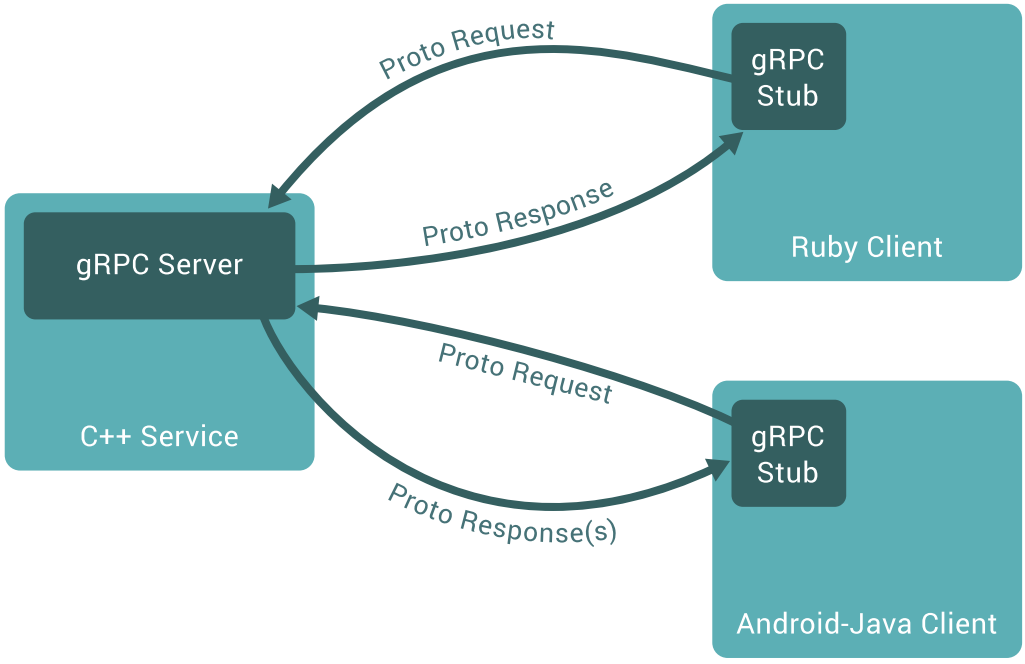
GRPC – First step

This document introduces you to gRPC and protocol buffers. gRPC can use protocol buffers as both its Interface Definition Language (IDL) and as its underlying message interchange format. If you're new to gRPC and/or protocol buffers, read this!

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 available for each language. The reference documentation also includes a formal specification for the `.proto` file format.

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 Concepts – Second Step

This document introduces some key gRPC concepts with an overview of gRPC's architecture and RPC life cycle.

Overview

Service definition

Like 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. By default, gRPC uses protocol buffers as the Interface Definition Language (IDL) for describing both the service interface and the structure of the payload messages. It is possible to use other alternatives if desired.

```
service HelloService
{
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest
{
  string greeting = 1;
}

message HelloResponse
{
  string reply = 1;
}
```

gRPC lets you define four kinds of service method:

- Unary RPCs where the client sends a single request to the server and gets a single response back, just like a normal function call.

```
rpc SayHello(HelloRequest) returns (HelloResponse){
}
```

- Server streaming RPCs where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages. gRPC guarantees message ordering within an individual RPC call.

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse){
}
```

- Client streaming RPCs where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it

waits for the server to read them and return its response. Again gRPC guarantees message ordering within an individual RPC call.

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse) {
}
```

- Bidirectional streaming RPCs where both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved.

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse){
}
```

We'll look at the different types of RPC in more detail in the RPC life cycle section below.

Using the API surface

Starting from a service definition in a .proto file, gRPC provides protocol buffer compiler plugins that generate client- and server-side code. gRPC users typically call these APIs on the client side and implement the corresponding API on the server side.

- On the server side, the server implements the methods declared by the service and runs a gRPC server to handle client calls. The gRPC infrastructure decodes incoming requests, executes service methods, and encodes service responses.
- On the client side, the client has a local object known as *stub* (for some languages, the preferred term is *client*) that implements the same methods as the service. The client can then just call those methods on the local object, wrapping the parameters for the call in the appropriate protocol buffer message type - gRPC looks after sending the request(s) to the server and returning the server's protocol buffer response(s).

Synchronous vs. asynchronous

Synchronous RPC calls that block until a response arrives from the server are the closest approximation to the abstraction of a procedure call that RPC aspires to. On the other hand, networks are inherently asynchronous and in many scenarios it's useful to be able to start RPCs without blocking the current thread.

The gRPC programming surface in most languages comes in both synchronous and asynchronous flavors. You can find out more in each language's tutorial and reference documentation (complete reference docs are coming soon).

RPC life cycle

Now let's take a closer look at what happens when a gRPC client calls a gRPC server method. We won't look at implementation details, you can find out more about these in our language-specific pages.

Unary RPC

First let's look at the simplest type of RPC, where the client sends a single request and gets back a single response.

- Once the client calls the method on the stub/client object, the server is notified that the RPC has been invoked with the client's metadata for this call, the method name, and the specified deadline if applicable.
- The server can then either send back its own initial metadata (which must be sent before any response) straight away, or wait for the client's request message - which happens first is application-specific.
- Once the server has the client's request message, it does whatever work is necessary to create and populate its response. The response is then returned (if successful) to the client together with status details (status code and optional status message) and optional trailing metadata.
- If the status is OK, the client then gets the response, which completes the call on the client side.

Server streaming RPC

A server-streaming RPC is similar to our simple example, except the server sends back a stream of responses after getting the client's request message. After sending back all its responses, the server's status details (status code and optional status message) and optional trailing metadata are sent back to complete on the server side. The client completes once it has all the server's responses.

Client streaming RPC

A client-streaming RPC is also similar to our simple example, except the client sends a stream of requests to the server instead of a single request. The server sends back a single response, typically but not necessarily after it has received all the client's requests, along with its status details and optional trailing metadata.

Bidirectional streaming RPC

In a bidirectional streaming RPC, again the call is initiated by the client calling the method and the server receiving the client metadata, method name, and deadline. Again the server can choose to send back its initial metadata or wait for the client to start sending requests.

What happens next depends on the application, as the client and server can read and write in any order - the streams operate completely independently. So, for example, the server could wait until it has received all the client's messages before writing its responses, or the server and client could "ping-pong": the server gets a request, then sends back a response, then the client sends another request based on the response, and so on.

Deadlines/Timeouts

gRPC allows clients to specify how long they are willing to wait for an RPC to complete before the RPC is terminated with the error `DEADLINE_EXCEEDED`. On the server side, the server can query to see if a particular RPC has timed out, or how much time is left to complete the RPC.

How the deadline or timeout is specified varies from language to language - for example, not all languages have a default deadline, some language APIs work in terms of a deadline (a fixed point in time), and some language APIs work in terms of timeouts (durations of time).

RPC termination

In gRPC, both the client and server make independent and local determinations of the success of the call, and their conclusions may not match. This means that, for example, you could have an RPC that finishes successfully on the server side (“I have sent all my responses!”) but fails on the client side (“The responses arrived after my deadline!“). It’s also possible for a server to decide to complete before a client has sent all its requests.

Cancelling RPCs

Either the client or the server can cancel an RPC at any time. A cancellation terminates the RPC immediately so that no further work is done. It is *not* an “undo”: changes made before the cancellation will not be rolled back.