gRPC

by

Dr. Günter Kolousek

gRPC

- high performance RPC Framework
  - $\rightarrow$ Microservices
- cross-platform
  - Windows, Linux, MacOS, Android, iOS
  - Python, C++, Java, C#, PHP, Node.js, Ruby, Go, Objective-C
    - weitere: Swift, Dart, Rust, Haskell, Erlang, Elixir,…
- für: ”low latency, highly scalable, distributed systems”
  - wird auch für IoT propagiert (”power-efficient”)
- mittels Extensions: load balancing, tracing, health checking, Authentifizierung
- basiert auf
  - protobuf Version 3
  - HTTP/2
- Google, aber: open source

Name?

- abhängig von Version!
- 1.0: g … gRPC
- 1.1: g … good
- 1.2: g … green
- 1.3: g … gentle
- …
- 1.18: g … goose (Gans?)
- 1.19: g … gold

Quelle: https://github.com/grpc/grpc/blob/master/doc/g_stands_for.md

Features

- (bidirektionales) Streaming
  - nicht nur Request/Response
    - $\rightarrow$ HTTP/2
- protobuf (anstatt z.B. JSON) $\rightarrow$ schnelles Serialisierien und Deserialisieren!
  - 2016: Google Data Center: ca. 10 Milliarden RPC Calls/s
- kürzere Entwicklungszeit
  - neue Endpunkte leicht aufsetzbar
- Kontext je Funktionsaufruf $\rightarrow$ Daten über APIs weiterreichen
- eingebaute Authentifizierung
  - je Funktionsaufruf oder je Verbindung
- Eigenschaften…
  - lightweight, efficient, scalable

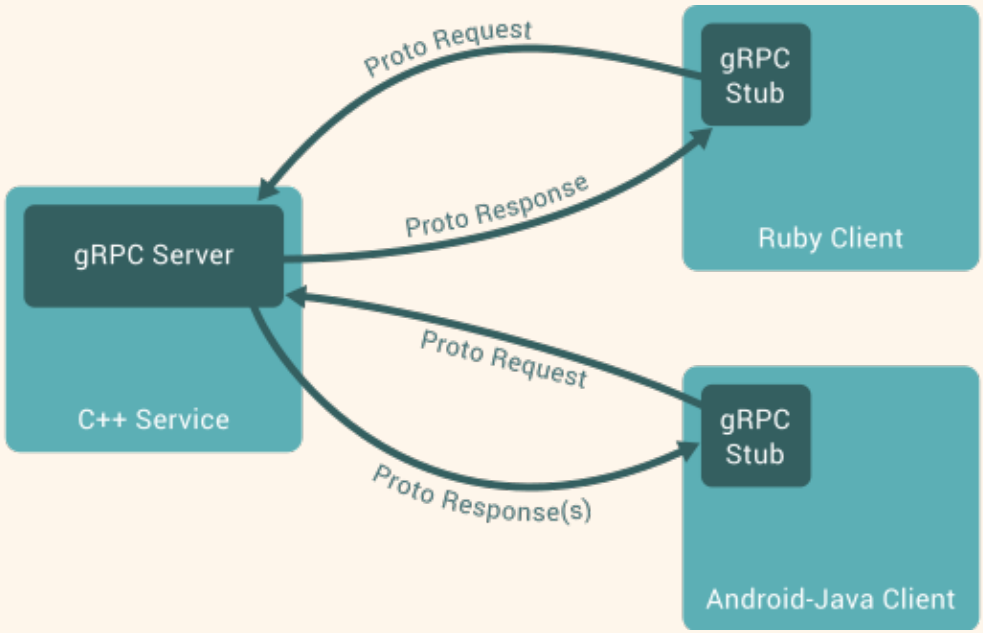
Einsatz und Alternativen

- Einsatz
  - Microservices
  - (meist) intern
- Alternativen
  - CORBA, DCOM, Java RMI,…
  - XML-RPC, SOAP mit WSDL (und UDDI)
  - JSON-RPC
  - REST, GraphQL

gRPC vs. REST

- gRPC
  - schneller
    - Latenz
    - Durchsatz
    - Wiederverwendung und gleichzeitige Verwendung von Verbindungen ($\rightarrow$ HTTP/2)
  - robuster $\rightarrow$ IDL, Versionen
  - bequemer
  - Unterstützung im Browser noch nicht…
- REST
  - allgemeiner
  - *alle* Programmiersprachen
  - Menschen-lesbar
  - kein Schema notwendig…

Überblick


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

Projekt aufsetzen

```
$ python -m venv example
$ source example/bin/activate
$ cd example
$ python -m pip install --upgrade pip
...
$ python -m pip install grpcio
...
$ python -m pip install grpcio-tools
...
$ mkdir proto
$ mkdir client
$ mkdir server
```

Workflow

1. Schreiben einer Service-Definition (in `.proto`-Datei)
2. aus `.proto`-Datei gRPC Code generieren ($\rightarrow$ `protoc`)
3. Server implementieren (z.B. in C++)
4. Client schreiben, der Service in Anspruch nimmt
   - mittels generierten Stub
5. Server und Clients starten

Definition des Service

greeter.proto:

```
syntax = ”proto3”;

package greeter;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply)
}

message HelloRequest {
  string name = 1;
}

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

Code generieren

```
$ cd proto
$ python -m grpc.tools.protoc --python_out=.\
  --grpc_python_out=. --proto_path=. greeter.proto
$ cp *.py ../client
$ cp *.py ../server
```

Server implementieren

```
from concurrent import futures
import time, grpc
import greeter_pb2, greeter_pb2_grpc

class Greeter(greeter_pb2_grpc.GreeterServicer):
    def say_hello(self, request, context):
        return greeter_pb2.HelloResponse(message="Hello
          + request.name)
def serve():
    server = grpc.server(
      futures.ThreadPoolExecutor(max_workers=10))
    greeter_pb2_grpc.
       add_GreeterServicer_to_server(Greeter(), server)
    server.add_insecure_port('localhost:8888')
    server.start()
    try: while True: time.sleep(60)
    except KeyboardInterrupt: server.stop(0)
serve()
```

Client implementieren

```
import grpc

import greeter_pb2
import greeter_pb2_grpc

def run():
  channel = grpc.insecure_channel('localhost:8888')
  stub = greeter_pb2_grpc.GreeterStub(channel)
  response = stub.say_hello(
    greeter_pb2.HelloRequest(name='Maxi'))
  print(”Greeter client received: ” +
    response.message)

run()
```

Server und Client starten

```
$ cd server
$ python server.py&
...
$ cd ../client
$ python client.py
Greeter client received: Hello Maxi
```

Arten eines API

- unary API: Nachricht hin und zurück

  ```
  rpc say_hello(HelloRequest) returns (HelloResponse){}
  ```

- streaming
  - client streaming API: Stream von Nachrichten hin und Nachricht zurück

    ```
    rpc many_reqs(stream HelloRequest) returns
      (HelloResponse){}
    ```

  - server streaming API: Nachricht hin und Stream von Nachrichten zurück

    ```
    rpc many_replies(HelloRequest) returns
      (stream HelloResponse){}
    ```

  - bidirectional streaming API