Core concepts, architecture and lifecycle

An introduction to key gRPC concepts, with an overview of gRPC architecture and RPC life cycle.

Not familiar with gRPC? First read Introduction to gRPC. For language-specific details, see the quick start, tutorial, and reference documentation for your language of choice.

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);
  ```

You'll learn more about the different types of RPC in the RPC life cycle section below.

Using the API

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 API 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

In this section, you'll take a closer look at what happens when a gRPC client calls a gRPC server method. For complete implementation details, see the language-specific pages.

Unary RPC

First consider the simplest type of RPC where the client sends a single request and gets back a single response.

1. Once the client calls a stub method, 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.
2. 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.
3. Once the server has the client's request message, it does whatever work is necessary to create and populate a 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.
4. If the response status is OK, then the client gets the response, which completes the call on the client side.

Server streaming RPC

A server-streaming RPC is similar to a unary RPC, except that the server returns a stream of messages in response to a client's request. After sending all its messages, the server's status details (status code and optional status message) and optional trailing metadata are sent to the client. This completes processing on the server side. The client completes once it has all the server's messages.

Client streaming RPC

A client-streaming RPC is similar to a unary RPC, except that the client sends a stream of messages to the server instead of a single message. The server responds with a single message (along with its status details and optional trailing metadata), typically but not necessarily after it has received all the client's messages.

Bidirectional streaming RPC

In a bidirectional streaming RPC, the call is initiated by the client invoking the method and the server receiving the client metadata, method name, and deadline. The server can choose to send back its initial metadata or wait for the client to start streaming messages.

Client- and server-side stream processing is application specific. Since the two streams are independent, the client and server can read and write messages in any order. For example, a server can wait until it has received all of a client's messages before writing its messages, or the server and client can play "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 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.

Specifying a deadline or timeout is language specific: some language APIs work in terms of timeouts (durations of time), and some language APIs work in terms of a deadline (a fixed point in time) and may or may not have a default deadline.

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.

> Warning
>
> 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.
