stub =
                GreeterGrpc.newBlockingStub(channel);

        HelloRequest request = HelloRequest
                .newBuilder()
                .setName("Luffy")
                .build();

        //chamada remota
        HelloReply reply = stub.sayHello(request);

        System.out.println("Resposta: " + reply.getMessage());
        channel.shutdown();
    }
}
```

Prática: RPC

Executando a aplicação

- Há quatro formas de execução:
  1. cliente e servidor em Python.
  2. cliente e servidor em Java.
  3. cliente em Python e servidor em Java.
  4. cliente em Java e servidor em Python.
- Em Python, executar:

```
python3 server.py
python3 client.py
```

- Em Java executar via IDE ou com Maven CLI:

```
mvn exec:java -D"exec.mainClass"="Server"
mvn exec:java -D"exec.mainClass"="Client"
```

Considerações finais

- O programador deve se preocupar em definir uma interface de serviço e implementar o serviço.
- A interface de serviço deve ser bem definida.
- O cliente e o servidor são simples de implementar.
- A comunicação e a representação externa de dados são transparentes para o programador.

Prática: RPC

Atividades

1. Faça uma implementação usando gRPC para um serviço remoto de gerenciamento de músicas/livros que possibilita adicionar, remover e consultar músicas/livros.
   Opcional: O servidor e o cliente devem ser implementados em linguagens de programação distintas.

Referências Bibliográficas

Referências

- gRPC. Disponível em `https://grpc.io/`. Acessado em 27/08/2020.
- A basic tutorial introduction to gRPC in Python. Disponível em `https://grpc.io/docs/languages/python/basics/`. Acessado em 27/08/2020.
- A basic tutorial introduction to gRPC in Java. Disponível em `https://grpc.io/docs/languages/java/basics/`. Acessado em 27/08/2020.
- Quick Start (Python). Disponível em `https://grpc.io/docs/languages/python/quickstart/`. Acessado em 27/08/2020.