gRPC y Protocol Buffers

Aplicacions i Serveis Web

Adrià Abad, David Jiménez, Thiago Mulero, Raül Sampietro

Introducción

Protocol Buffers proporcionan un mecanismo extensible, independiente del lenguaje y de la plataforma, para serializar datos estructurados de manera compatible con versiones anteriores y posteriores.

google **R**emote **P**rocedure **C**alls es un sistema de llamada a procedimiento remoto, open source, que puede ejecutarse en cualquier entorno.

De forma predeterminada, gRPC utiliza Protocol Buffers como núcleo tecnológico para la gestión y estructura de los datos.

Evolución Histórica: Protocol Buffers

Versión 2.6.0

Primera versión open source de proto2.

Agosto 2014

Julio 2016

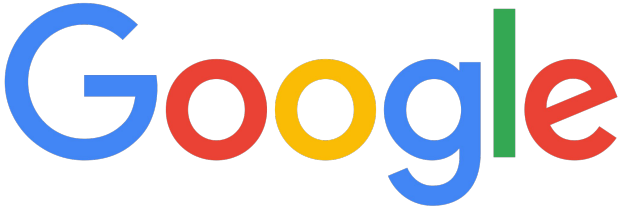
Versión 3.0.0

Primera versión estable de proto3.

Versión 3.19.4

Última versión actualizada de proto3.

Enero 2022



Evolución Histórica: gRPC

Versión 0.5.0

Creación de gRPC en formato open source.

Febrero 2015

Agosto 2016

Versión 1.0.0

Primera versión estable de gRPC.

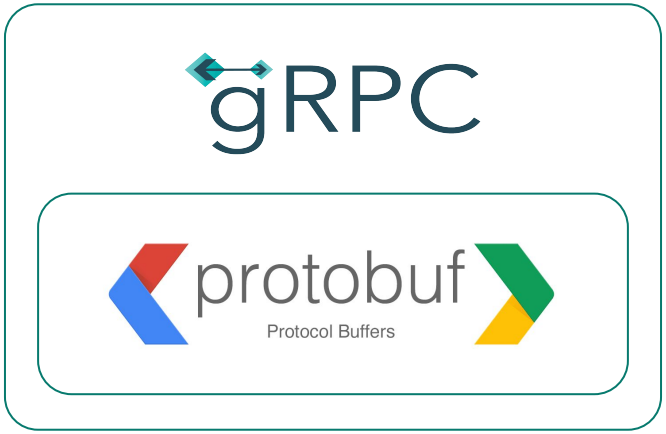
Versión 1.44.0

Última versión actualizada de gRPC.

Febrero 2022

GitHub

Protocol Buffers y gRPC



Archivo .proto

El **formato** de los mensajes Protocol Buffers se definen en un archivo **.proto**.

También se pueden definir las **reglas** del protocolo.

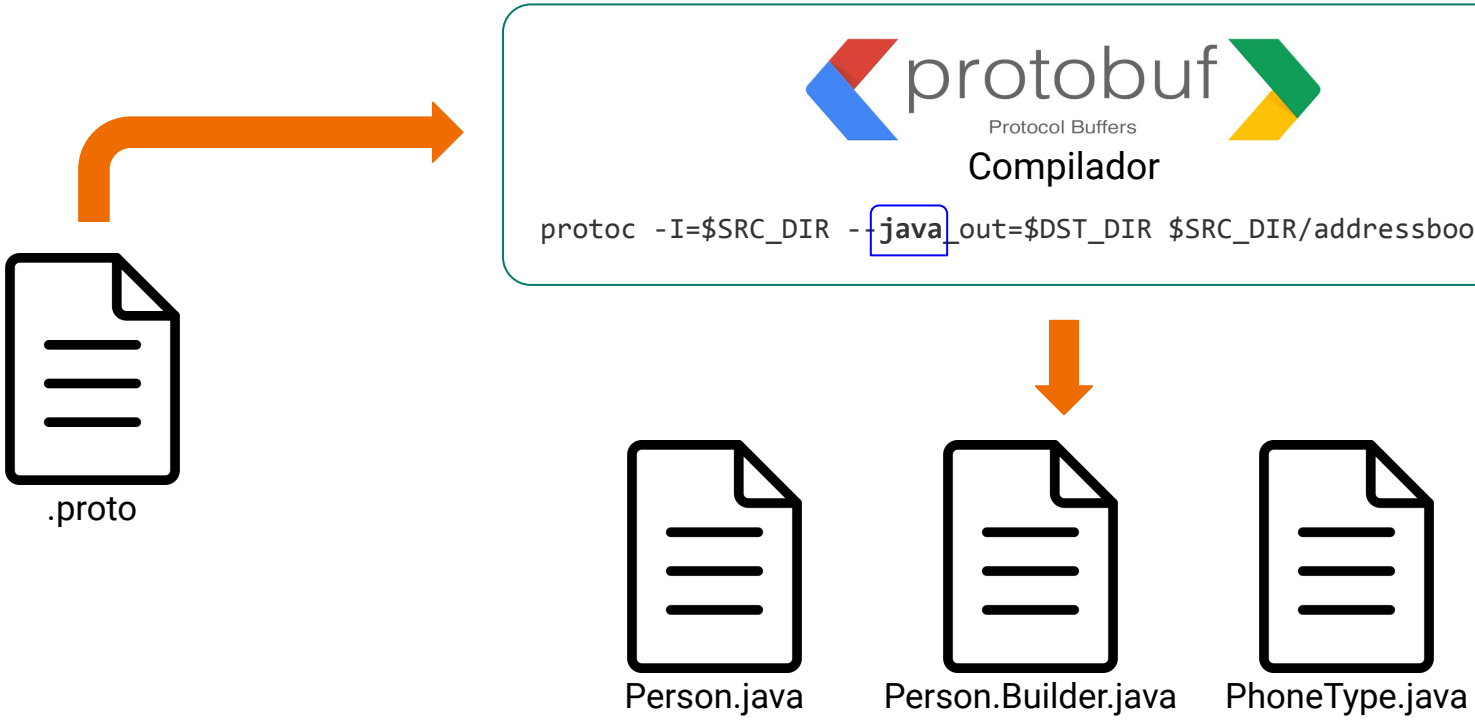
addressbook.proto

Mensaje **Person**
- Enum **PhoneType**
- Mensaje **PhoneNumber**

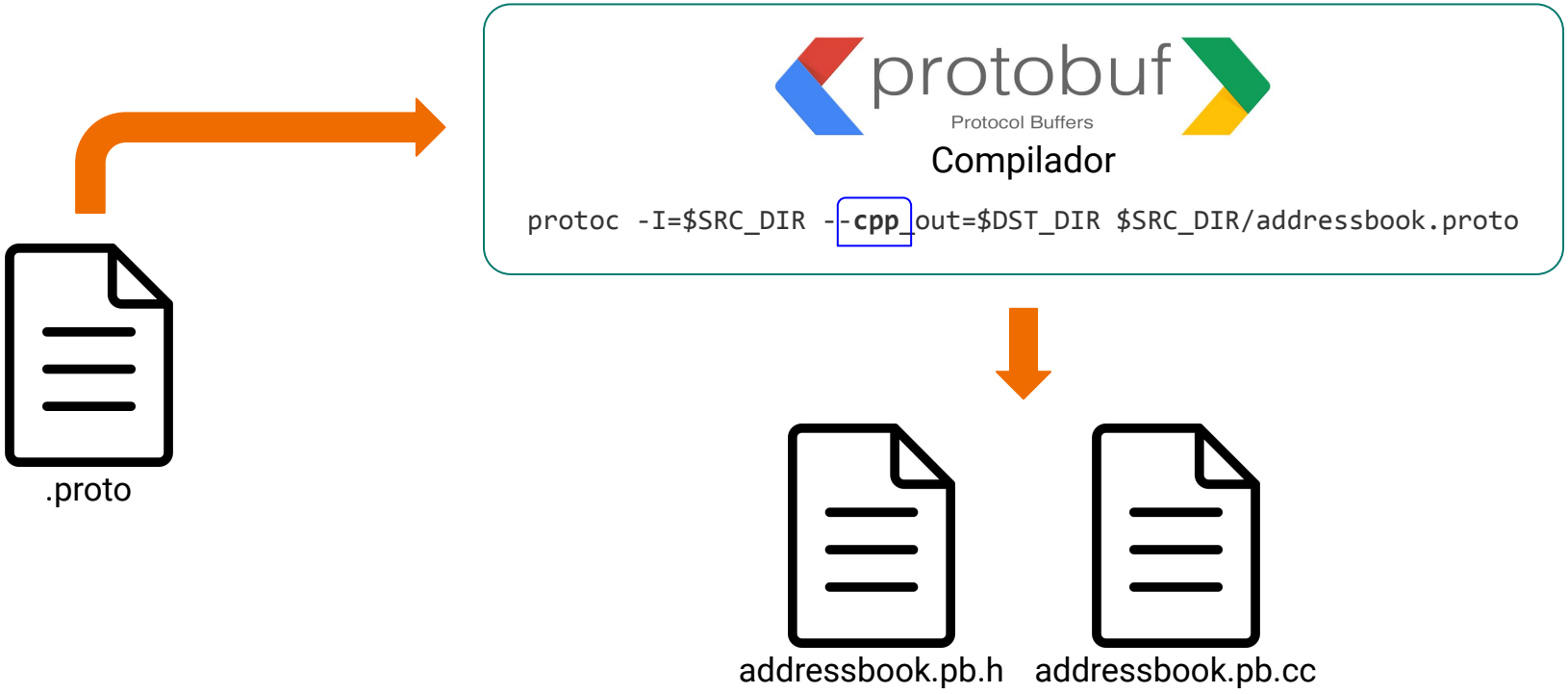
Mensaje **AddressBook**

```
syntax = "proto2";

package tutorial;

option java_multiple_files = true;
option java_package = "com.example.tutorial.protos";
option java_outer_classname = "AddressBookProtos";

message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

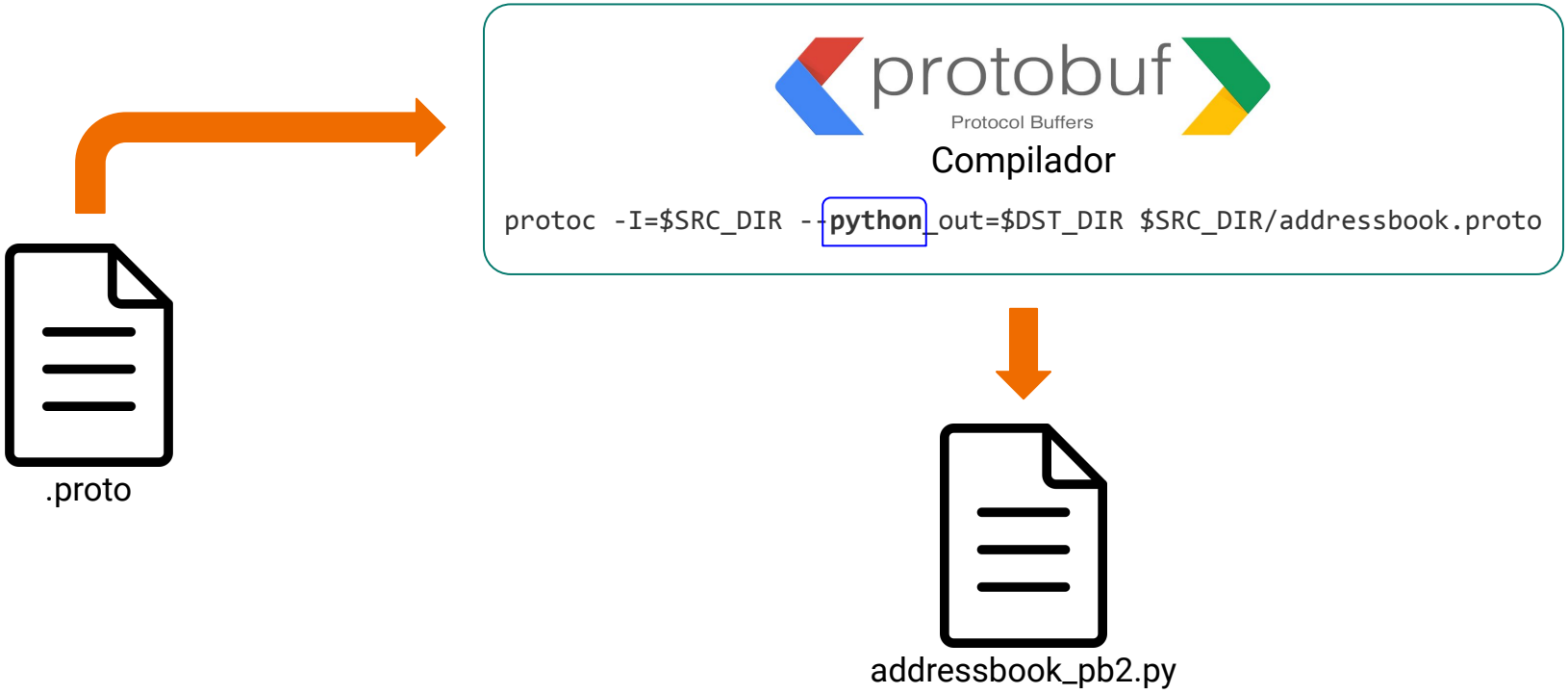
Generación de código Java



Generación de código C++



Generación de código Python



Lenguajes compatibles

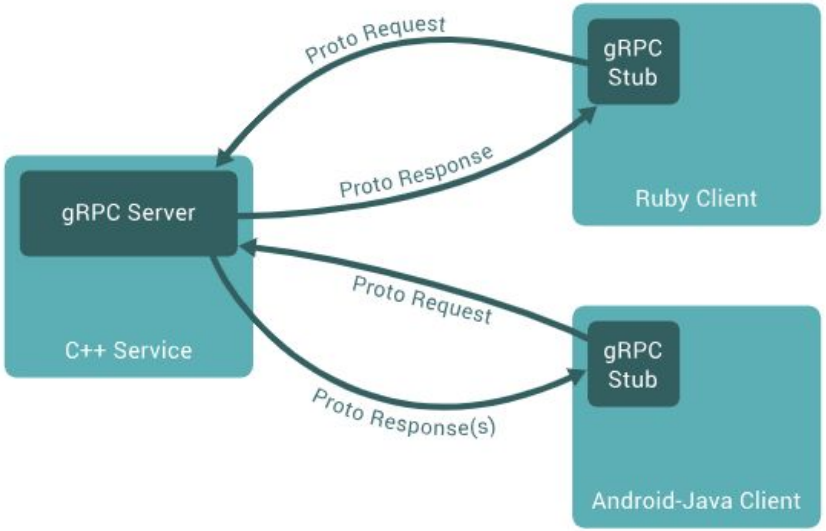
Protocol Buffers vs JSON

- Datos binarizados: mensajes 70% más pequeños
- Independiente del lenguaje
- Puede definir reglas del protocolo

- Cadenas de texto
- Mejor integración con JS y Python
- Solo define el formato de los mensajes

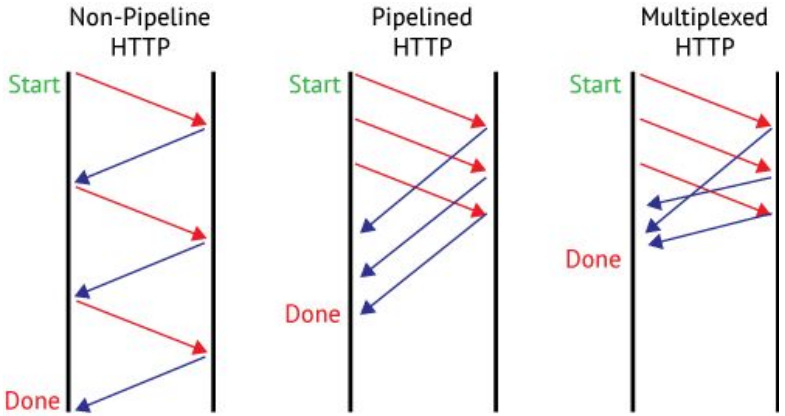
gRPC

Alternativa a las APIs REST que usa **Protocol Buffers** como mecanismo de serialización de datos estructurados.

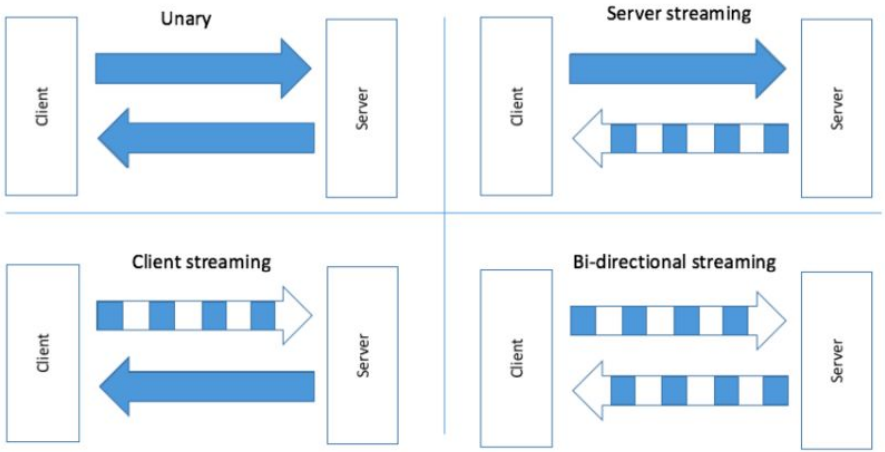


gRPC usa HTTP/2

Multiplexación



Streaming bidireccional



Usos de gRPC

Poca ocupación de red.

- Microservicios
- Internet of Things
- Domotica
- etc.

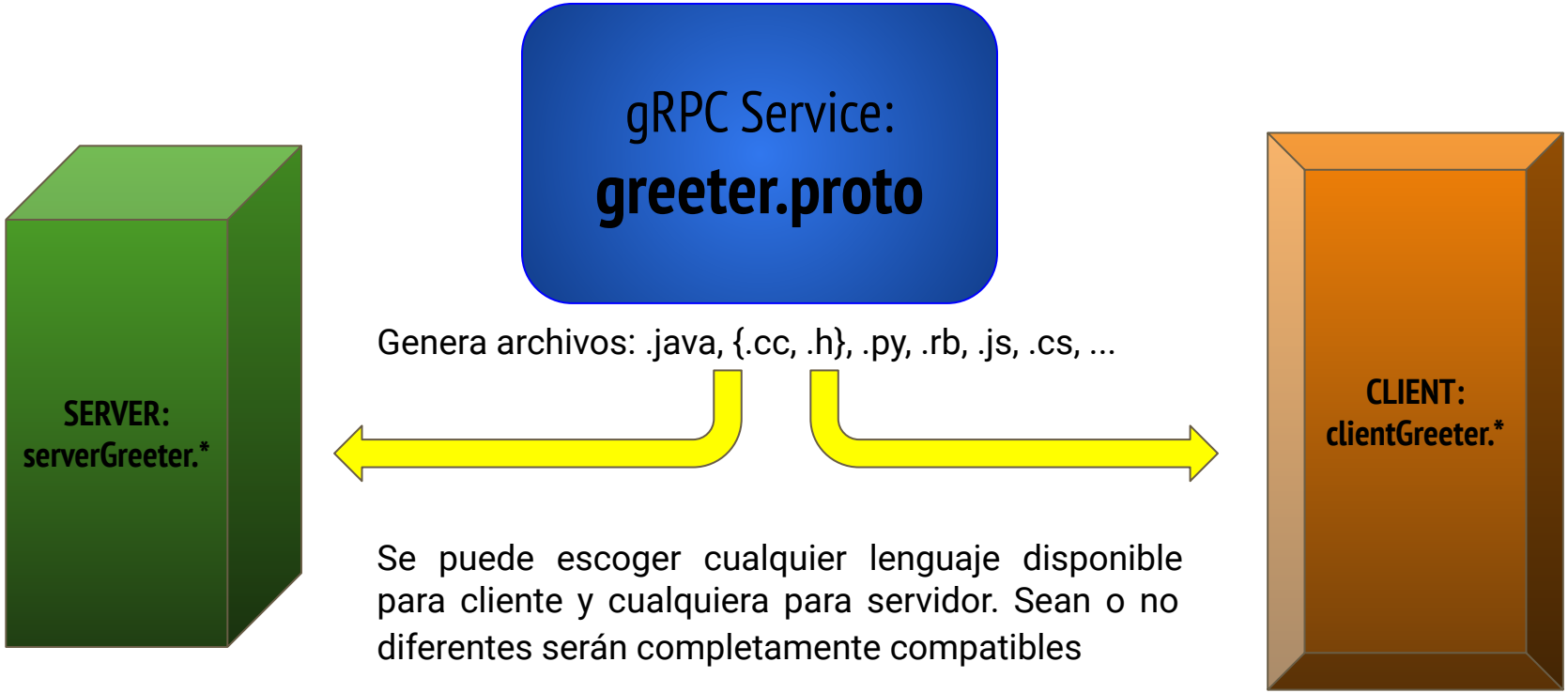
Ejemplo de uso: Crear gRPC Service

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Este código pertenece a un **archivo .proto**.

Lo primero que haremos será definir el servicio, los métodos deseados y los mensajes con los que vamos a comunicar el Servidor con el Cliente

Ejemplo de uso: Generar Cliente y Servidor



Se puede escoger cualquier lenguaje disponible para cliente y cualquiera para servidor. Sean o no diferentes serán completamente compatibles

Ejemplo de uso: Usar versión Servidor

gRPC **NO** genera la implementación completa, sinó que genera las clases y métodos definidos en el archivo .proto para ser implementados por el desarrollador.

En este ejemplo de código en JAVA podemos ver una posible implementación del servidor con las herramientas proporcionadas por gRPC.

```
private class GreeterImpl extends GreeterGrpc.GreeterImplBase {

  @Override
  public void sayHello(HelloRequest req, StreamObserver<HelloReply> responseObse
    HelloReply reply = HelloReply.newBuilder().setMessage("Hello " + req.getName
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
  }

  @Override
  public void sayHelloAgain(HelloRequest req, StreamObserver<HelloReply> respons
    HelloReply reply = HelloReply.newBuilder().setMessage("Hello again " + req.g
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
  }
}
```

Ejemplo de uso: Usar versión Servidor

```cpp
class GreeterServiceImpl final : public Greeter::Service {
  Status SayHello(ServerContext* context, const HelloRequest* request,
                  HelloReply* reply) override {
    // ...
  }

  Status SayHelloAgain(ServerContext* context, const HelloRequest* request,
                       HelloReply* reply) override {
    std::string prefix("Hello again ");
    reply->set_message(prefix + request->name());
    return Status::OK;
  }
};
```

C++

```ruby
class GreeterServer < Helloworld::Greeter::Service

  def say_hello(hello_req, _unused_call)
    Helloworld::HelloReply.new(message: "Hello #{hello_req.name}")
  end

  def say_hello_again(hello_req, _unused_call)
    Helloworld::HelloReply.new(message: "Hello again, #{hello_req.name}")
  end
end
```

RUBY

```python
class Greeter(helloworld_pb2_grpc.GreeterServicer):

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

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

PYTHON

Misma implementación en diferentes lenguajes.

Generando de nuevo la versión del lenguaje deseado del **MISMO** archivo .proto

Ejemplo de uso: Usar versión Cliente

En este ejemplo de código en JAVA podemos ver una posible implementación del cliente con las herramientas proporcionadas por gRPC.

```
public void greet(String name) {
  logger.info("Will try to greet " + name + " ...");
  HelloRequest request = HelloRequest.newBuilder().setName(name).build();
  HelloReply response;
  try {
    response = blockingStub.sayHello(request);
  } catch (StatusRuntimeException e) {
    logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    return;
  }
  logger.info("Greeting: " + response.getMessage());
  try {
    response = blockingStub.sayHelloAgain(request);
  } catch (StatusRuntimeException e) {
    logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    return;
  }
  logger.info("Greeting: " + response.getMessage());
}
```

Ejemplo de uso: Usar versión Cliente

PYTHON

```python
def run():
  channel = grpc.insecure_channel('localhost:50051')
  stub = helloworld_pb2_grpc.GreeterStub(channel)
  response = stub.SayHello(helloworld_pb2.HelloRequest(name='you'))
  print("Greeter client received: " + response.message)
  response = stub.SayHelloAgain(helloworld_pb2.HelloRequest(name='you'))
  print("Greeter client received: " + response.message)
```

RUBY

```ruby
def main
  stub = Helloworld::Greeter::Stub.new('localhost:50051', :this_channel_is_insecure)
  user = ARGV.size > 0 ? ARGV[0] : 'world'
  message = stub.say_hello(Helloworld::HelloRequest.new(name: user)).message
  p "Greeting: #{message}"
  message = stub.say_hello_again(Helloworld::HelloRequest.new(name: user)).message
  p "Greeting: #{message}"
end
```

C++

```cpp
class GreeterClient {
 public:
  // ...
  std::string SayHello(const std::string& user) {
     // ...
  }

  std::string SayHelloAgain(const std::string& user) {
    // Follows the same pattern as SayHello.
    HelloRequest request;
    request.set_name(user);
    HelloReply reply;
    ClientContext context;

    // Here we can use the stub's newly available method we just added.
    Status status = stub_->SayHelloAgain(&context, request, &reply);
    if (status.ok()) {
      return reply.message();
    } else {
      std::cout << status.error_code() << ": " << status.error_message()
                << std::endl;
      return "RPC failed";
    }
  }
```

MISMA implementación en diferentes lenguajes

Puntos fuertes

- Transmisión eficiente
- Solución de lenguaje neutral
- Codifica enumeraciones

- Utiliza HTTP/2
- Mayor eficacia de la red

Puntos débiles

- Comparación de mensajes
- Lenguajes no orientados a objetos
- No es legible por humanos

- Servicios HTTP/2 en navegadores web
- Falta de herramientas para HTTP/2

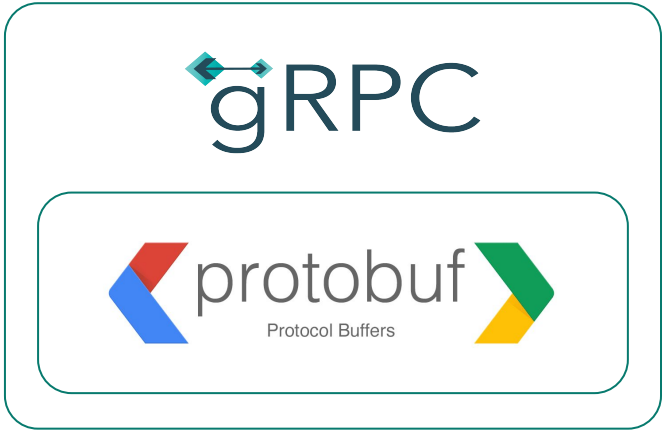
Conclusiones

Alternativa muy válida a las APIs REST

Mayor velocidad y eficiencia del uso de red

Independiente del lenguaje y de la plataforma

Facilita la creación de sistemas distribuidos



Referencias

- https://grpc.io/about/
- https://grpc.io/docs/what-is-grpc/introduction/
- https://developers.google.com/protocol-buffers/docs/overview
- https://github.com/protocolbuffers/protobuf/releases
- https://github.com/grpc/grpc/releases
- https://grpc.io/blog/principles/
- https://www.bizety.com/2018/11/12/protocol-buffers-vs-json/
- https://www.educba.com/protobuf-vs-json/
- https://developers.google.com/protocol-buffers/docs/overview
- https://en.wikipedia.org/wiki/Protocol_Buffers
- https://sakshichahal53.medium.com/json-vs-protocol-buffer-simplified-dbd6b69ca528
- https://www.altexsoft.com/blog/what-is-grpc/
- https://docs.microsoft.com/es-es/aspnet/core/grpc/comparison?view=aspnetcore-6.0
- https://daily.dev/blog/grpc-vs-rest

Reparto

Adrià Abad:
- Puntos Fuertes
- Puntos Débiles

David Jiménez:
- Ejemplos de la tecnología

Thiago Mulero:
- Introducción
- Evolución Histórica

Raül Sampietro:
- Características Tecnológicas