

Links: [RPC](#)

gRPC

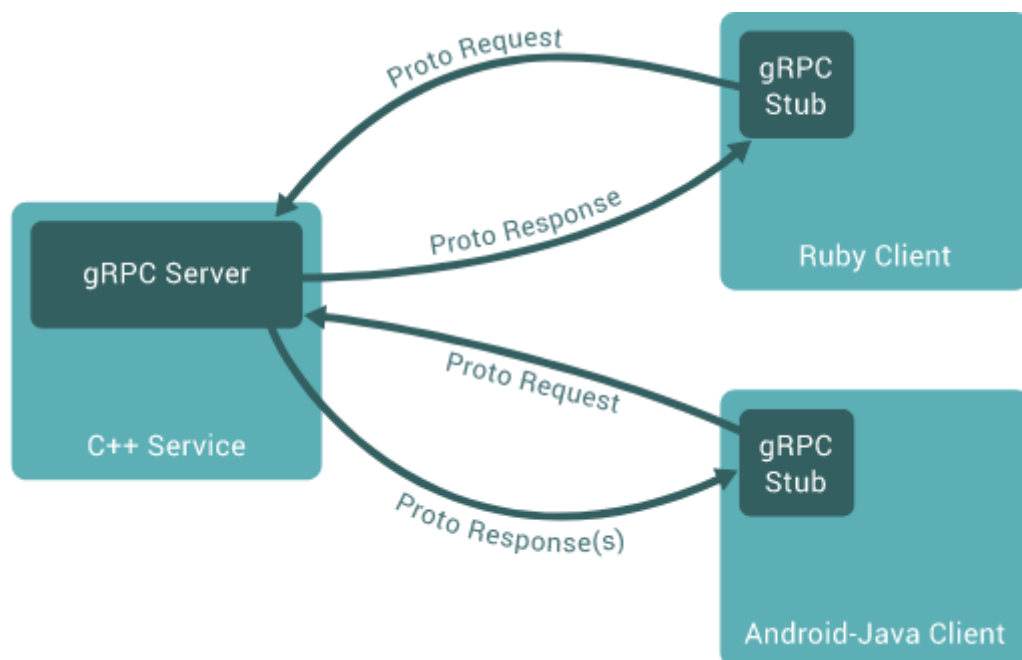
What is RPC

Remote procedure call (RPC) is a request–response protocol.

In distributed computing, **RPC** is when a computer program causes a procedure (subroutine) to execute in a different computer in the network, which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction.

gRPC Introduction

gRPC allows client to call server's methods directly.
Making it easy to create distributed applications.
While using HTTP/2 protocol.



Protocol Buffers

Structure definition

By default, gRPC uses [Protocol Buffers](#) it is mechanism for serializing data. Its not mandatory JSON can be used as well.

First step while writing proto buffer is to define structure for the data you wish to serialize

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

Service definition

You may also define gRPC services in proto files

```
// 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;  
}
```

In order to turn your proto files into usable code Google uses `protoc` compiler which turns `.proto` files into code from language of your choice.

Types of service method

Unary

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 to client stream

Where client send a request to server and receives a stream. Client reads the stream until there are no more messages.

gRPC guarantees message ordering within an individual RPC call

```
rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
```

Client to server stream

Client writes a sequence of messages and sends them to the server. When client has finished writing messages, it waits for the server to read them and return response.

gRPC guarantees message ordering within an individual RPC call

```
rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
```

Bidirectional stream

Both sides (client and server) sends a sequence of messages using a read-write stream. Streams operate **independently** so clients and server can read and write in multiple orders.

- server could wait to receive all the client messages before writing its responses
- server could alternately read a message then write a message

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
```

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 a `DEADLINE_EXCEEDED` error. 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.

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 an RPC

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.

 **Changes made before a cancellation are not rolled back.**

Metadata

Metadata is information about a particular RPC call (such as [authentication details](#)) in the form of a list of key-value pairs, where the keys are strings and the values are typically strings, but can be binary data.

Metadata is opaque to gRPC itself - it lets the client provide information associated with the call to the server and vice versa.

Access to metadata is language dependent.

Channels

A gRPC channel provides a connection to a gRPC server on a specified host and port. It is used when creating a client stub. Clients can specify channel arguments to modify gRPC's default behavior, such as switching message compression on or off. A channel has state, including `connected` and `idle`.

How gRPC deals with closing a channel is language dependent. Some languages also permit querying channel state.

To learn more

<https://grpc.io/>

<https://grpc.io/blog/principles/>