gRPC

Programación Distribuida y Tiempo Real

¿Qué es gRPC?

gRPC

Definición

A language-neutral, platform-neutral remote procedure call (RPC) framework and toolset developed at Google

gRPC

Conceptos generales

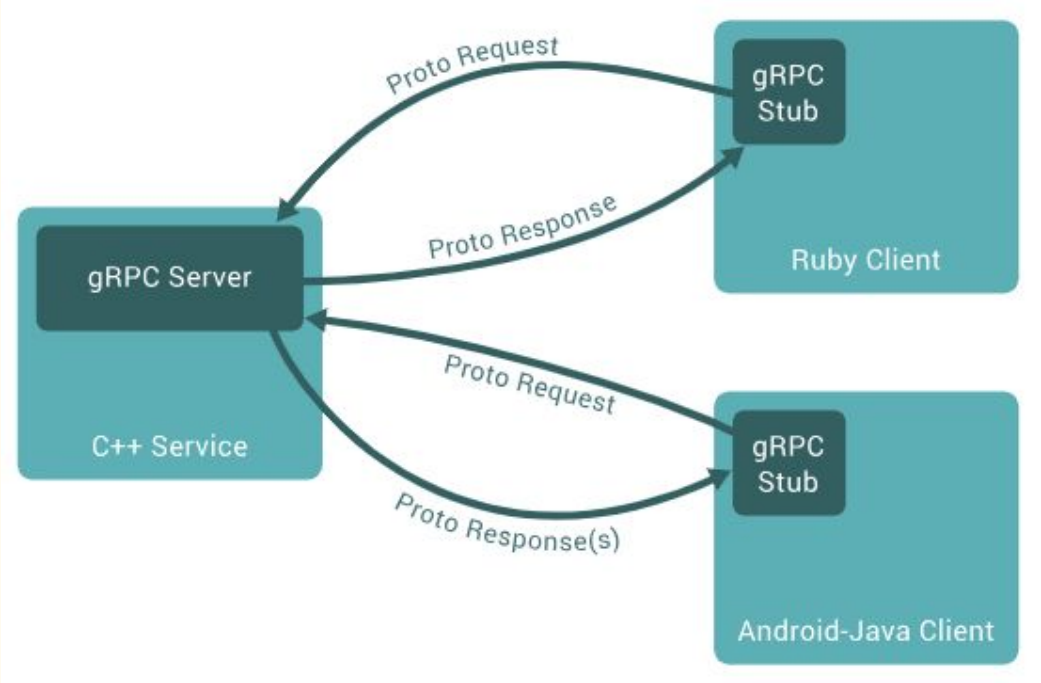
- Permite definir un servicio utilizando Protocol Buffers (protobuf)
- Permite generar clientes idiomáticos y servidores stubs desde la definición del servicio en varios lenguajes (Android Java, C#/.Net, Kotlin/JVM, Go, Node.js, PHP, Python, etc)

gRPC

Conceptos generales

- Interfaces RPC: Definen los métodos que un cliente puede invocar en el servidor.
- HTTP/2: Mayor eficiencia en la comunicación.

Esquema cliente/servidor



Workflow gRPC

gRPC Workflow

en PDyTR

- Utilizar Maven
  - Generador de los proyectos
  - Instalador de paquetes
  - Compilador
  - Ejecutor
- Con el proyecto generado, se crea el `.proto`

gRPC Implementación

.proto

```
syntax = "proto3";

package app.pdytr;

option java_package = "app.pdytr";
option java_outer_classname = "BookServiceProto";

message Book {
  int32 id = 1;
  string title = 2;
  string author = 3;
}

message AddBookRequest {
  Book book = 1;
}

message AddBookResponse {
  Book book = 1;
}

message ListBooksRequest {}

message ListBooksResponse {
  repeated Book books = 1;
}

service BookService {
  rpc AddBook(AddBookRequest) returns (AddBookResponse);
  rpc ListBooks(ListBooksRequest) returns (ListBooksResponse);
}
```

Servidor

Implementación del servidor usando Java y las clases generadas a partir del .proto

```java
public class BookServer {
    Run | Debug
    public static void main(String[] args) throws IOException, In
        Server server = ServerBuilder.forPort(port:8080)
                .addService(new BookServiceImpl())
                .build();

        System.out.println("Starting server...");
        server.start();
        System.out.println("Server started on port 8080");

        server.awaitTermination();
    }
}
```

Cliente

Ejemplo de implementación del cliente

```java
public class BookClient {
    Run | Debug
    public static void main(String[] args) {
        ManagedChannel channel = ManagedChannelBuilder.forAddress(name:"localhost", port:8080)
                .usePlaintext()
                .build();

        BookServiceGrpc.BookServiceBlockingStub stub = BookServiceGrpc.newBlockingStub(channel);

        ListBooksRequest listBooksRequest = ListBooksRequest.newBuilder().build();
        ListBooksResponse listBooksResponse = stub.listBooks(listBooksRequest);

        System.out.println("List of books:");
        listBooksResponse.getBooksList().forEach(System.out::println);

        channel.shutdown();
    }
}
```

gRPC vs Rest

gRPC vs REST

gRPC

- **Protocolo**: HTTP/2 (soporte para multiplexación y streaming bidireccional).
- **Formato de datos**: Protocol Buffers (binario, más compacto y eficiente).
- **Operaciones**: RPC (Remote Procedure Calls), llamadas de funciones directas.
- **Ventajas**:
    - Alto rendimiento y baja latencia.
    - Soporte nativo para streaming bidireccional.
    - Eficiente en la comunicación entre microservicios.
- **Desventajas**:
    - Requiere más configuración inicial (Protocol Buffers).
    - Menos adecuado para APIs públicas o sistemas con clientes variados.

Rest

- **Protocolo**: HTTP/1.1 (o HTTP/2 en algunos casos).
- **Formato de datos**: JSON o XML (basados en texto, fácil de leer, pero menos eficiente).
- **Operaciones**: CRUD (GET, POST, PUT, DELETE) alineadas con los verbos HTTP.

Ventajas:

- Sencillo de implementar y ampliamente adoptado.
- Compatibilidad multiplataforma, ideal para APIs públicas.
- Flexible en el formato de datos.

Desventajas:

- Mayor overhead debido a JSON y HTTP/1.1.
- No tiene soporte nativo para streaming bidireccional.
- Menos eficiente para microservicios y aplicaciones de alto rendimiento.

gRPC vs REST: ¿Cuándo usar cada una?

Usar gRPC cuando:

- Se requiere **alto rendimiento** y baja latencia, especialmente en **microservicios**.
- La aplicación necesita **streaming bidireccional** (comunicaciones en tiempo real, sistemas IoT, streaming de datos).
- Se busca una **comunicación eficiente** entre servicios con mensajes compactos.
- Se maneja un **ecosistema controlado** (clientes conocidos, infraestructura interna).

Usar REST cuando:

- Se necesita **compatibilidad multiplataforma** (web, móviles, sistemas de terceros).
- Las APIs son **públicas** o expuestas a un amplio rango de clientes y dispositivos.
- **Simplicidad** y facilidad de implementación son importantes.
- Las operaciones son principalmente CRUD y no se necesita streaming bidireccional.

¡Muchas gracias!

¿Preguntas?

Links útiles

Google mini lab in GCP with gRPC and Java

- Building a gRPC service with Java

gRPC Concepts

- https://grpc.io/docs/what-is-grpc/core-concepts/