What is gRPC?

Edit **index.md**

Guides

This document introduces you to gRPC and protocol buffers. gRPC can use protocol buffers as both its IDL and as its underlying message interchange format. If you're new to gRPC and/or protocol buffers, read this! If you just want to dive in and see gRPC in action first, see our Quick Starts (../quickstart).

Overview

In gRPC a client application can directly call methods on a server application on a different machine as if it was a local object, making it easier for you to create distributed applications and services. As in many RPC systems, gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. On the server side, the server implements this interface and runs a gRPC server to handle client calls. On the client side, the client has a stub (referred to as just a client in some languages) that provides the same methods as the server.
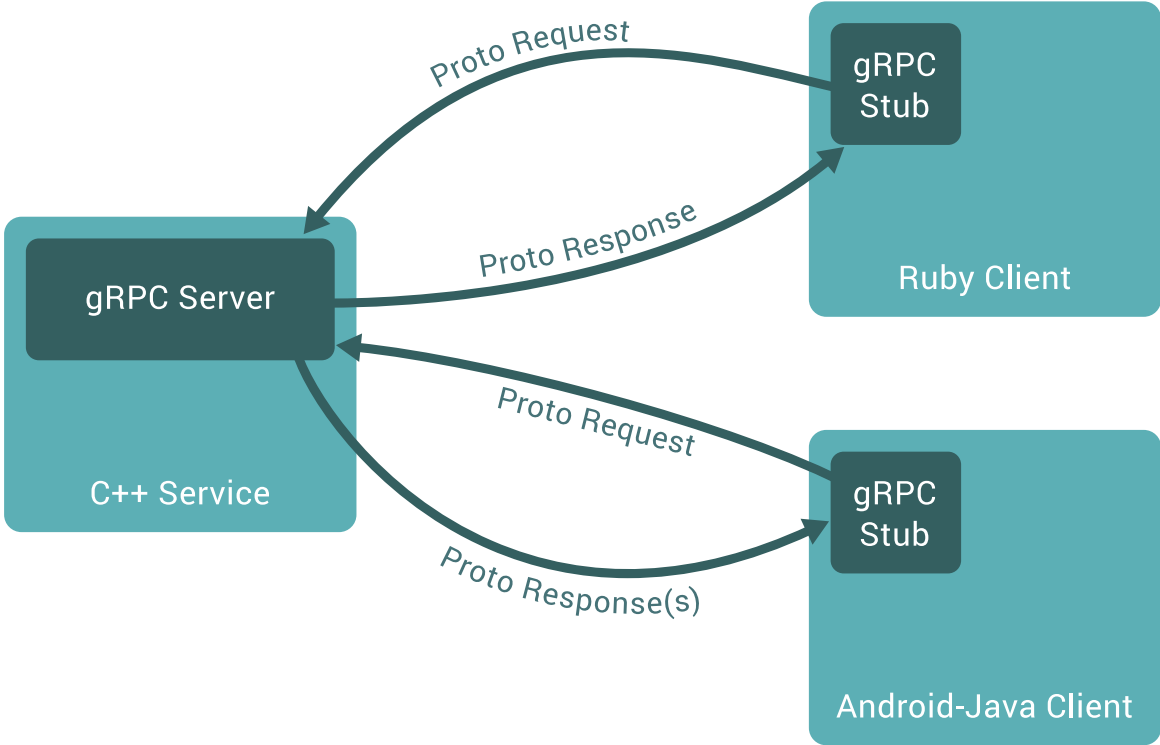
gRPC clients and servers can run and talk to each other in a variety of environments - from servers inside Google to your own desktop - and can be written in any of gRPC's supported languages. So, for example, you can easily create a gRPC server in Java with clients in Go, Python, or Ruby. In addition, the latest Google APIs will have gRPC versions of their interfaces, letting you easily build Google functionality into your applications.

Working with Protocol Buffers

By default gRPC uses protocol buffers , Google's mature open source mechanism for serializing structured data (although it can

be used with other data formats such as JSON). Here's a quick intro to how it works. If you're already familiar with protocol buffers, feel free to skip ahead to the next section.

The first step when working with protocol buffers is to define the structure for the data you want to serialize in a *proto file*: this is an ordinary text file with a `.proto` extension. Protocol buffer data is structured as *messages*, where each message is a small logical record of information containing a series of name-value pairs called *fields*. Here's a simple example:

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

Then, once you've specified your data structures, you use the protocol buffer compiler `protoc` to generate data access classes in your preferred language(s) from your proto definition. These provide simple accessors for each field (like `name()` and `set_name()`) as well as methods to serialize/parse the whole structure to/from raw bytes – so, for instance, if your chosen language is C++, running the compiler on the above example will generate a class called `Person`. You can then use this class in your application to populate, serialize, and retrieve Person protocol buffer messages.

As you'll see in more detail in our examples, you define gRPC services in ordinary proto files, with RPC method parameters and return types specified as protocol buffer messages:

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (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;
}
```

gRPC also uses `protoc` with a special gRPC plugin to generate code from your proto file. However, with the gRPC plugin, you get generated gRPC client and server code, as well as the regular protocol buffer code for populating, serializing, and retrieving your message types. We'll look at this example in more detail below.

You can find out lots more about protocol buffers in the Protocol Buffers documentation , and find out how to get and install `protoc` with gRPC plugins in your chosen language's Quickstart.

Protocol buffer versions

While protocol buffers have been available for open source users for some time, our examples use a new flavor of protocol buffers called proto3, which has a slightly simplified syntax, some useful new features, and supports lots more languages. This is currently available in Java, C++, Python, Objective-C, C#, a lite-runtime (Android Java), Ruby, and JavaScript from the protocol buffers Github repo , as well as a Go language generator from the golang/protobuf Github repo , with more languages in development. You can find out more in the proto3 language guide and the reference documentation for each language (where available), and see the major differences from the current default version in the release notes. Even more proto3 documentation is coming soon.

In general, while you can use proto2 (the current default protocol buffers version), we recommend that you use proto3 with gRPC as it lets you use the full range of gRPC-supported languages, as well as avoiding compatibility issues with proto2 clients talking to proto3 servers and vice versa.

GRPC

Documentation	Resources	Community
Overview	About	Github
Quick Start	Blog	Twitter
Guides	FAQ	Stack Overflow
Tutorials		YouTube
Reference APIs		