Ira A. Fulton Schools of Engineering
Arizona State University
Engineering | School of Computing, Informatics, and Decision Systems Engineering

CSE 531: Distributed and Multiprocessor Operating Systems

gRPC Written Report

Introduction

gRPC (general/Google Remote Procedure Call) API (application programming interface) is an RPC-based protocol published by Google in 2015 [1,2]. gRPC is cross-platform, language and platform independent, general-purpose infrastructure [2]. gRPC was developed to be useable over the web and it run over HTTP/2.0. gRPC is fast speed processing compared to REST (REpresentational State Transfer) API, because gRPC can process a remote procedure call with a single connection through the multiplexed stream function [1]. Another advantage of gRPC, is that API developers do not need HTTP commands in their codes. The API developers only need to select the procedure to call and necessary parameters [3].

gRPC can automatically generate idiomatic client and server stubs for service in a variety of languages and platforms using Protocol Buffers that is a flexible, efficient, automated mechanism for binary serialization of structured data [2]. It is critical that users define how they want their data to be structured once in Protocol Buffers language (proto3) and the signature of the methods that will be called remotely. The defined data structure is stored in .proto files in the form of a small logical record of information. This information consist of series of name-value pairs known as protocol buffer message. After the user define their messages, they need to run the protocol buffer compiler for their choice of application's language on their .proto file to generate data access classes and provides the stub implementations. gRPC has higher performance because Protocol buffers are 3 to 10 times smaller and 20 to 100 times faster than XML for serializing structured data [2].

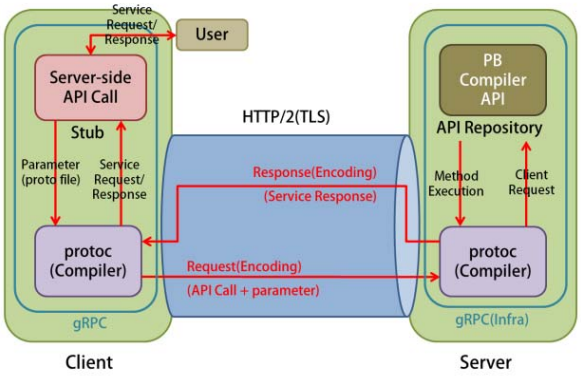

Figure 1: gRPC Client-Server Operation Process [adopted from Du, Sang, Jong Lee, and Keecheon Kim, 2018].

Problem Statement

Modern banks have many branches in different cities in a country or all over the world. Therefore there could be multiple transactions on the same account in different branches/locations. This means that bank database server performance should be kept optimal [4]. Distributed banking systems provide an alternative to centralized bank-based systems; and in 2017, distributed banking was projected to reduce the world's unbanked population by more than a third [5].

In this project, the intention is to build a distributed banking system in which specific customer communicate with specific branch and carries out transactions (query on balance, withdrawal and deposit). This specific branch to customer communication is supposed to be unique ID based. As each customer update a specific replica branch, there should be a system for that replica to update other replica for efficient distributed banking. At the end of the day, it is expected that the global balance is appropriately updated. Also the transactions are expected to occur in such a way as that there is no concurrent update of the balance. The branches are the servers and the customers are the clients in this bidirectional distributed network communication.

Goal

The goal of this project is to enable the implementer/programmer/student utilize gRPC API in implementing a distributed banking system that fulfills the above problem statement. As a result of the implementation, the specific goals of the project is to ensure that the student is able to:
a. understand the basics concept of gRPC API
b. define how they want their data to be structured in Protocol Buffers language (proto3) and store it in .proto file.
c. utilize Python gRPC to automatically generate idiomatic client and server stubs for service using protocol buffer compiler, for binary serialization of structured data.
d. complete the provided server (Branch.py) and client (Customer.py) scripts.
e. execute the implementation to achieve the problem statement's requirements.

Setup

This project was executed in Python using gRPC framework which is based on Protocol Buffers language (proto3) and runs over HTTP/2. All the implementation and testing was carried out in Ubuntu 20.04.2 LTS using Visual Studio Code.

Implementation Processes

The project was implemented as following in a single folder which I named HannahAjoge_gRPC_Project_Code:

A. **Definition of service in a .proto file:** Here I implemented my .proto file, which I named as example.proto. As expected this file consisted of a bunch of definitions to describe the service that the server provides. First, I described the syntax to use specific protocol buffer version 3 (proto3). proto3 is recommended over the default proto2, because proto3 allows the use of the full range of gRPC-supported languages, as well as avoids compatibility issues with proto2/3 clients-servers communication[6]. I did not specify a package, since I have only a single .proto file in the folder. Secondly, I defined a service named RPC (taking hint from the class in the given Branch.py file) which defines the interaction of the servers with the clients. To keep it simple I had a single PPC called MsgDelivery (taking hint from the appropriate function in the given Branch.py file). Finally, I implemented the message schema's data types to be structured as strings. The idea here, is that the client sends a string consisting of a list of two components - customer [corresponding branch] unique ID and a dictionary of transaction requests(s). The server execute the transaction and replies with a string consisting of the unique ID, transaction processing ID and the transaction result of success/fail.

B. **Automatic generation of client and server stubs for service using protocol buffer compiler:** The result was two files - example_pb2.py (containing generated protocol buffer code) and example_pb2_grpc.py (containing client and server classes corresponding to protobuf-defined services). I executed the following command from within the folder in the terminal: *python -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=. example.proto*

C. ***Input test file:*** The provided test input was copied and pasted into a file and saved as "input.json". This json file contains a list of Customers and Branch processes.

D. **Implementing the servers:** In addition to updating given Branch.py file, I also implemented functions in a file (example.py). Branch.py calls a function (branchupdate) in example.py in order to process requests from the clients. The function accepts the request from the client, split the "events" part into dictionaries and pass each dictionary to another function (eventdict) in the example.py file for processing. The eventdict function takes a subset of the client's request (provided by the branchupdate function) and update the balance in a "balance.txt" file, if a withdrawal or deposit is made. This process essentially does the work of Branch to Branch interface, by updating the balance through a file lock system to avoid concurrent update. The balance.txt file serves as the "database" of the balance which is accessible and updateable by each Branch. Additionally, the eventdict returns a list to the branchupdate which is processed into the final message that is returned to the client. This message includes the unique ID, transaction processing ID and the transaction result of success/fail.

a) ***Branch to branch communication:*** In the live event discussion the instructor suggested possible use of lock to ensure nonconcurrent update of replica. In the live event, I also ask if we can have the balance stored in a "database" which is accessible and updateable by each branch and the instructor was ok with it. Also with the fact that we allowed to not necessary stick to the format (classes/functions) in the Branch.py file, I removed the provided __init__ function in Branch.py. Instead I utilized the balance.txt "database" with

Lock from threading library to implement the branch to branch "interface" that ensures serialized update of the global balance.

b) ***Handling Request from Customers:*** In the Branch.py file, I updated the MsgDelivery function to handle requests from customers by calling the branchupdate function in example.py as explained above. As required, from within the same function MsgDelivery function, storage of the processID of the branches was accomplished by each branch saving the process IDs to a *"Branch[IDnumber]_IDs.txt"* file. Where IDnumber is replaced with the branch ID.

c) ***Running and listening for request:*** I ensured that on running the Branch.py file that first, it search for the balance.txt "database" and if it does not exist, the database is created by reading through the "input.json" file to obtain the initial bank balance. Secondly, the server listen on a specific port and wait for request from a specific client so that it can send a response back to the client. The specific port is calculated to be specific for a specific customer by ensuring that the port is equal to 50047 plus the unique branch ID. To run the servers, I execute the command "*python Branch.py*" in the terminal. This prompt for user input (the unique ID) which is used to indicate the port to which to listen from. Since in the test case, there are three unique IDs for the three customer, I executed the command above three times and provided the unique IDs (1, 2 and 3 respectively) each time. This resulted in three servers listening on ports 50048, 50049 and 50050 respectively. On receiving a request from the appropriate client, the server processes the request and respond as expected.

E. **Implementing the client:** I updated the given Customer.py by deleting the createStub function and updating the executeEvents function for both creating the Customer stub and sending out events to the branches. Summarily, the following is achieved in the executeEvents function:

a) gRPC channel that is specific for branch-customer transactions is opened. Just as in the server side, the calculation involves adding 50047 to the unique ID to ensure the appropriate server receives the request. With the channel, a stub is created.

b) To send out requests to the appropriate branch, a valid request message is created and then a call is made.

Finally, I ensured that on running the Customer.py file, the "input.json" file is read and process for sending request to the appropriate server. Then response from server is processed to the expected format and stored in a file ("output.json"). In addition to the requirement I added the two following operations:

a) That after each request has been responded to and written to the output file. The client should shut down the corresponding server.

b) Also the client after shutting down all the servers, it should rename the "balance.txt file to "old_balance.txt" file. If this is not implemented and the whole imlpementation is repeated in the folder, the balance.txt wile will be updated instead of recreated and that will lead to a balance of $600 after the first repeat run, $700 after second repeat run, etc.

Results

On executing the implementation process explained in the preceding section, an output file (output.json) with the expected output format is generated. In addition, files containing the processing IDs are generated (in the test case Branch1_IDs.txt, Branch3_IDs.txt and Branch2_IDs.txt). The final balance is also stored in a file (old_balance.txt). While the justification for each step of the implementation has been explained along with each step in the preceding section, I wish to succinctly explain the overall results and the justification as following:

a) Using the implemented Branch.py, Customer.py, example.proto and example.py files as well as the appropriate input file (input,json); the a distributed banking system was built. This system ensured that specific customer communicate with specific branch and carries out transactions (query on balance, withdrawal and deposit) appropriately based on unique IDs.

b) The introduction of the "balance.txt" database system and the use of Locks, ensured that as each customer's request is executed in specific branch the global balance is updated in a nonconcurrent manner. At the end of the process the expected global balance is achieved and graciously retire as "old_balance.txt".

c) The implementation ensured that the branches are the servers and the customers are the clients in a bidirectional distributed network communication. Finally, to mimic the real life scenario of exiting connection with a bank branch once a clients has successfully completed their transactions, each client stub terminates the server after writing the expected result to the output.json file.

References

1. Du, Sang, Jong Lee, and Keecheon Kim. "Proposal of GRPC as a New Northbound API for Application Layer Communication Efficiency in SDN." *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*. ACM, 2018. 1–6. Web.

2. Kiraly, Sandor, and Szilveszter Szekely. "Analysing RPC and Testing the Performance of Solutions." Informatica (Ljubljana) 42.4 (2018): 555–561. Web.

3. Albert Fang. REST vs gRPC: Understanding Two Very Different API Styles. Accessed March 30th 2021, Last updated February 22, 2021. https://rapidapi.com/blog/rest-vs-grpc-understanding-two-very-different-api-styles/

4. Nugroho, Agus Cahyo, and Mychael Maoeretz Engel. "Hybrid Distributed Application in Banking Transaction Using Remote Method Invocation." Telkomnika 17.5 (2019): 2208–. Web.

5. Fan, Wenjun et al. "Blockchain-Based Distributed Banking for Permissioned and Accountable Financial Transaction Processing." 2020 29th International Conference on Computer Communications and Networks (ICCCN). IEEE, 2020. 1–9. Web.

6. The gRPC Documentaion. https://grpc.io/docs/what-is-grpc/introduction/