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

Problem Statement

The problem statement asks us to create a distributed banking system that enables multiple customers to make withdrawals or deposits into a shared bank account. Each customer engages with a particular branch, and each branch maintains a replica of the account balance which is consistent with other branches.

Goal

The goal is to create a functional distributed banking system in which inter-process communication is established via gRPC. This is accomplished by the following objectives:
- Define a service in a .proto file.
- Generate server and client code / stubs using the protocol buffer compiler present in the grpcio-tools package in python
- Use the Python gRPC API to create client and server components.
- Implement communication between customer and branch processes and branch and branch processes using gRPC.
- Implement interfaces (Query, Deposit, and Withdraw) for customer-branch and branch-branch interactions.
- Ensure that the balance is consistent across all branches by propagating in case of deposit and withdraw interactions.
- Execute customer events sequentially, waiting for propagation to other branches before the next customer event.
- Provide successful response records to customers and write the results in an output file.

Setup

Installations:

- Install Python 3.9 or above (I used Python 3.9).

- Install pip version 9.0.1 or higher.

- (OPTIONAL) If necessary, upgrade your version of pip:

```
 python -m pip install --upgrade pip
```

If you cannot upgrade pip due to a system-owned installation, you can run the example in a virtualenv:

```
$ python -m pip install virtualenv
```

$ virtualenv venv
$ source venv/bin/activate
$ python -m pip install --upgrade pip

- Install gRPC:
 $ python -m pip install grpcio

- Install gRPC tools:
 python -m pip install grpcio-tools

Running the project:

- Unzip from canvas

- Navigate to the folder where main.py is present.

- This step is OPTIONAL:
 If you want to regenerate the code generated by the gRPC library run:
python -m grpc_tools.protoc -I./protos --python_out=. --pyi_out=. --grpc_python_out=. ./protos/banking_system.proto
This will regenerate the files: banking_system_pb2.py, banking_system_pb2.pyi, banking_system_pb2_grpc.py

- One main requirement is that if your input file has branches from 1 to n then there should be n empty ports starting from 50001 to run the gRPC branch servers. E.g. If the branch id is 9 then its server runs on port 50009. If the ports are not available on your machine then you can change the value of BASE_PORT in constants.py to a value that suits your machine.

- Run:
 python main.py --input_file "./inputs/input_1.json"
where –input_file is the path of your input file.

- The output will be present in ./outputs/output.json (The path is relative to the folder containing main.py)

Implementation Processes

The implementation processes involve:

- Defining a service in a .proto file for gRPC along with the message format for request and response.
- Generating server and client code / stubs from the .proto file using grpcio-tools package in python
- Creating the branch.py file containing the following:
    - Query functionality
    - Deposit functionality
    - Withdraw functionality
    - propagate_deposit functionality using gRPC
    - propagate_withdraw functionality using gRPC
    - For the propagate_deposit and propagate_withdraw functionality it is important to call other branch stubs with the propagate flag set to False otherwise the program will be stuck in an infinite loop.
    - message_delivery functionality which reads the customer request and performs the action accordingly.
- Creating the customer.py file containing the following:
    - execute_events functionality: This reads the events stored in the customers event list and calls the stub for the branch associated with the customer using gRPC. It then stores the response in the customer's recd_msgs.
    - get_customer_results(): This method returns the results received by the customer in the format mentioned in the output in the project document.
- Creating the main.py file which does the following:
    - create the empty output file
    - read data from the input file
    - start branch processes and initialize their stubs for communicating with other branches and other required data like id, balance and ids of other branches
    - start cutomer processes and initialize their stubs for communicating with their corresponding branch and also initialize other required details like their id and events they have to process. The customers are executed in a sequential manner in the sense that one branch process is started only after the previous once is finished. The customer processes also append to the output file once they have finished all their transactions
    - propagate_deposit functionality
    - propagate_withdraw functionality
    - Once all the customer processes are finished stop the branch server so that the program exits gracefully.
- creating the constant.py file which contains all constants used in the project.
- creating the logging_util.py file for well formatted customized logging for different processes.

- All interprocess communication has been done using gRPC and a sleep of 3 seconds has been placed before every query operation so that all deposit / withdraw propagates that were fired before the query operation are finished when the balance is being read for a customer.

Results

- For deposits and withdraws, the balance of self is changed and the request is propagated to other branches.
- For query operation, there a 3 second pause before we get the result of the operation result on our console.
- The results for a small sample request are as follows:
  Input:

```
[
 {
   "id": 1,
   "type": "customer",
   "events": [
     {
       "id": 1,
       "interface": "query"
     }
   ]
 },
 {
   "id": 2,
   "type": "customer",
   "events": [
     {
       "id": 2,
       "interface": "deposit",
       "money": 170
     },
     {
       "id": 3,
       "interface": "query"
     }
   ]
 },
 {
   "id": 3,
   "type": "customer",
   "events": [
     {
       "id": 4,
       "interface": "withdraw",
        "money": 70
      },
      {
        "id": 5,
        "interface": "query"
      }
    ]
  },
  {
    "id": 1,
    "type": "branch",
    "balance": 400
  },
  {
    "id": 2,
    "type": "branch",
    "balance": 400
  },
  {
    "id": 3,
    "type": "branch",
    "balance": 400
  }
]
```

Output:

```
[
  {
    "id": 1,
    "recv": [
      {
        "interface": "query",
        "result": "success",
        "balance": 400
      }
    ]
  },
  {
    "id": 2,
    "recv": [
      {
        "interface": "deposit",
        "result": "success"
      },
      {
        "interface": "query",
        "result": "success",
        "balance": 570
      }
    ]
  },
  {
    "id": 3,
    "recv": [
      {
        "interface": "withdraw",
        "result": "success"
      },
      {
        "interface": "query",
        "result": "success",
        "balance": 500
      }
    ]
  }
]
```

- For the above case the initial balance is 400 in each branch. We can see that customer 2 has deposited 170 after which the query return 570 as balance. Then there is a withdraw of 70 by customer 3 which reflects as 500 after we get the response for query operation. We can see that all the operations are successful since they have “success” in their result field.
- For the main test case, the output is contained in ./output/output_2.json. In this the initial balance is 400 and there are 50 users. All 50 users deposit 10 in a sequential manner so the balance increases from 400 to 900 in intervals of 10 after each operation. Then, all the users withdraw 10 so the balance deceases by 10 after each operation. Finally, the balance returns to 400 after all the deposit and withdraw operations are completed.
- The above 2 examples show us that the deposits and withdraws are being propagated properly across different branches and that our distributed banking system is working via inter process communication using gRPC. If our system was not working then final_balance would not be equal to (initial_balance + (total_deposits) - (total_withdraws)).