gRPC

Einführung

In diesem Projekt werde ich zuerst eine Kommunikation mithilfe von gRPC auf HelloWorld beschränken und danach eine ganze ElectionData senden und empfangen.

Projektbeschreibung

This exercise is intended to demonstrate the functionality and implementation of Remote Procedure Call (RPC) technology using the Open Source High Performance gRPC Framework
gRPC Frameworks (https://grpc.io).
It shows that this framework can be used to develop a middleware system for connecting several services developed with different programming languages.

Theorie

Die Videos im Elearning Kurs geben eine kleine Einführung, wie gRPC funktioniert und wie man so ein Programm realisiert

Arbeitsschritte

GKü

Als erstes habe ich mir ein neues Projekt erstellt. Danach habe ich das erste Proto File erstellt. In diesem File habe ich meine ersten Requirements festgelegt. Diese waren schon in einer externen Datei vorgegeben und beinhalteten eine default message und ein Starten der RPC Anwendung. Danach konnte ich mit dem Befehl

```
python -m grpc_tools.protoc -I. --python_out=. --grpc_python_
```

Die beiden helloworld.proto files erstellen lassen. In diesen war meine Struktur festgelegt. Danach musste ich nur noch zwei Klassen erstellen, den Client und

den Server. Im Server:

```
from concurrent import futures
import grpc
import helloworld_pb2
import helloworld_pb2_grpc


# Implement the HelloWorld service
class HelloWorldServicer(helloworld_pb2_grpc.HelloWorldServic
    def SayHello(self, request, context):
        message = f"Hello, {request.name}!"
        return helloworld_pb2.HelloResponse(message=message)


def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_worke
    helloworld_pb2_grpc.add_HelloWorldServicer_to_server(Hell
    server.add_insecure_port('[::]:50051')
    print("Server is running on port 50051...")
    server.start()
    server.wait_for_termination()


if __name__ == '__main__':
    serve()
```

Hier habe ich einen Service erstellt, der eine Hello World Message ausgibt. In der Methode serve() wird der Server auf dem Port 50051 gestartet.

Client Implementation

```
import grpc
import helloworld_pb2
import helloworld_pb2_grpc


def run():
```

```
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = helloworld_pb2_grpc.HelloWorldStub(channel)
        name = input("Enter your name: ")
        response = stub.SayHello(helloworld_pb2.HelloRequest(
        print(f"Server response: {response.message}")


if __name__ == '__main__':
    run()
```

In meiner Client Klasse, rufe ich die zwei von der Proto Klasse erstellten Dateien auf, indem ich in der einen einen Channel erstelle, durch den kommuniziert werden kann und in der anderen die Grundstruktur an die Server Klasse weitergebe.

GKv

Um jetzt das ganze auf GKv umzuändern, muss ich zuerst ein Datenformat von ElectionData importieren. Dafür habe ich eine XML Ausgabe der ersten Übung als XML Datei abgespeichert. Für die Darstellung im Proto File musste ich das Proto File in das Schema des XML Files umändern. Das sah dann wie folgt aus:

```
syntax = "proto3";

package electiondata;

// Nachricht für Kandidaten
message Candidate {
  int32 listnumber = 1;
  string name = 2;
  int32 votes = 3;
}

// Nachricht für Parteien
message Party {
  string partyID = 1;
  int32 amountVotes = 2;
  repeated Candidate preferedVotes = 3;
}
```

```
// Nachricht für Wahlregion
message ElectionData {
  int32 regionID = 1;
  string regionName = 2;
  string regionAddress = 3;
  int32 regionPostalCode = 4;
  string federalState = 5;
  string timestamp = 6;
  repeated Party countingData = 7;
}

// gRPC-Dienst für die Übertragung von ElectionData
service ElectionService {
  rpc SendElectionData (ElectionData) returns (ElectionRespon
}

// Antwortnachricht
message ElectionResponse {
  string confirmation=1;
}
```

Durch diese Struktur sind die einzelnen Klassen ineinander verschachtelt, so wie die anderen Klassen auch sind. Aus diesem Proto File werden auch wie in der Übung davor die beiden Dateien automatisch erstellt.

Server

Für die Implementierung vom Server musste ich nur die Daten auf die Variablennamen der Protofile umändern. Das hat dann so ausgeschaut:

```
import grpc
from concurrent import futures
import helloworld_pb2
import helloworld_pb2_grpc


# Implementierung des ElectionService
class ElectionServiceServicer(helloworld_pb2_grpc.ElectionSer
```

```
    def SendElectionData(self, request, context):
        # Log der empfangenen Daten
        print(f"Region: {request.regionName}, State: {request
        for party in request.countingData:
            print(f"Party: {party.partyID}, Votes: {party.amo
            for candidate in party.preferedVotes:
                print(f"  Candidate: {candidate.name}, Votes:

        # Bestätigung zurückgeben
        return helloworld_pb2.ElectionResponse(confirmation="


# Server starten
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_worke
    helloworld_pb2_grpc.add_ElectionServiceServicer_to_server
    server.add_insecure_port('[::]:50051')
    print("Server started on port 50051")
    server.start()
    server.wait_for_termination()


if __name__ == '__main__':
    serve()
```

Hier gebe ich die unterschiedlichen Daten auf der Konsole aus. Dadurch dass sie verschachtelt sind, muss ich mehrere for-Schleifen machen.

Client

Der Client funktioniert genauso wie der vorherige Client, nur dass statt einem String ein ElectionData Objekt übergeben wird. Diese wird lokal erstellt und dann weitergegeben.

Zusammenfassung

In diesem Projekt habe ich gelernt, wie ich eine gRPC Anwendung umsetze und diese dann auf meine individuellen Bedürfnisse anpassen kann.

Quellen

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

https://grpc.io/docs/protoc-installation/