Article
José Pereira · May 2, 2022 16m read

gRPC - what is it and a hello world

Introduction

This article aims to give an introduction to what gRPC is and an example of how to play with the official Hello World using IRIS Embedded Python.

You can find all the code exposed here, in this project repo.

gRPC

The gRPC (gRPC remote procedure call) is an API architectural style based on the RPC protocol. The project was created by Google in 2015 and is licensed under Apache 2.0. Currently, the project is supported by the Cloud Native Computing Foundation (CNCF).

Successful cases of its use are related to connecting services between backends, such as services in microservices style architecture.

Protocol buffer

Most RPC-based protocols use an IDL (interface description language) to define a communication contract between a server and a client.

The gRPC uses a serializer mechanism format called Protocol Buffer.

The purpose of such a format is similar to a WSDL, where you can define methods and data structures. However, unlike WSDL, which is defined using XML, Protocol Buffer uses a language (protocol buffer language) similar to a blend of most common languages.

For instance, to define a message to interoperate information, you can use the following protocol buffer definition:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

You can also define service methods contracts for messages. For instance:

```
// 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;
}
```

The use of protocol buffers in gRPC makes it follow a functional rather than resource-based design principle, used by REST.

gRPC tools

What you define in the protocol buffers language, cannot be used directly. You should transpile the protocol buffers language to another one, which is supported by gRPC.

Such transpiling is done by a package called gRPC tools. Currently, the gRPC platform supports such languages as Java, C++, Dart, Python, Objective-C, C#, Ruby, JavaScript and Go.

In this article, we are going to employ Python support in order to use gRPC with the Embedded Python feature in IRIS.

For instance, with this command from gRPC tools, you can transpile the protocol buffers definition for the Greeter service, the HelloRequest and HelloReply messages to Python:

```
python3 -m grpc_tools.protoc -I ../../protos --python_out=. --grpc_python_out=. ../..
/protos/helloworld.proto
```

Such command produces these Python files:

```
-rwxrwxrwx  1 irisowner irisowner 2161 Mar 13 20:01 helloworld_pb2.py*
-rwxrwxrwx  1 irisowner irisowner 3802 Mar 13 20:01 helloworld_pb2_grpc.py*
```

These files are the Python source code generated from the proto file for messages and service methods, respectively. The server implements the service methods, and the client (also called stub) calls them.

Thus, you can use Embedded Python to send/receive Hello messages through the Greeter service.

Another useful tool for gRPC is one equivalent to curl, the grpcurl utility. After introducing our hello world example, an overview of how to use such a tool will be presented.

Service methods types

The clients may vary based on how they send and receive messages from service methods. Messages could be sent and received one per call or in a stream. For each combination, gRPC has a service method type:

- Simple RPC or Unary: clients send a simple message and receive a simple response from the server, i.e., a standard function call;
- Response-streaming or server streaming: clients send a simple message and receive a stream of

messages from the server;
- Request-streaming or client streaming: clients send a stream of messages and receive a simple message from the server;
- Bidirectional-streaming: clients send a stream of messages and receive another stream of messages from the server.

Communication through such methods could be async (default) or sync, depending on the client ' s needs.

The gRPC core-concepts page defines those types as features from the gRPC life cycle. It also highlights other features which are not under the scope of this introduction, but you can check the links below if you would like to get more information:

- Deadlines/Timeouts
- RPC termination
- RPC cancelation
- Metadata
- Channels

Pros and cons

Here are some pros and cons I have found in some articles:

Pros:

- Lighter messages. The Protocol buffer is a binary format, so it avoids the JSON overhead created by special characters.
- Fast serialization/deserialization. Again, due to its binary format, protocol buffers could be serialized/deserialized into client stubs using specific languages without interpreters.
- Built-in clients (stubs). Protocol buffers have built-in generators for most used languages, unlike JSON which depends on third-party tools like OpenAPI and its client generators.
- Parallel requests. HTTP/1 allows up to 6 simultaneous connections, blocking any other requests until all 6 connections are finished - an issue known as HOL (head of line blocking); HTTP/2 fixes such limitations.
- Contract-first API design. Although REST APIs could expose a contract through third-party tools like OpenAPI, in gRPC such a contract is explicitly declared by the protocol buffers.
- Native streaming. Thanks to the streaming capabilities of HTTP/2, gRPC allows a native bidirectional streaming model, unlike REST, which must mimic such behavior over HTTP/1.

Cons:

- No flexibility in protocol buffers (loose server coupling).
- No human readability in protocol buffers.
- A way bigger amount of specialized people, resources and tools/projects for REST/JSON than for gRPC/protocol buffers.

However, those pros and cons are not a consensus. They are highly dependent on the needs of your application.

For instance, a supposed disadvantage of REST/JSON is the need for third-party tools like OpenAPI. However, it might not be an issue at all since such tools are heavily supported, maintained and used by several development communities/companies around the world.

On the other hand, if your project needs to deal with complexities that gRPC could address better than REST, you should select gRPC, even if that decision would bring some complications, like creating a qualified developer team.

Where/when to use gRPC?

Here are some use cases when gRPC might be a good choice:

- Microservices communication
- Client/server application, where clients run on limited hardware and/or network, assuming that HTTP/2 is available
- Interoperability facilitated by a strong contract API design

gRPC players

Some big techs are utilizing gRPC to deal with specific challenges:

- Salesforce: gRPC leads the company ' s platform with an increase in interoperability robustness due to the strong contract design given by protocol buffers.
- Netflix: uses gRPC to improve its microservices environment.
- Spotify: similar to Netflix, uses gRPC to handle microservices challenges and to deal with lots of APIs.

Hello world using Embedded Python

Ok, after a brief introduction of what gRPC is and what it does, you now can manage it, so let ' s play around a bit. As this article title says, this is just an adaptation of the original hello world example using IRIS Embedded Python.

This example is heavily based on those ones available in the gRPC sample repository: helloworld and hellostreamingworld. With the help of it, I would like to show you how to send and receive a simple message in single and stream modes. Regardless of it being a simple example without really useful features, it will help you to understand the main concepts related to developing a gRPC application.

Installing gRPC for Python

First, let ' s install the required package for using gRPC in Python. If you are running the example from the container defined in my github project, you probably have these packages already installed, so you can skip this step.

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade --target /usr/irissys/mgr/python grpcio grpcio-tools
```

Defining the service contract

Now, let ' s see the service contract, i.e. the protocol buffer file (or just protobuf for short), with the schema of the messages and available methods:

```
syntax = "proto3";

package helloworld;

// The greeting service definition.
service MultiGreeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends multiple greetings
  rpc SayHelloStream (HelloRequest) returns (stream HelloReply) {}
}

// The request message containing the user's name and how many greetings
```

```
// they want.
message HelloRequest {
  string name = 1;
  Int32 num_greetings = 2;
}

// A response message containing a greeting
message HelloReply {
  string message = 1;
}
```

This protobuf file defines a service called MultiGreeter with two RPC methods, SayHello() and SayHelloStream().

The SayHello() method receives a HelloRequest message and sends a HelloReply message. Similarly, the method SayHelloStream() receives and sends the same messages, but it sends a stream of HelloRequestmessages instead of a single one.

After the service definition, there are the definitions of the messages, HelloRequest and HelloReply. The HelloRequest message just encapsulates two fields: a string one called name and an integer one called numgreetings. The HelloReply message contains only one field, a string one called message.

The numbers after the fields are called field numbers. These numbers must not be changed once the message is being used because they act as identifiers.

Generating Python code from the service contract

As you could probably notice, we do not need to write any code in the protobuf definition, only interfaces. The task of implementing the code for different programming languages is done by the protobuf compiler protoc. There is a protoc compiler for each of the languages supported by gRPC.

For Python, the compiler is deployed as a module called grpctools.protoc.

To compile the protobuf definition to Python code, execute the following command (assuming you are using my project):

```
cd /irisrun/repo/jrpereira/python/grpc-test
/usr/irissys/bin/irispython -m grpc_tools.protoc -I ./ --python_out ./ --grpc_python_
out ./ helloworld.proto
```

This command invokes the module grpctools.protoc - the protoc compiler for Python, with the following parameters:

- helloword.proto: the main .proto file for the service contract
- -I: the location for .proto files where the compiler will look for dependencies
- --pythonout: the location of generated Python code for messages
- --grpcpythonout: the location of generated Python code for a server and a stub (client) based on RPC methods in the service definition

In this case, all those location parameters are set up for the current directory.

The code generated by the protoc compiler is not the best example of readability, although it is not that hard to understand it. You can check it out in the directory passed to the protoc compiler.

Anyway, those files are intended to be imported into your own code, so let ' s use them, implementing a server and a

client.

Implementing a server for the service

In order to implement a server for the service defined above, let ' s use Embedded Python.

First, let ' s define a server using a Python file where the server logic is implemented. I have decided to implement it this way due to the need to use a Python concurrency library.

```
"""
The Python implementation of the GRPC helloworld.Greeter server.
Adapted from:
    - https://github.com/grpc/grpc/blob/master/examples/python/helloworld/async_greet
er_server.py
    - https://github.com/grpc/grpc/blob/master/examples/python/hellostreamingworld/as
ync_greeter_server.py
    - https://groups.google.com/g/grpc-io/c/6Yi_oIQsh3w
"""

from concurrent import futures
import logging
import signal
from typing import Any
from argparse import ArgumentParser, ArgumentDefaultsHelpFormatter

import grpc
from helloworld_pb2 import HelloRequest, HelloReply
from helloworld_pb2_grpc import MultiGreeterServicer, add_MultiGreeterServicer_to_ser
ver

import iris

NUMBER_OF_REPLY = 10

parser = ArgumentParser(formatter_class=ArgumentDefaultsHelpFormatter)
parser.add_argument("-p", "--port", default="50051", help="Server port")
args = vars(parser.parse_args())

class Greeter(MultiGreeterServicer):

    def SayHello(self, request: HelloRequest, context) -> HelloReply:
        logging.info("Serving SayHello request %s", request)
        obj = iris.cls("dc.jrpereira.gRPC.HelloWorldServer")._New()
        # hook to your ObjectScript code
        return obj.SayHelloObjectScript(request)

    def SayHelloStream(self, request: HelloRequest, context: grpc.aio.ServicerContext
) -> HelloReply:
        logging.info("Serving SayHelloStream request %s", request)
        obj = iris.cls("dc.jrpereira.gRPC.HelloWorldServer")._New()
        n = request.num_greetings
        if n == 0:
            n = NUMBER_OF_REPLY
        for i in range(n):
            # hook to your ObjectScript code
            yield obj.SayHelloObjectScript(request)
```

```
def get_server():
    port = args["port"]
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    add_MultiGreeterServicer_to_server(Greeter(), server)
    listen_addr = f"[::]:{port}"
    server.add_insecure_port(f"[::]:{port}")
    logging.info("Starting server on %s", listen_addr)
    return server

def handle_sigterm(*_: Any) -> None :
    """Shutdown gracefully."""
    done_event = server.stop(None)
    done_event.wait(None)
    print('Stop complete.')

logging.basicConfig(level=logging.INFO)

server = get_server()
server.start()

# https://groups.google.com/g/grpc-io/c/6Yi_oIQsh3w
signal.signal(signal.SIGTERM, handle_sigterm)

server.wait_for_termination()
```

As you can see, here the methods defined in the protobuf specification - SayHello() and SayHelloStream(), are implemented.

The method SayHello() just sends a single value, whereas the method SayHelloStream() returns a number of messages to the client equal to NUMBER_OF_REPLY, through the Python operator yield.

Also, note that I have created a hook to inject logic defined in ObjectScript. Thus, I defined a method called SayHelloObjectScript in dc.jrpereira.gRPC.HelloWorldServer class:

```
Method SayHelloObjectScript(request)
{
    Set sys = $system.Python.Import("sys")
    Do sys.path.append("/usr/irissys/mgr/python/grpc-test/")

    Set helloworldpb2 = $system.Python.Import("helloworld_pb2")

    Set reply = helloworldpb2.HelloReply()
    Set reply.message = "Hi "_request.name_"! :)"

    Return reply
}
```

In this way, you can receive requests from gRPC clients from Python and process them using a mix of logic coded in Python and ObjectScript.

Implementing a client for the service

As a client code does not need concurrency, I have implemented it using Python code directly in an ObjectScript class:

```
ClassMethod ExecutePython() [ Language = python ]
{
    import sys
    sys.path.append('/usr/irissys/mgr/python/grpc-test/')

    import grpc
    from helloworld_pb2 import HelloRequest
    from helloworld_pb2_grpc import MultiGreeterStub

    channel = grpc.insecure_channel('localhost:50051')
    stub = MultiGreeterStub(channel)

    response = stub.SayHello(HelloRequest(name='you'))
    print("Greeter client received: " + response.message)

    for response in stub.SayHelloStream(HelloRequest(name="you")):
        print("Greeter client received from stream: " + response.message)
}
```

First, we add the grpc-test directory to the Python path in order to be able to import the generated protobuf code.

Then, a connection to localhost on port 50051 and a stub (or client) are created.

With such a client, we can request info for the server listening on localhost:50051, through methods SayHello() and SayHelloStream().

The method SayHello() just returns a single value, so we just need to make a request and use its response. On the other hand, the method SayHelloStream() returns a stream of data in a collection, so we need to iterate through it in order to get all its data.

Testing the code

Ok, now let ' s test our code.

You can check out all this code in my hello world project. Follow these steps to get it running:

```
git clone https://github.com/jrpereirajr/iris-grpc-example
cd iris-grpc-example
docker-compose up -d
```

Then, open an IRIS terminal through the system terminal or through Visual Studio Code:

```
docker exec -it iris-grpc-example_iris_1 bash
iris session iris
```

Start our gRPC server:

```
Set server = ##class(dc.jrpereira.gRPC.HelloWorldServer).%New()
Do server.Start()
```

Now, let ' s create a gRPC client to interact with this server:

```
Set client = ##class(dc.jrpereira.gRPC.HelloWorldClient).%New()
Do client.ExecutePython()
```

If everything is OK, you should see a bunch of greeting messages in the terminal.

Finally, let's stop the server:

```
Do server.Stop()
```

Using the grpcurl utility within our hello world

As I said before, the grpcurl utility is an equivalent to curl, but here instead of acting like an HTTP client (like curl), we use grpcurl as a gRPC client to test services from a running gRPC server. So, let ' s use it to play a little bit more with our hello world.

First, let ' s download and install the grpcurl utility:

```
cd /tmp
wget https://github.com/fullstorydev/grpcurl/releases/download/v1.8.6/grpcurl_1.8.6_l
inux_x86_64.tar.gz
tar -zxvf grpcurl_1.8.6_linux_x86_64.tar.gz
```

Check if the installation is OK, by typing:

```
./grpcurl --help
```

If everything is OK, you should receive an output with all grpcurl options.

Now, let ' s ask what services are available in the server:

```
./grpcurl \
    -plaintext \
    -import-path /irisrun/repo/jrpereira/python/grpc-test \
    -proto helloworld.proto \
    localhost:50051 \
    list
```

You should receive this response:

```
helloworld.MultiGreeter
```

As you can see, the utility returned our service defined in the proto file (helloworld.MultiGreeter) as a response for listing all available services .

In the command above, I put each parameter in a separated line. So, let ' s explain each one:

-plaintext: allows using gRPC with no TLS (insecure mode); we are using it here because we did not implement a secure connection for our server. Of course, it should be used only in a non-production environment
-import-path and -proto: path and name for the .proto file (service definition); necessary if the server does not implement reflection

After these parameters, we provide the server hostname and port, and then a grpcurl command - list in this case.

Now, let ' s ask for all the methods in the service helloworld.MultiGreeter:

```
./grpcurl \
    -plaintext \
    -import-path /irisrun/repo/jrpereira/python/grpc-test \
    -proto helloworld.proto \
    localhost:50051 \
    list helloworld.MultiGreeter
```

You should receive this output:

```
helloworld.MultiGreeter.SayHello
helloworld.MultiGreeter.SayHelloStream
```

As you can see, these are the methods defined in the proto file used to generate code for our server.

Ok, now let ' s test the SayHello() method:

```
./grpcurl \
    -plaintext  \
    -d '{"name":"you"}' \
    -import-path /irisrun/repo/jrpereira/python/grpc-test \
    -proto helloworld.proto \
    localhost:50051 \
    helloworld.MultiGreeter.SayHello
```

Here is the expected output (just like the one our client implemented earlier):

```
{
  "message": "Hi you! :)"
}
```

Also, let ' s test the other method, SayHelloStream():

```
./grpcurl \
    -plaintext -d '{"name":"you"}' \
    -import-path /irisrun/repo/jrpereira/python/grpc-test \
    -proto helloworld.proto localhost:50051 \
    helloworld.MultiGreeter.SayHelloStream
```

And, we should got a stream with 10 greeting messages:

```
{
  "message": "Hi you! :)"
}
{
  "message": "Hi you! :)"
}
...
{
  "message": "Hi you! :)"
}
```

Finally, let ' s do a slight change on this command to use another property in the protobuf message, the numgreetings one. This property is used by the server to control how many messages will be sent in the stream.

So, this command asks the server to return only 2 messages in the stream, instead of 10 by default:

```
./grpcurl \
    -plaintext -d '{"name":"you", "num_greetings":2}' \
    -import-path /irisrun/repo/jrpereira/python/grpc-test \
    -proto helloworld.proto localhost:50051 \
    helloworld.MultiGreeter.SayHelloStream
```

And this should be what you will see in the terminal:

```
{
  "message": "Hi you! :)"
}
{
  "message": "Hi you! :)"
}
```

Conclusion

In this article, an overview of gRPC was given, with its pros and cons - mainly over REST. Also, some examples of its use with IRIS were tested, by adapting some samples for Python, presented in the official gRPC repo.

As said earlier, gRPC has some use cases where it has been employed, and interoperability is one of the possible ones, so making an IRIS Interoperability adapter is a natural way of thinking in a practical use for gRPC within IRIS.

However, it will demand more effort, so it will be the subject of another article. =)

Hope you found the information presented here useful! See you!

References

https://grpc.io/docs/what-is-grpc/introduction/
https://developers.google.com/protocol-buffers
https://en.wikipedia.org/wiki/GRPC
https://www.imaginarycloud.com/blog/grpc-vs-rest/
https://www.vineethweb.com/post/grpc/
https://www.capitalone.com/tech/software-engineering/grpc-framework-for-...
https://www.altexsoft.com/blog/what-is-grpc/

#Embedded Python #InterSystems IRIS