gRPC

Aufgabe:

helloworld.proto

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.helloworld";
option java_outer_classname = "HelloWorldProto";
option objc_class_prefix = "HLW";

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}

  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}

  rpc SayHelloStreamReply (HelloRequest) returns (stream Hell

  rpc SayHelloBidiStream (stream HelloRequest) returns (strea
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
```

```
  string message = 1;
}
```

Server

```
from concurrent import futures
import logging

import grpc
import helloworld_pb2
import helloworld_pb2_grpc


class Greeter(helloworld_pb2_grpc.GreeterServicer):
    def SayHello(self, request, context):
        return helloworld_pb2.HelloReply(message="Hello, %s!"

    def SayHelloAgain(self, request, context):
        return helloworld_pb2.HelloReply(message="Hello again


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


if __name__ == "__main__":
    logging.basicConfig()
    serve()
```

Client

```
from __future__ import print_function

import logging

import grpc
import helloworld_pb2
import helloworld_pb2_grpc

def run():
    # NOTE(gRPC Python Team): .close() is possible on a chann
    # used in circumstances in which the with statement does
    # of the code.
    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(
        print("Greeter client received: " + response.message)
        response = stub.SayHelloAgain(helloworld_pb2.HelloReq
        print("Greeter client received: " + response.message)


if __name__ == "__main__":
    logging.basicConfig()
    run()
```

Fragen:

- What is gRPC and why does it work accross languages and platforms?
- Describe the RPC life cycle starting with the RPC client?
- Describe the workflow of Protocol Buffers?
- What are the benefits of using protocol buffers?
- When is the use of protocol not recommended?

- List 3 different data types that can be used with protocol buffers?

gRPC is a modern, high-performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services within and between data centers and provides support for load balancing, tracing, health checking, and authentication. It allows for the automatic generation of client and server stubs for your service in a variety of languages and platforms. Therefore, gRPC works across languages and platforms.

The RPC lifecycle begins with the definition of a service in a .proto file. gRPC uses a code generator to generate client and server code in various programming languages. The client creates a stub, a local representation of the remote service. The stub provides a method with the same signature as the remote method. The client can call this method as if it were a local function. The stub serializes the method parameters into a binary format and sends the method parameters over an HTTP/2 connection to the server.

Protocol Buffers is a language-neutral, platform-neutral, extensible mechanism for serializing structured data. You define once how you want your data structured, and then you can use specially generated source code to easily write your structured data into a variety of data streams and read from these and use in a variety of languages. The code generated by Protocol Buffers provides utility methods for retrieving data from files and streams, extracting individual values from the data, checking whether data is present, serializing data back into a file or stream, and other useful functions.

Protocol Buffers are ideal for any situation where you need to serialize structured, record-like, typed data in a language-neutral, platform-neutral, extensible way. Some of the advantages of using Protocol Buffers are compact data storage, fast parsing, availability in many programming languages, and optimized functionality through automatically generated classes.

The use of protocols is not recommended if they are outdated and have been compromised by the NSA, such as PPTP. Also, protocols such as SSL 3.0 or TLS 1.0, which are used by default by older versions of the .NET Framework like 4.5.2 or 3.5, are not recommended.

Protocol Buffers support various data types, including BoolValue for boolean values, BytesValue for byte sequences, and DoubleValue for double values.

Quellen:

[1] https://grpc.io/docs/languages/java/quickstart/

[2] https://grpc.io/

[3] https://www.makeuseof.com/grpc-what-why-use/

[4] https://protobuf.dev/overview/

[5] https://www.privacyaffairs.com/vpn-protocols/