gRPC

gRPC is a modern, open-source, general-purpose remote procedure call (RPC) framework developed by Google. It enables applications written in different languages and running on different platforms to communicate easily. gRPC uses Protocol Buffers (protobuf) as its default serialization protocol.

Key concepts:

- Services and methods: Services and their callable remote methods are defined in a `.proto` file.
- Client and server-side code generation: Interfaces and data types are generated automatically based on the protobuf definition.

Sample Example

Create a virtual environment:

```
  python -m virtualenv ./venv
```

Then activate it:

```
  ./venv/Scripts/activate
```

Install the required dependencies:

```
  python -m pip install grpcio
  python -m pip install grpcio-tools
```

Create the `./proto` directory and add the IDL file named `helloworld.proto`:

```
syntax = "proto3";

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

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

// The response message containing the greetings
message HelloReply {
  string message = 1;
```

```
}
```

Run the stub generator with the following command:

```
  python -m grpc_tools.protoc -I ./protos/ --grpc_python_out=. --
python_out=. .\protos\helloworld.proto
```

The files `helloworld_pb2.py` and `helloworld_pb2_grpc.py` will be generated in the root folder.

The client and server code is as follows:

`greeter_client.py`

```
import grpc
import helloworld_pb2
import helloworld_pb2_grpc

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

`greeter_server.py`

```
from concurrent import futures

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!" %
request.name)

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()
```

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

Start the server and the client.

Advanced Features

https://grpc.io/docs/guides/

- Authentication – Methods for authentication, including custom mechanisms.
- Benchmarking – Tools and methods for measuring performance.
- Cancellation – Cancelling RPC calls.
- Compression – Compressing data before transmission.
- Custom Backend Metrics – Collecting custom metrics on client and server side.
- Custom Load Balancing Policies – Implementing custom load balancing strategies.
- Custom Name Resolution – Customizing name resolution logic.
- Deadlines – Using timeouts to avoid waiting on unresponsive services.
- Debugging – Using `grpcdebug` for debugging.
- Error Handling – Understanding and handling error codes.
- Flow Control – Manual flow control of data streams.
- Graceful Shutdown – Shutting down servers without disrupting clients.
- Health Checking – Supporting health checks on client and server side.
- Interceptors – Middleware for logging, authentication, or metrics.
- Keepalive – HTTP/2-based connection keep-alive mechanisms.
- Metadata – Sending additional data via headers.
- OpenTelemetry Metrics – Observability and metric collection.
- Performance Best Practices – Language-specific performance tuning tips.
- Reflection – Querying service definitions at runtime.
- Request Hedging – Re-sending delayed requests in parallel.
- Retry – Fine-grained retry control.
- Service Config – Client behavior configuration via service config files.
- Status Codes – Detailed list of status codes and their meanings.
- Wait-for-Ready – Ensuring client waits until the server is ready.

Tasks

1. Try the **Deadlines** feature to learn how to handle timeouts properly.