gRPC

RPC the right way

Bruno Mendes

Fernando Rego

José Costa

TABLE OF CONTENTS

01

What is RPC?

A small introduction on RPC and its history

02

gRPC (and protobuf)

Core concepts, architecture, lifecycle and features

03

gRPC for microservices

Advantages and downsides in a microservices context

04

Demo

A calculator app using gRPC to communicate between heterogeneous architecture components.

"if bridge building were like programming, halfway through we'd find out that the far bank was now 50 meters farther out, that it was actually mud rather than granite, and that rather than building a footbridge we were instead building a road bridge."

–Sam Newman, Building Microservices: Designing Fine-Grained Systems

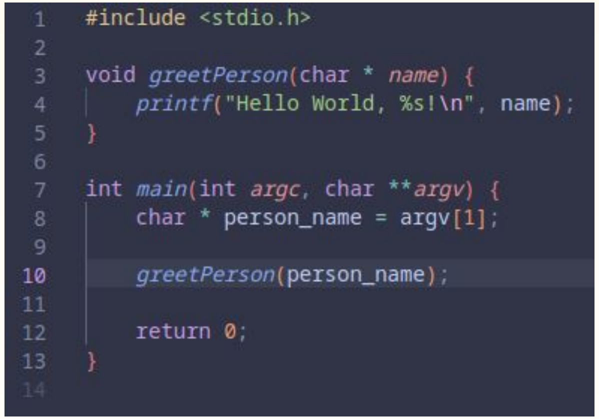
01

What is RPC?

A small introduction on RPC and its history

Context

- Calling a local procedure



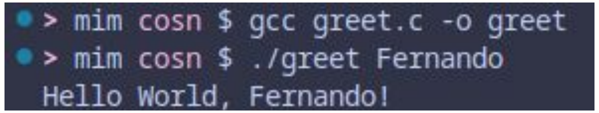
```c
#include <stdio.h>

void greetPerson(char * name) {
    printf("Hello World, %s!\n", name);
}

int main(int argc, char **argv) {
    char * person_name = argv[1];

    greetPerson(person_name);

    return 0;
}
```



```
> mim cosn $ gcc greet.c -o greet
> mim cosn $ ./greet Fernando
Hello World, Fernando!
```

What if we want to call a procedure in a different address space, i.e., remote computer?

Concept

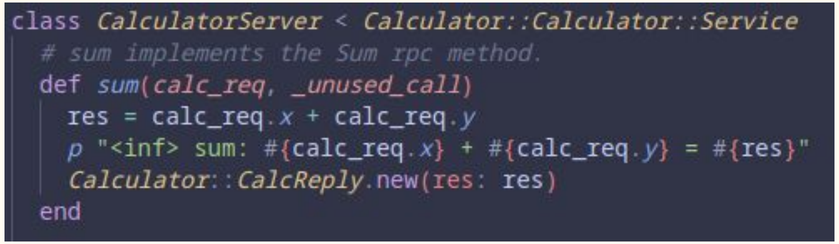
Make remote calls just as simple as local calls

- RPC is a protocol that enables users to work with remote procedures as if it were a local procedure call without explicit details about remote interactions
- Implemented by various platforms and programming languages (e.g. Java RMI for Java and RPyC for Python)

History:

- 1970s - theoretical proposals of remote procedure calls
- 1980s - practical implementations of RPC
- 1981 - Bruce J. Nelson became known as the creator of the RPC

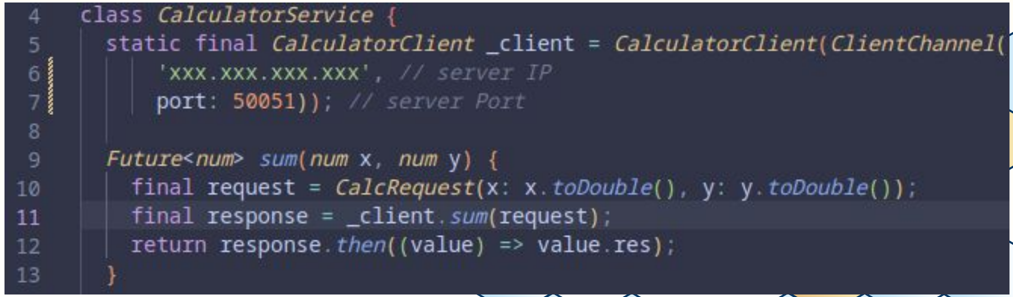
Example: generated code



```ruby
class CalculatorServer < Calculator::Calculator::Service
  # sum implements the Sum rpc method.
  def sum(calc_req, _unused_call)
    res = calc_req.x + calc_req.y
    p "<inf> sum: #{calc_req.x} + #{calc_req.y} = #{res}"
    Calculator::CalcReply.new(res: res)
  end
```

Server - developed using Ruby

Client - developed using Dart



```dart
class CalculatorService {
  static final CalculatorClient _client = CalculatorClient(ClientChannel(
      'xxx.xxx.xxx.xxx', // server IP
      port: 50051)); // server Port

  Future<num> sum(num x, num y) {
    final request = CalcRequest(x: x.toDouble(), y: y.toDouble());
    final response = _client.sum(request);
    return response.then((value) => value.res);
  }
```

02

gRPC (and protobuf)

Core concepts, architecture, lifecycle and features

What is gRPC? - introduction

- gRPC (Google Remote Procedure Call) is an **open-source remote procedure call** system.
- Developed initially by Google in 2015.
- Uses **Protocol Buffers** and enables **authentication**, **bidirectional streaming** and **flow control**, among other features.
- This technology is used by various companies, such as Netflix, Cisco, and others.

gRPC - presentation

What does gRPC stand for?

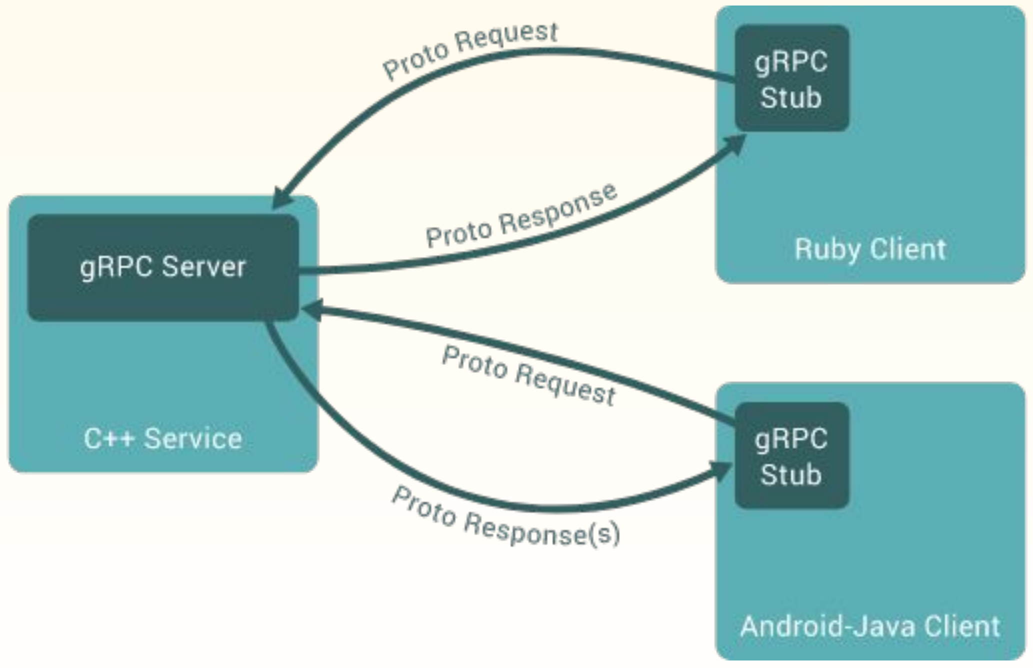
- gRPC Remote Procedure Calls, of course!

But what it is?

- gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

More on gRPC: History and Principles

gRPC - architecture overview



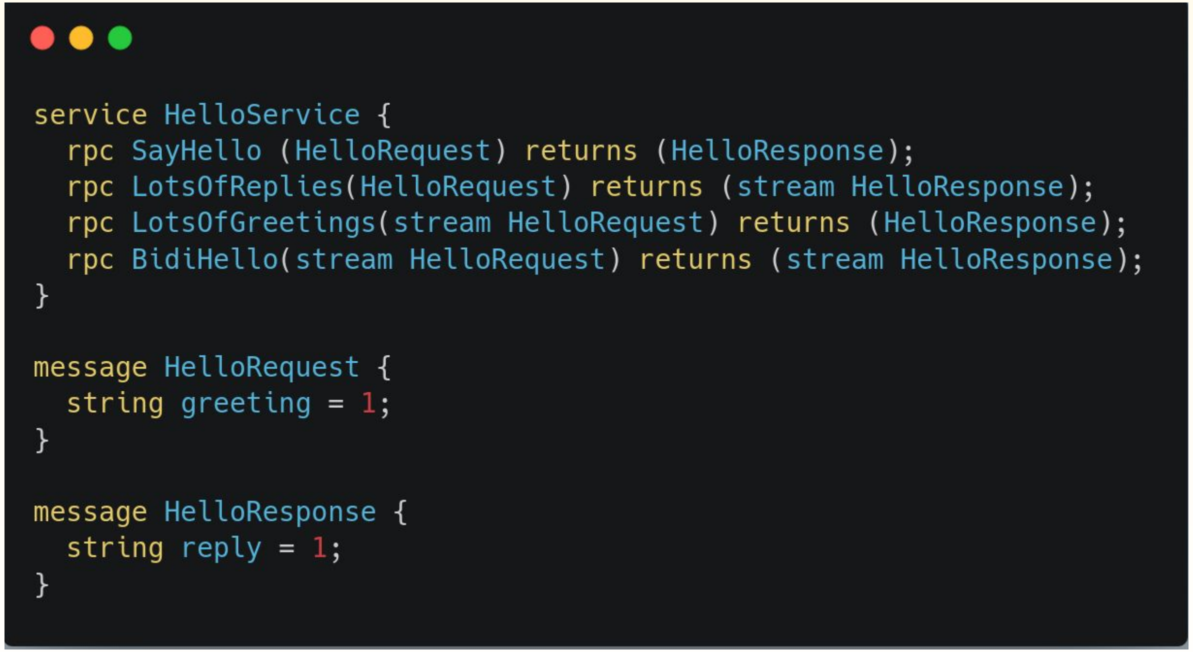
gRPC - protobuf

Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.

Main features:

- Smaller and faster than JSON
- Native language bindings
- Allow for update of definition without code update

gRPC - protobuf



```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
  rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
  rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
  rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

gRPC - core concepts

- Synchronous and asynchronous flavours
- Deadlines/timeouts
  Specification is languanguage specific (using durations or fixed points in time)
- Independent and local termination
  Server and client may not agree on call success
- Server or client can cancel RPC at any time
- Metadata
  Extra information about a specific RPC call
- Channels
  Provides a connection to a server on a specified host and port

gRPC - extra features

- Authentication
- Compression
- Custom metrics and load balancing
- Health checking
- Metadata
- Wait-for-ready
- Cascading call-cancellation

03

gRPC for microservices

Advantages and downsides in a microservices context

gRPC for microservices - pros

Performance	Better support for large data transfers (binary encoding)
Language agnostic	Support for heterogeneous implementations, using different technologies (generated code)
Ease of use	Strong typing and well defined interfaces reduce errors and improve developer experience
Highly extensible	Interfaces can be extended without introducing breaking changes to deployed code

gRPC for microservices - cons

- Initial setup
- Possibly unnecessary complexity
- Tech stack restrictions
- Problems when handling failed distributed transactions

04

Demo

A calculator app using gRPC to communicate between heterogeneous architecture components

References

- https://grpc.io/docs/
- https://codeopinion.com/where-should-you-use-grpc-and-where-not/
- https://github.com/fernandorego/feup-cosn-grpc-calc-demo