A simplified guide to gRPC in Python

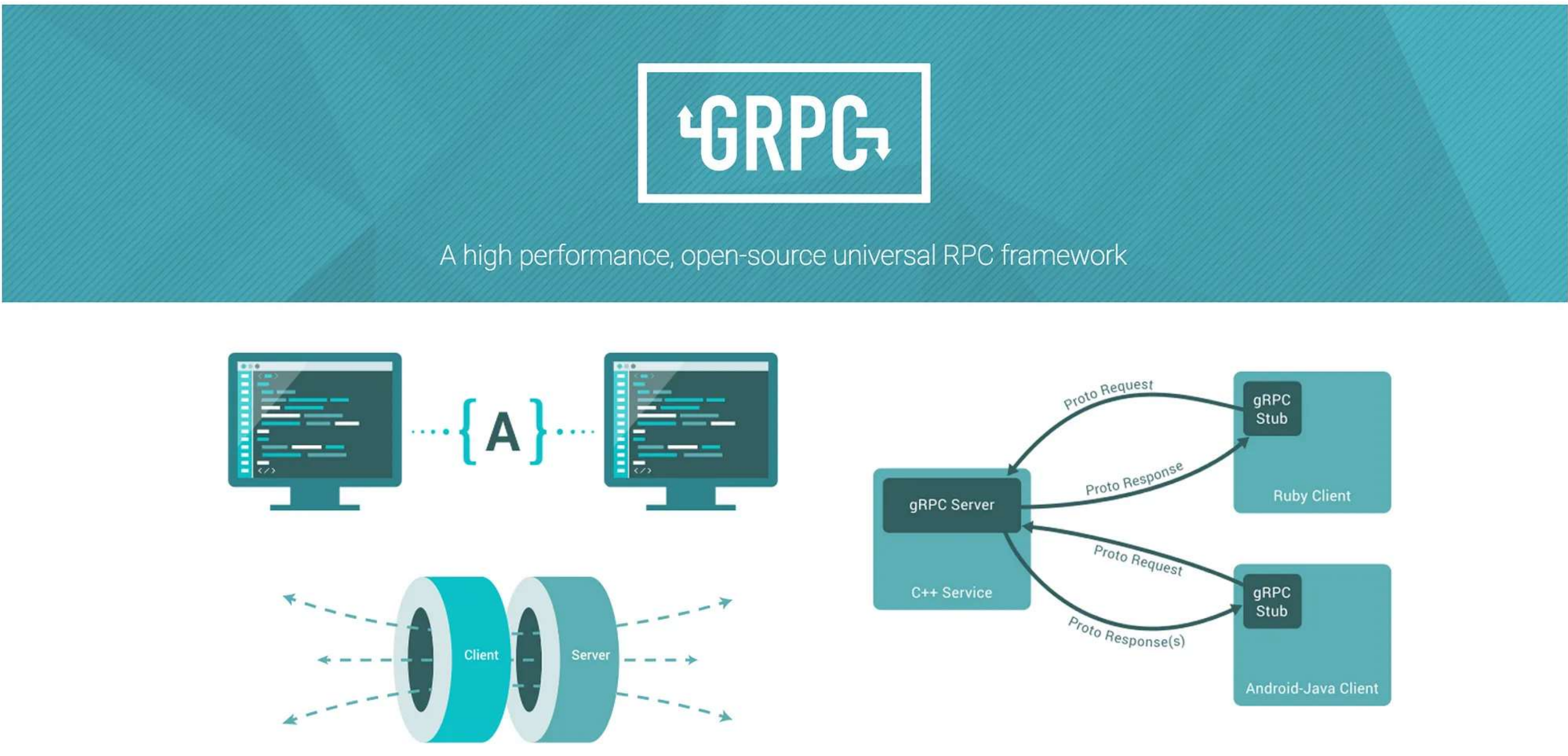
Ramanan Balakrishnan · Follow

Published in Engineering@Semantics3

3 min read · Sep 25, 2017

Listen Share More

Google's gRPC provides a framework for implementing RPC (Remote Procedure Call) workflows. By layering on top of HTTP/2 and using protocol buffers, gRPC promises a lot of benefits over conventional REST+JSON APIs.



Source: grpc.io

Considering the promised goodies, I decided to get my hands dirty and roll gRPC for some of the service-oriented environments at Semantics3.

I headed over to the official documentation, opened the section for my current language of choice (Python), and promptly got lost in the all the pre-written code and black magic that seemed to happen under the hood.

This post is an attempt to start from scratch, take a simple function and expose it via a gRPC interface.

So, let's get building.

· · ·

0. Define the function

Let's create a function (*procedure*) that we want to expose (*remotely call*) — `square_root`, located in `calculator.py`

```python
import math

def square_root(x):
  y = math.sqrt(x)
  return y
```

calculator.py hosted with ❤ by GitHub view raw

`square_root` take an input `x` and returns the square root as `y`. The rest of this post will focus on how `square_root` can be exposed via gRPC.

· · ·

1. Set up protocol buffers

Protocol buffers are a language-neutral mechanism for serializing structured data. Using it comes with the requirement to explicitly define values and their data types.

Let's create `calculator.proto`, which defines the `message` and `service` structures to be used by our service.

```proto
syntax = "proto3";

message Number {
    float value = 1;
}

service Calculator {
    rpc SquareRoot(Number) returns (Number) {}
}
```

calculator.proto hosted with ❤ by GitHub view raw

You can think of the `message` and `service` definitions as below:

- `Number.value` will be used to contain variables `x` and `y`
- `Calculator.SquareRoot` will be used for the function `square_root`

. . .

2. Generate gRPC classes for Python

This section is possibly the most *"black-boxed"* part of the whole process. We will be using special tools to automatically generate classes.

New files (and classes), following certain naming conventions, will be generated when running these commands. *(You can refer to the documentation on the various flags used. In this post, all files are located in a single folder and the commands are run in that same folder.)*

```
$ pip install grpcio
$ pip install grpcio-tools

$ python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=.
calculator.proto
```

The files generated will be as follows:

`calculator_pb2.py` — contains message classes

- `calculator_pb2.Number` for request/response variables (`x` and `y`)

`calculator_pb2_grpc.py` — contains server and client classes

- `calculator_pb2_grpc.CalculatorServicer` for the server
- `calculator_pb2_grpc.CalculatorStub` for the client

. . .

3. Create a gRPC server

We now have all the pieces required to create a gRPC server, `server.py` as below. Comments, inline, should explain each section.

```python
import grpc
from concurrent import futures
import time

# import the generated classes
import calculator_pb2
import calculator_pb2_grpc

# import the original calculator.py
import calculator

# create a class to define the server functions, derived from
# calculator_pb2_grpc.CalculatorServicer
class CalculatorServicer(calculator_pb2_grpc.CalculatorServicer):

    # calculator.square_root is exposed here
    # the request and response are of the data type
    # calculator_pb2.Number
    def SquareRoot(self, request, context):
        response = calculator_pb2.Number()
        response.value = calculator.square_root(request.value)
        return response


# create a gRPC server
server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))

# use the generated function `add_CalculatorServicer_to_server`
# to add the defined class to the server
calculator_pb2_grpc.add_CalculatorServicer_to_server(
        CalculatorServicer(), server)

# listen on port 50051
print('Starting server. Listening on port 50051.')
server.add_insecure_port('[::]:50051')
server.start()

# since server.start() will not block,
# a sleep-loop is added to keep alive
try:
    while True:
        time.sleep(86400)
except KeyboardInterrupt:
    server.stop(0)
```

server.py hosted with ❤ by **GitHub** view raw

We can start the server using the command,

```
$ python server.py
Starting server. Listening on port 50051.
```

Now we have a gRPC server, listening on port `50051`.

. . .

4. Create a gRPC client

With the server setup complete, we create `client.py` — which simply calls the function and prints the result.

```python
import grpc

# import the generated classes
import calculator_pb2
import calculator_pb2_grpc

# open a gRPC channel
channel = grpc.insecure_channel('localhost:50051')

# create a stub (client)
stub = calculator_pb2_grpc.CalculatorStub(channel)

# create a valid request message
number = calculator_pb2.Number(value=16)

# make the call
response = stub.SquareRoot(number)

# et voilà
print(response.value)
```

client.py hosted with ❤ by GitHub view raw

That's it!

With the server already listening, we simply run our client.

```
$ python client.py
```

```
4.0
```

. . .

Taking it from the top

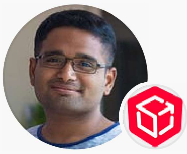
All the files can be found on GitHub at ramananbalakrishnan/basic-grpc-python. For quick reference, here is what each file is used for.

```
basic-grpc-python/
├── calculator.py          # module containing a function
│
├── calculator.proto       # protobuf definition file
│
├── calculator_pb2_grpc.py # generated class for server/client
├── calculator_pb2.py      # generated class for message
│
├── server.py              # a server to expose the function
└── client.py              # a sample client
```

This post, using a *very* simple example to convert a function into a remote procedure, just scratches the surface.

Of course, gRPC can be used in more advanced modes (*request-streaming, response-streaming, bidirectional-streaming)* with additional features such as error-handling and authentication. But hey, we all have to begin somewhere and I hope this post serves as a good reference for those just starting out.

Python Grpc Rpc Programming



Follow

Written by Ramanan Balakrishnan

147 Followers · Writer for Engineering@Semantics3