RPC File transfer

Distributed System Group 4

5/3/2020

Contents

1   Introduction

Like many RPC systems, gRPC is based around the idea of defining a service,
specifying the methods that can be called remotely with their parameters and
return types. By default, gRPC uses protocol buffers as the Interface Definition
Language (IDL) for describing both the service interface and the structure of
the payload messages. It is possible to use other alternatives if desired.
Grpc was built with HTTP/2 protocol, its packages are not Json or XML but
Proto (google.protobuf)

2   Design

In gRPC, a client application can directly call a method on a server application
on a different machine as if it were a local object, making it easier for you to
create distributed applications and services. As in many RPC systems, gRPC
is based around the idea of defining a service, specifying the methods that can
be called remotely with their parameters and return types. On the server side,
the server implements this interface and runs a gRPC server to handle client
calls. On the client side, the client has a stub (referred to as just a client in
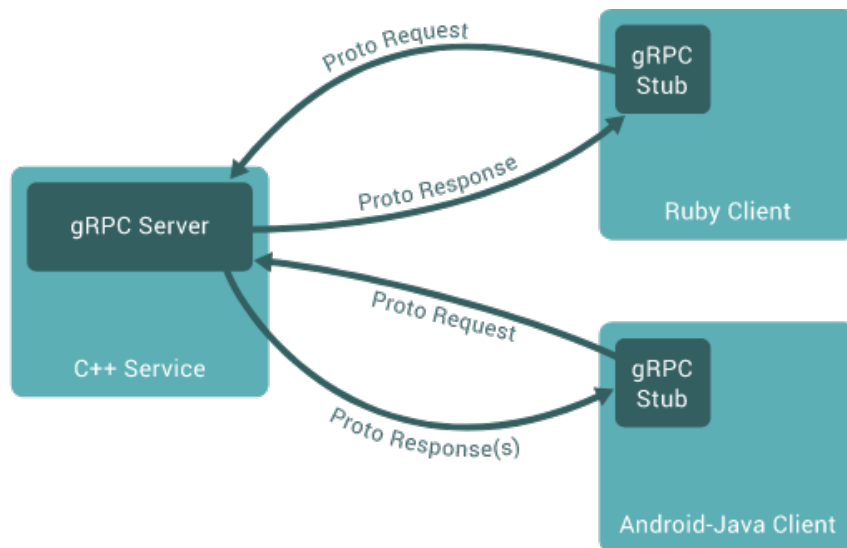some languages) that provides the same methods as the server.

Figure 1: gRPC

gRPC clients and servers can run and talk to each other in a variety of envi-
ronments - from servers inside Google to your own desktop - and can be written
in any of gRPC's supported languages. So, for example, you can easily create
a gRPC server in Java with clients in Go, Python, or Ruby. In addition, the

latest Google APIs will have gRPC versions of their interfaces, letting you easily build Google functionality into your applications.

```
1  syntax = "proto3";
2
3  package fileTransfer;
4
5  enum Status{
6      ON_CREATE = 0;
7      ON_UPLOAD = 1;
8      ON_FINISH = 2;
9  }
10
11 message uploadByte{
12     Status code = 1;
13     string data = 2;
14 }
15
16 message status{
17     Status code = 1;
18 }
19
20 service fileTransfer{
21     rpc uploadFile (uploadByte) returns (status) {}
22 }
```

Listing 1: file.proto

3   Code

3.1   Client

```
1      const PROTO_PATH = __dirname + '/../protos/file.proto'
2      const grpc = require('grpc')
3      const fs = require('fs')
4      const chunkingStreams = require('chunking-streams')
5      const SizeChunker = chunkingStreams.SizeChunker
6      const protoLoader = require('@grpc/proto-loader')
7      const packageDefinition = protoLoader.loadSync(PROTO_PATH, {
           keepCase: true, longs: String, enums: String, defaults:
           true, oneofs: true })
8      const fileTransfer_proto = grpc.loadPackageDefinition(
           packageDefinition).fileTransfer
9
10     const client = new fileTransfer_proto.fileTransfer('localhost
           :8000', grpc.credentials.createInsecure())
11
12     const filePath = __dirname + '/send/random.txt'
13     const defaultSize = 10 * 1024
14     const input = fs.createReadStream(filePath)
15     const uploadFileProcess = async() => {
16         client.uploadFile({ code: 0, data: "random.txt" }, (err,
               response) => {
17             console.log(response.code)
18             chunker = new SizeChunker({
```

```
19                    chunkSize: defaultSize ,
20                    flushTail: true
21            })
22            chunker.on('data', (chunk) => {
23                client.uploadFile({ code: 1, data: chunk.data.
                        toString() }, (err, response) => {
24                    console.log(response.code)
25                })
26            })
27            input.pipe(chunker)
28            input.on('end', () => {
29                client.uploadFile({ code: 2, data: "" }, (err,
                        response) => {
30                    console.log(response.code)
31                    input.close()
32                })
33            })
34        })
35    }
36    uploadFileProcess ()
```

3.2 Server

```
1    const PROTO_PATH = __dirname + '/../protos/file.proto'
2
3    const grpc = require('grpc')
4    const fs = require('fs')
5    const protoLoader = require('@grpc/proto-loader')
6    const packageDefinition = protoLoader.loadSync(PROTO_PATH, {
         keepCase: true, longs: String, enums: String, defaults:
         true, oneofs: true })
7    const fileTransfer_proto = grpc.loadPackageDefinition(
         packageDefinition).fileTransfer
8
9    var output;
10
11   const uploadFile = (call, callback) => {
12       switch (call.request.code) {
13           case "ON_CREATE":
14               output = fs.createWriteStream(`${call.request.data
                    }`)
15               callback(null, { code: call.request.code })
16               break
17           case "ON_UPLOAD":
18               try {
19                   output.write(Buffer.from(call.request.data))
20                   callback(null, { code: call.request.code })
21               } catch (err) {
22                   console.log(err);
23               }
24               break
25           case "ON_FINISH":
26               callback(null, { code: call.request.code })
27               break
28       }
29   }
30
31   const server = new grpc.Server();
32   server.addService(fileTransfer_proto.fileTransfer.service, {
         uploadFile: uploadFile })
33   server.bind('localhost:8000', grpc.ServerCredentials.
         createInsecure())
34   server.start()
```

4 Who does what?

Bui Quang Huy : Research on gRPC and design nodejs app for RPC
Nguyen Viet Dung and Nguyen Quang Trung : Design and write the report.
Do Minh Hoang: Research about the protocol.