Basics tutorial

A basic tutorial introduction to gRPC in C++.

This tutorial provides a basic C++ programmer's introduction to working with gRPC.

By walking through this example you'll learn how to:

- Define a service in a `.proto` file.
- Generate server and client code using the protocol buffer compiler.
- Use the C++ gRPC API to write a simple client and server for your service.

It assumes that you have read the Introduction to gRPC and are familiar with protocol buffers. Note that the example in this tutorial uses the proto3 version of the protocol buffers language: you can find out more in the proto3 language guide and C++ generated code guide.

Why use gRPC?

Our example is a simple route mapping application that lets clients get information about features on their route, create a summary of their route, and exchange route information such as traffic updates with the server and other clients.

With gRPC we can define our service once in a `.proto` file and generate clients and servers in any of gRPC's supported languages, which in turn can be run in environments ranging from servers inside a large data center to your own tablet — all the complexity of communication between different languages and environments is handled for you by gRPC. We also get all the advantages of working with protocol buffers, including efficient serialization, a simple IDL, and easy interface updating.

Example code and setup

The example code is part of the `grpc` repo under examples/cpp/route_guide. Get the example code and build gRPC:

1. Follow the Quick start instructions to build and locally install gRPC from source.

2. From the repo folder, change to the route guide example directory:

   ```
   $ cd examples/cpp/route_guide
   ```

3. Run `cmake`

   ```
   $ mkdir -p cmake/build
   $ cd cmake/build
   $ cmake -DCMAKE_PREFIX_PATH=$MY_INSTALL_DIR ../..
   ```

Defining the service

Our first step (as you'll know from the Introduction to gRPC) is to define the gRPC *service* and the method *request* and *response* types using protocol buffers. You can see the complete `.proto` file in `examples/protos/route_guide.proto`.

To define a service, you specify a named `service` in your `.proto` file:

```
service RouteGuide {
   ...
}
```

Then you define `rpc` methods inside your service definition, specifying their request and response types. gRPC lets you define four kinds of service method, all of which are used in the `RouteGuide` service:

- A *simple RPC* where the client sends a request to the server using the stub and waits for a response to come back, just like a normal function call.

```
// Obtains the feature at a given position.
rpc GetFeature(Point) returns (Feature) {}
```

- A *server-side streaming RPC* 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. As you can see in our example, you specify a server-side streaming method by placing the `stream` keyword before the *response* type.

```
// Obtains the Features available within the given Rectangle.  Resu
// streamed rather than returned at once (e.g. in a response messag
// repeated field), as the rectangle may cover a large area and con
// huge number of features.
rpc ListFeatures(Rectangle) returns (stream Feature) {}
```

- A *client-side streaming RPC* 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 all and return its response. You specify a client-side streaming method by placing the `stream` keyword before the *request* type.

```
// Accepts a stream of Points on a route being traversed, returning
// RouteSummary when traversal is completed.
rpc RecordRoute(stream Point) returns (RouteSummary) {}
```

- A *bidirectional streaming RPC* 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. You specify this type of method by placing the `stream` keyword before both the request and the response.

```
// Accepts a stream of RouteNotes sent while a route is being travel
// while receiving other RouteNotes (e.g. from other users).
rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
```

Our `.proto` file also contains protocol buffer message type definitions for all the request and response types used in our service methods - for example, here's the `Point` message type:

```
// Points are represented as latitude-longitude pairs in the E7 represent
// (degrees multiplied by 10**7 and rounded to the nearest integer).
// Latitudes should be in the range +/- 90 degrees and longitude should b
// the range +/- 180 degrees (inclusive).
message Point {
  int32 latitude = 1;
  int32 longitude = 2;
}
```

Generating client and server code

Next we need to generate the gRPC client and server interfaces from our `.proto` service definition. We do this using the protocol buffer compiler `protoc` with a special gRPC C++ plugin.

For simplicity, we've provided a CMakeLists.txt that runs `protoc` for you with the appropriate plugin, input, and output (if you want to run this yourself, make sure you've installed protoc and followed the gRPC code installation instructions first):

```
$ make route_guide.grpc.pb.o
```

which actually runs:

```
$ protoc -I ../../protos --grpc_out=. --plugin=protoc-gen-grpc=`which grp
$ protoc -I ../../protos --cpp_out=. ../../protos/route_guide.proto
```

Running this command generates the following files in your current directory:

- `route_guide.pb.h` , the header which declares your generated message classes
- `route_guide.pb.cc` , which contains the implementation of your message classes
- `route_guide.grpc.pb.h` , the header which declares your generated service classes
- `route_guide.grpc.pb.cc` , which contains the implementation of your service classes

These contain:

- All the protocol buffer code to populate, serialize, and retrieve our request and response message types

- A class called `RouteGuide` that contains
  - a remote interface type (or *stub*) for clients to call with the methods defined in the `RouteGuide` service.
  - two abstract interfaces for servers to implement, also with the methods defined in the `RouteGuide` service.

Creating the server

First let's look at how we create a `RouteGuide` server. If you're only interested in creating gRPC clients, you can skip this section and go straight to [Creating the client](#) (though you might find it interesting anyway!).

There are two parts to making our `RouteGuide` service do its job:

- Implementing the service interface generated from our service definition: doing the actual "work" of our service.
- Running a gRPC server to listen for requests from clients and return the service responses.

You can find our example `RouteGuide` server in [examples/cpp/route_guide/route_guide_server.cc](#). Let's take a closer look at how it works.

Implementing RouteGuide

As you can see, our server has a `RouteGuideImpl` class that implements the generated `RouteGuide::Service` interface:

```
class RouteGuideImpl final : public RouteGuide::Service {
...
}
```

In this case we're implementing the *synchronous* version of `RouteGuide`, which provides our default gRPC server behaviour. It's also possible to implement an asynchronous interface, `RouteGuide::AsyncService`, which allows you to further customize your server's threading behaviour, though we won't look at this in this tutorial.

`RouteGuideImpl` implements all our service methods. Let's look at the simplest type first, `GetFeature`, which just gets a `Point` from the client and returns the corresponding feature information from its database in a `Feature`.

```
Status GetFeature(ServerContext* context, const Point* point,
                  Feature* feature) override {
  feature->set_name(GetFeatureName(*point, feature_list_));
  feature->mutable_location()->CopyFrom(*point);
  return Status::OK;
}
```

The method is passed a context object for the RPC, the client's `Point` protocol buffer request, and a `Feature` protocol buffer to fill in with the response information. In the method we populate the `Feature` with the appropriate information, and then `return` with an `OK` status to tell gRPC that we've finished dealing with the RPC and that the `Feature` can be returned to the client.

Note that all service methods can (and will!) be called from multiple threads at the same time. You have to make sure that your method implementations are thread safe. In our example, `feature_list_` is never changed after construction, so it is safe by design. But if `feature_list_` would change during the lifetime of the service, we would need to synchronize access to this member.

Now let's look at something a bit more complicated - a streaming RPC. `ListFeatures` is a server-side streaming RPC, so we need to send back multiple `Feature` s to our client.

```
Status ListFeatures(ServerContext* context, const Rectangle* rectangle,
                    ServerWriter<Feature>* writer) override {
  auto lo = rectangle->lo();
  auto hi = rectangle->hi();
  long left = std::min(lo.longitude(), hi.longitude());
  long right = std::max(lo.longitude(), hi.longitude());
  long top = std::max(lo.latitude(), hi.latitude());
  long bottom = std::min(lo.latitude(), hi.latitude());
  for (const Feature& f : feature_list_) {
    if (f.location().longitude() >= left &&
        f.location().longitude() <= right &&
        f.location().latitude() >= bottom &&
        f.location().latitude() <= top) {
      writer->Write(f);
    }
  }
  return Status::OK;
}
```

As you can see, instead of getting simple request and response objects in our method parameters, this time we get a request object (the `Rectangle` in which our client wants to find `Feature` s) and a special `ServerWriter` object. In the method, we populate as many `Feature` objects as we need to return, writing them to the `ServerWriter` using its `Write()` method. Finally, as in our simple RPC, we `return Status::OK` to tell gRPC that we've finished writing responses.

If you look at the client-side streaming method `RecordRoute` you'll see it's quite similar, except this time we get a `ServerReader` instead of a request object and a single response. We use the `ServerReader` s `Read()` method to repeatedly read in our client's requests to a request object (in this case a `Point`) until there are no more messages: the server needs to check the return value of `Read()` after each call. If `true` , the stream is still good and it can continue reading; if `false` the message stream has ended.

```
while (stream->Read(&point)) {
  ...//process client input
}
```

Finally, let's look at our bidirectional streaming RPC `RouteChat()` .

```
Status RouteChat(ServerContext* context,
                 ServerReaderWriter<RouteNote, RouteNote>* stream) overr
  RouteNote note;
  while (stream->Read(&note)) {
    std::unique_lock<std::mutex> lock(mu_);
    for (const RouteNote& n : received_notes_) {
      if (n.location().latitude() == note.location().latitude() &&
          n.location().longitude() == note.location().longitude()) {
        stream->Write(n);
      }
    }
    received_notes_.push_back(note);
  }

  return Status::OK;
}
```

This time we get a `ServerReaderWriter` that can be used to read *and* write messages. The syntax for reading and writing here is exactly the same as for our client-streaming and server-streaming methods. Although each side will always get the other's messages in the order they were written, both the client and server can read and write in any order — the streams operate completely independently.

Note that since `received_notes_` is an instance variable and can be accessed by multiple threads, we use a mutex lock here to guarantee exclusive access.

Starting the server

Once we've implemented all our methods, we also need to start up a gRPC server so that clients can actually use our service. The following snippet shows how we do this for our `RouteGuide` service:

```
void RunServer(const std::string& db_path) {
  std::string server_address("0.0.0.0:50051");
  RouteGuideImpl service(db_path);

  ServerBuilder builder;
  builder.AddListeningPort(server_address, grpc::InsecureServerCredentia
  builder.RegisterService(&service);
  std::unique_ptr<Server> server(builder.BuildAndStart());
  std::cout << "Server listening on " << server_address << std::endl;
  server->Wait();
}
```

As you can see, we build and start our server using a `ServerBuilder`. To do this, we:

1. Create an instance of our service implementation class `RouteGuideImpl`.
2. Create an instance of the factory `ServerBuilder` class.
3. Specify the address and port we want to use to listen for client requests using the builder's `AddListeningPort()` method.
4. Register our service implementation with the builder.
5. Call `BuildAndStart()` on the builder to create and start an RPC server for our service.
6. Call `Wait()` on the server to do a blocking wait until process is killed or `Shutdown()` is called.

Creating the client

In this section, we'll look at creating a C++ client for our `RouteGuide` service. You can see our complete example client code in examples/cpp/route_guide/route_guide_client.cc.

Creating a stub

To call service methods, we first need to create a *stub*.

First we need to create a gRPC *channel* for our stub, specifying the server address and port we want to connect to - in our case we'll use no SSL:

```
grpc::CreateChannel("localhost:50051", grpc::InsecureChannelCredentials()
```

> **Note**
>
> In order to set additional options for the *channel*, use the `grpc::CreateCustomChannel()` api with any special channel arguments - `grpc::ChannelArguments`.

Now we can use the channel to create our stub using the `NewStub` method provided in the `RouteGuide` class we generated from our `.proto`.

```
public:
 RouteGuideClient(std::shared_ptr<ChannelInterface> channel,
                  const std::string& db)
     : stub_(RouteGuide::NewStub(channel)) {
   ...
 }
```

Calling service methods

Now let's look at how we call our service methods. Note that in this tutorial we're calling the *blocking/synchronous* versions of each method: this means that the RPC call waits for the server to respond, and will either return a response or raise an exception.

Simple RPC

Calling the simple RPC `GetFeature` is nearly as straightforward as calling a local method.

```
Point point;
Feature feature;
point = MakePoint(409146138, -746188906);
GetOneFeature(point, &feature);

...

bool GetOneFeature(const Point& point, Feature* feature) {
  ClientContext context;
  Status status = stub_->GetFeature(&context, point, feature);
  ...
}
```

As you can see, we create and populate a request protocol buffer object (in our case `Point`), and create a response protocol buffer object for the server to fill in. We also create a `ClientContext` object for our call - you can optionally set RPC configuration values on this object, such as deadlines, though for now we'll use the default settings. Note that you cannot reuse this object between calls. Finally, we call the method on the stub, passing it the context, request, and response. If the method returns `OK`, then we can read the response information from the server from our response object.

```
std::cout << "Found feature called " << feature->name()  << " at "
          << feature->location().latitude()/kCoordFactor_ << ", "
          << feature->location().longitude()/kCoordFactor_ << std::endl;
```

Streaming RPCs

Now let's look at our streaming methods. If you've already read [Creating the server]() some of this may look very familiar - streaming RPCs are implemented in a similar way on both sides. Here's where we call the server-side streaming method `ListFeatures`, which returns a stream of geographical `Feature` s:

```
std::unique_ptr<ClientReader<Feature> > reader(
    stub_->ListFeatures(&context, rect));
while (reader->Read(&feature)) {
  std::cout << "Found feature called "
            << feature.name() << " at "
            << feature.location().latitude()/kCoordFactor_ << ", "
            << feature.location().longitude()/kCoordFactor_ << std::endl
}
Status status = reader->Finish();
```

Instead of passing the method a context, request, and response, we pass it a context and request and get a `ClientReader` object back. The client can use the `ClientReader` to read the server's responses. We use the `ClientReader` s `Read()` method to repeatedly read in the server's responses to a response protocol buffer object (in this case a `Feature`) until there are no more messages: the client needs to check the return value of `Read()` after each call. If `true`, the stream is still good and it can continue reading; if `false` the message stream has ended. Finally, we call `Finish()` on the stream to complete the call and get our RPC status.

The client-side streaming method `RecordRoute` is similar, except there we pass the method a context and response object and get back a `ClientWriter`.

```
std::unique_ptr<ClientWriter<Point> > writer(
    stub_->RecordRoute(&context, &stats));
for (int i = 0; i < kPoints; i++) {
  const Feature& f = feature_list_[feature_distribution(generator)];
  std::cout << "Visiting point "
            << f.location().latitude()/kCoordFactor_ << ", "
            << f.location().longitude()/kCoordFactor_ << std::endl;
  if (!writer->Write(f.location())) {
    // Broken stream.
    break;
  }
  std::this_thread::sleep_for(std::chrono::milliseconds(
      delay_distribution(generator)));
}
writer->WritesDone();
Status status = writer->Finish();
if (status.IsOk()) {
  std::cout << "Finished trip with " << stats.point_count() << " points\
            << "Passed " << stats.feature_count() << " features\n"
            << "Travelled " << stats.distance() << " meters\n"
            << "It took " << stats.elapsed_time() << " seconds"
            << std::endl;
} else {
  std::cout << "RecordRoute rpc failed." << std::endl;
}
```

Once we've finished writing our client's requests to the stream using `Write()`, we need to call `WritesDone()` on the stream to let gRPC know that we've finished writing, then `Finish()` to complete the call and get our RPC status. If the status is `OK`, our response object that we initially passed to `RecordRoute()` will be populated with the server's response.

Finally, let's look at our bidirectional streaming RPC `RouteChat()`. In this case, we just pass a context to the method and get back a `ClientReaderWriter`, which we can use to both write and read messages.

```
std::shared_ptr<ClientReaderWriter<RouteNote, RouteNote> > stream(
    stub_->RouteChat(&context));
```

The syntax for reading and writing here is exactly the same as for our client-streaming and server-streaming methods. Although each side will always get the other's messages in the order they were written, both the client and server can read and write in any order — the streams operate completely independently.

Try it out!

Build the client and server:

```
$ make
```

Run the server:

```
$ ./route_guide_server --db_path=path/to/route_guide_db.json
```

From a different terminal, run the client:

```
$ ./route_guide_client --db_path=path/to/route_guide_db.json
```

Last modified July 14, 2023: cpp: Update basics.md - Format file extensions in code font (#1165) (7841685)

View page source · Edit this page · Create child page · Create documentation issue · Create project issue