Purpose

The goal of this project is to build a distributed banking system that allows multiple customers to withdraw or deposit money from multiple branches in the bank. We assume that all the customers share the same bank account and each customer accesses only one specific branch. In this project, we also assume that there are no concurrent updates on the bank account. Each branch maintains a replica of the money that needs to be consistent with the replicas in other branches. The customer communicates with only a specific branch that has the same unique ID as the customer. Although each customer independently updates a specific replica, the replicas stored in each branch need to reflect all the updates made by the customer.

Objectives

Students will be able to:

- Define a service in a .proto file.
- Generate server and client code using the protocol buffer compiler.
- Use the Python gRPC API to write a simple client and server for your service.
- Build a distributed system that meets specific criteria.
- Determine the problem statement.
- Identify the goal of the problem statement.
- List relevant technologies for the setup and their versions.
- Explain the implementation processes.
- Explain implementation results.

Technology Requirements

- Access to Github
- Python
- gRPC
- The basic templates for Customer and Branch processes are provided as “Customer.py” and “Branch.py” can be found within the project description in the course.
- Personal computer with 8 GB RAM or higher.

Helpful Resources

- gRPC Quick Start Guide(https://grpc.io/docs/languages/python/quickstart/)

Directions

Part 1: Written Report
Your written report must be a single PDF with the correct naming convention: Your Name_gRPCF_Written Report.
Using the provided Student Template_Your Name_gRPCF_Written Report, compose a report addressing the questions:
1. What is the problem statement?
2. What is the goal of the problem statement?
3. What are the relevant technologies for the setup and their versions?
4. What are the implementation processes?
5. What are the results and their justifications?
*Students may add subheadings on the template to purposefully call attention to specific, organized details

Part 2: Project Code
The project code may be submitted as a .zip file of your source code or as a text file with a source repository link to your source code.

Major Tasks
1. Implement the customer and branch processes using Python processes
2. Implement the communication between the processes using gRPC
3. Implement the interfaces (Query, Deposit, and Withdraw) that are used between the customer processes and the branch processes
4. Implement the interfaces (Propogate_Deposit and Propogate_Withdraw) that are used between branch processes

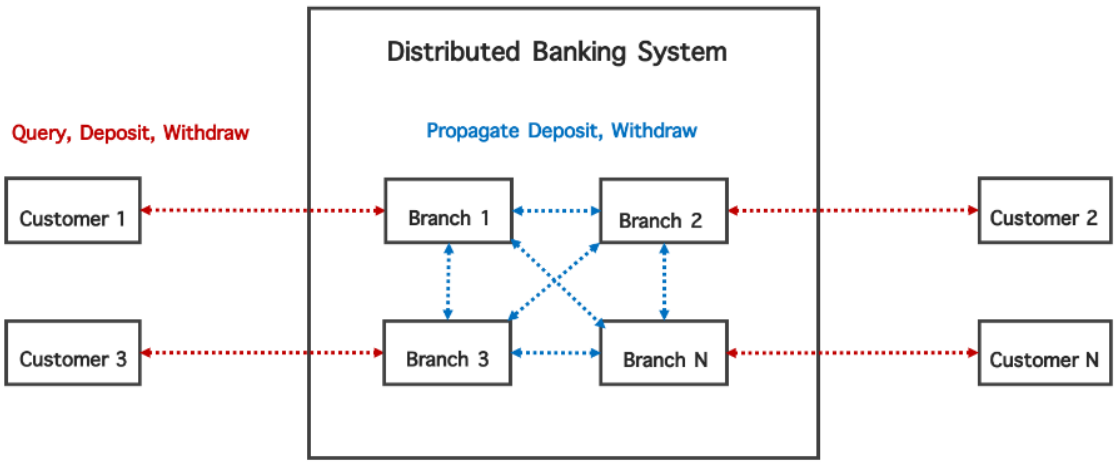

Diagram A: Topology of the Banking System

1. Description

In the first part of this project, you will implement an interface using gRPC so that the processes can communicate with each other.

1.1. gRPC

The communication is implemented with the RPC that we have learned in our previous lecture (Unit 3 Module 2 Lecture 1). We will be using the simplest type of RPC in gRPC, where the customer sends a single request and gets back a single response.

1. The client process calls the method on the stub/client object, the server process is notified that the RPC has been invoked with the client's metadata for this call, the method name, and the specified deadline if applicable.
2. The server can then either send back its own initial metadata (which must be sent before any response) straight away, or wait for the client's request message - which happens first is application-specific.
3. Once the server has the client's request message, it does whatever work is necessary to create and populate its response. The response is then returned (if successful) to the client together with status details (status code and optional status message) and optional trailing metadata.
4. If the status is OK, the client then gets the response, which completes the call on the client side.

gRPC Quick Start Guide(https://grpc.io/docs/languages/python/quickstart/) explains the basic Python programmer's introduction to working with gRPC using a simple program. We strongly encourage you to refer to this resource.

1.2. Customer and Branch Processes

The basic templates for Customer and Branch processes are provided as "Customer.py" and "Branch.py". Students are expected to complete the function Customer.createStub and Customer.executeEvents in the Customer.py and use it to generate Customer processes. Customers use the Customer.createStub to communicate with the Branch process with the ID identical to the Customer. Customer.executeEvents processes the events from the list of events stored in the Customer class and submits the request to the Branch process.

Additionally, students are expected to complete the Branch.MsgDelivery of the Branch.py and use it to generate Branch processes. Branch.MsgDelivery processes the requests received from other processes and returns results to the requested process. Students should complete the described functions and the Customer and Branch to complete this assignment

1.3. Branch to Customer Interface

Branch.MsgDelivery receives the request from the Customer process. The Branch process should decrease or increase its amount of money, Branch.balance, according to the Customer's request. The customer stubs of the Branches are stored as a list in the Branch.stubList of the Branch class.

➔ Branch.Query interface reads the value from the Branch.balance and returns the result back to the Customer process.
➔ Branch.Withdraw interface decreases the Branch.balance with the amount specified in the Client's request, propagates the request to its fellow branches, and returns success / fail to the Customer process.
➔ Branch.Deposit interface increases the Branch.balance with the amount specified in the Client's request, propagates the request to its fellow branches, and returns success / fail to the Customer process.

1.4. Branch to Branch Interface
Branch.MsgDelivery receives the request from its fellow branches that propagate the updates. The Branch process should decrease or increase its replica, Branch.balance, according to the branches' request

➔ Branch.Propogate_Withdraw interface decreases the Branch.balance with the amount specified in the Branch's request and returns success / fail to the Branch process.
➔ Branch.Propogate_Deposit interface increases the Branch.balance with the amount specified in the Branch's request and returns success / fail to the Branch process.

1.5. Input and Output
The input file contains a list of Customers and Branch processes. For each customer process, it specifies the list that it will execute. All the events will be executed in the order specified by their unique identifiers. For each branch process, it specifies its initial account balance, which should be the same across all branches.

You need to implement a main program to launch and execute the Customer and Branch processes according to the input file. In this project, we assume that there are no concurrent operations to the bank account. Therefore, the customers in fact execute sequentially one by one, and you can use a sleep command before launching a new Customer process to wait for all the operations performed by the previous customer on one branch to be propagated to the other branches.

```
[ // Start of the Array of Branch and Customer processes
  { // Customer process #1
    "id" : {a unique identifier of a customer},
    "type" : "customer",
    "events" :  [{"interface":{deposit | withdraw}, "money": {an integer value}, "id": {unique
identifier of an event} }, {"interface":{query}, "id": {unique identifier of an event} } ]
  }
  { … } // Customer process #2
  { … } // Customer process #3
  { … } // Customer process #N
  { // Branch process #1
    "id" : {a unique identifier of a branch},
    "type" : "branch"
    "balance" : {the initial amount of money stored in the branch}
  }
  { … } // Branch process #2
  { … } // Branch process #3
  { … } // Branch process #N
] // End of the Array of Branch and Customer processes
```

The output contains the list of successful responses that the Customer processes have received from the Branch processes.

```
[ // Start of the Array of Customer processes
  { // Customer process #1
    “Id” : { a unique identifier of a customer  }
    “recv” {a list of successful returns of the events from the Branch process}
  }
 { … } // Customer process #2
 { … } // Customer process #3
 { … } // Customer process #N
] // End of the Array of Customer processes
```

Below is an example input file and the expected output file. Note that the final query event sleeps for three seconds to guarantee all the updates are propagated among the Branch processes.

Example of the input file

```
[
 {
  "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
 }
]
```

Expected output file:
Expected output file: {'id': 1, 'recv': [{'interface': 'query', 'result': 'balance': 400}]} {'id': 2, 'recv': [{'interface': 'deposit', 'result': 'success'}, {'interface': 'query', 'result': 'balance': 570}]} {'id': 3, 'recv': [{'interface': 'withdraw', 'result': 'success'}, {'interface': 'query', 'result': 'balance': 500}]}

All the customer and branch processes should be terminated after all the events specified in the input file are executed.