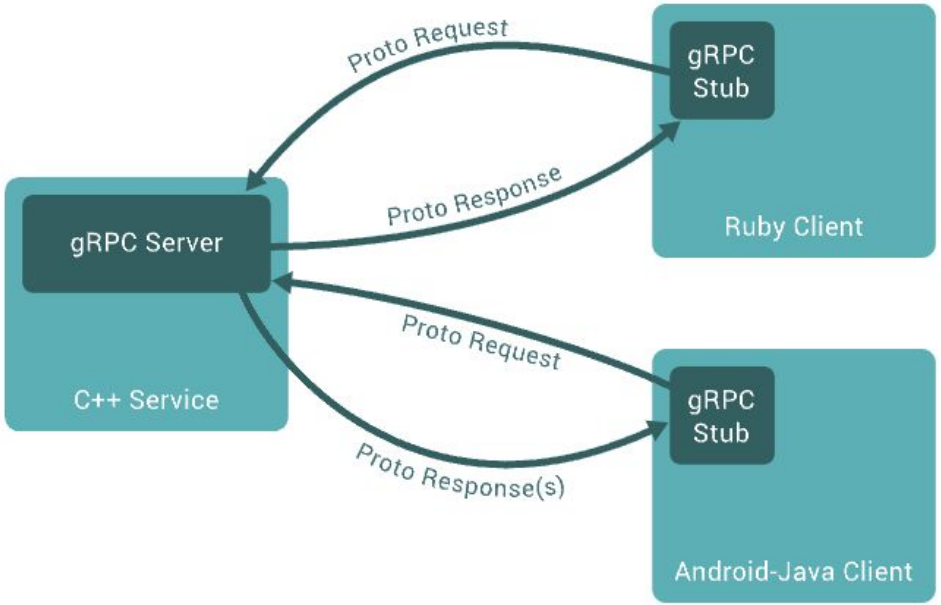
gRPC

remote procedure call framework

How it works


Proto Request
gRPC Stub
Proto Response
Ruby Client
gRPC Server
C++ Service
Proto Request
gRPC Stub
Proto Response(s)
Android-Java Client

Cross-language tool

- C# / .NET
- C++
- Dart
- Go
- Java
- Kotlin
- Node
- Objective-C
- PHP
- Python
- Ruby

Protocol buffer

.proto files

https://protobuf.dev/programming-guides/proto3/

```
syntax = "proto3";

service Greeter{

 rpc SayHello(Request) returns (Response) {}

}

message Request{
 string name = 1;
}

message Response{
 string message = 1;
}
```

Example service configuration

```
def serve():
    port = '50051'
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    helloworld_pb2_grpc.add_GreeterServicer_to_server(Greeter(), server)
    server.add_insecure_port('[::]:' + port)
    server.start()
    print("Server started, listening on " + port)
    server.wait_for_termination()
```

Example service class

```
class Greeter(helloworld_pb2_grpc.GreeterServicer):

    Yash Tibrewal
    def SayHello(self, request, context):
        return helloworld_pb2.HelloReply(message='Hello, %s!' % request.name)
```

Stream usage in single rpc

```
service Service {

  rpc StreamCall(stream StreamCallRequest) returns (stream StreamCallResponse);
}
```

Client call example

```
def run():
    print("Will try to greet world ...")
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = helloworld_pb2_grpc.GreeterStub(channel)
        response = stub.SayHello(helloworld_pb2.HelloRequest(name='you'))
        print("Greeter client received: " + response.message)
```

Bidirectional streaming

Server handle example

```
def RouteChat(self, request_iterator, context):
  prev_notes = []
  for new_note in request_iterator:
    for prev_note in prev_notes:
      if prev_note.location == new_note.location:
        yield prev_note
    prev_notes.append(new_note)
```

```
def ListFeatures(self, request, context):
  left = min(request.lo.longitude, request.hi.longitude)
  right = max(request.lo.longitude, request.hi.longitude)
  top = max(request.lo.latitude, request.hi.latitude)
  bottom = min(request.lo.latitude, request.hi.latitude)
  for feature in self.db:
    if (feature.location.longitude >= left and
        feature.location.longitude <= right and
        feature.location.latitude >= bottom and
        feature.location.latitude <= top):
      yield feature
```

Client call example

```
for feature in stub.ListFeatures(rectangle):
```

```
for received_route_note in stub.RouteChat(sent_route_note_iterator):
```

```
request_iterator = iter(
    [
        GreetingRequest(name="Alice"),
        GreetingRequest(name="Bob")
    ]
)
response_iterator = greeter_stub.Chat(request_iterator)
```

```python
import asyncio

from grpclib.utils import graceful_exit
from grpclib.server import Server

# generated by protoc
from .helloworld_pb2 import HelloReply
from .helloworld_grpc import GreeterBase


class Greeter(GreeterBase):

    async def SayHello(self, stream):
        request = await stream.recv_message()
        message = f'Hello, {request.name}!'
        await stream.send_message(HelloReply(message=message))


async def main(*, host='127.0.0.1', port=50051):
    server = Server([Greeter()])
    # Note: graceful_exit isn't supported in Windows
    with graceful_exit([server]):
        await server.start(host, port)
        print(f'Serving on {host}:{port}')
        await server.wait_closed()


if __name__ == '__main__':
    asyncio.run(main())
```

```python
import asyncio

from grpclib.client import Channel

# generated by protoc
from .helloworld_pb2 import HelloRequest, HelloReply
from .helloworld_grpc import GreeterStub


async def main():
    async with Channel('127.0.0.1', 50051) as channel:
        greeter = GreeterStub(channel)

        reply = await greeter.SayHello(HelloRequest(name='Dr. Strange'))
        print(reply.message)


if __name__ == '__main__':
    asyncio.run(main())
```

Async

First task

Your first task is to implement the base hello world procedure.

Look on helloworld.proto there is all you need to implement

server and client communication

Second task

Your second task is to create server that perform math tasks:

- addition
- subtraction
- multiplication
- division
- exponentiation

Third task

Your last task is to improve the usage of the server from the second task with ping-pong stream type that allow sending more data in a single procedure call. Add some error handling using context.

https://adityamattos.com/grpc-in-python-part-3-implementing-grpc-streaming

Here you can find them all.

git: https://github.com/AdrianKarolewski/gRPC-tutorial

https://grpc.io/docs/languages/python/quickstart/

if you don't complete them in classes, it's possible to send .zip or your GitHub url to me by email at **adrian.p.karolewski@student.put.poznan.pl** ***deadline - 15.06 22:00***