CSE 531
Project 1
14th October, 2023

Name: *Ipsit Sahoo*
1224872415

Problem Statement

The task outlined in the Project description entails the development of a simplistic gRPC-based distributed banking system. The architecture is composed of N branches and N customers each identified by an exclusive integer in the range of **[1, N]**.

Crucially, a customer, distinguished by a specific identifier, exclusively communicates with the branch sharing the same identifier. For example, a customer bearing the ID 3 will solely interact with the branch corresponding to ID 3. It's important to emphasize that only one customer is concurrently active in submitting requests for deposit, withdrawal, or balance queries.

Unlike a normal bank, the bank serves as a communal bank account as the balance is shared across the system. All the banks are also supposed to synchronize the balance among themselves. For this we need to have a branch-branch communication to allow for propagation of customer initiated transactions of deposit and withdrawal.

The entire schema for communication needs to be defined in .proto template file that is used for RPC communications for customer-branch communications using the gRPC interface.

Goal

The goal/purpose of this distributed system is to build a distributed banking system for multiple customers to deposit and withdraw money across branches. Customers share one bank account, and each is linked to a specific branch by a unique ID. We assume no concurrent updates, and each branch maintains consistent replicas of the account balance reflecting customer transactions. The primary goal is to understand **Concurrency**.

Setup & Run Instructions

Setup

1. Install git for Windows with version (git version **2.40.1.windows.1**)
2. Install python version : **3.9.13**. Ensure python is accessible via Powershell
3. Install pip (Version **23.2.1**) py -m ensurepip –upgrade Source
4. Create a new virtual environment for pip and install the following packages
   (a) grpcio==1.59.0
   (b) grpcio-tools==1.33.2

(c) protobuf==3.14.0

5. Activate the newly created virtual environment
6. Clone the repository: CSE 531 Project

Run

1. Run the command

```
cd CSE-531-Project/Project 1
```

2. Run the following the command the build the proto files:

```
python3 -m grpc_tools.protoc -I . .\BankService.proto --
   python_out=. --grpc_python_out=.
```

3. Run the main.py file in the folder. By default the input file taken is input.json and outfile taken is output.json.

```
python main.py -o output.json -i input.json
```

Result

The results will be printed on the console output, after you run the third command in the run section.

Implementation Processes

1. **Creating the Proto[2] file**

   I began by defining and populating two essential message structures: Request and Response. In this initial step, I ensured that the Request message could hold the 'amount' field, while the Response message was designed to convey a boolean indicating the success or failure of the operation, alongside detailed balance information. Subsequently, I established a service called 'Branch' and incorporated critical methods and interfaces such as 'Query,' 'Withdraw,' 'Deposit,' 'Propagate_Withdraw,' and 'Propagate_Deposit.' These interfaces comprehensively cover the necessary functionalities for Project 1. Remarkably, all these interfaces operate with the same Request message as input and return an identical Result message. The request message for deposit and withdraw contains the 'money' attribute which used to update the amount. The 'Query' function however does not need the money attribute and return the response containing the account balance. This design simplifies the message handling and interfaces within the banking system[1].

2. **Implementing the main.py**

(a) **Data Parsing**: Initially, we parse the input JSON file, extracting and segregating customer events and branch events. This separation is essential as the nature of these events and their handling differs significantly.

(b) **Categorizing Branch Events**: Branch events are inherently unique, with only one occurrence per branch. Therefore, we categorize and handle these events separately.

(c) **Grouping Customers**: Given that each customer interacts exclusively with the branch corresponding to their ID, we group customers accordingly. This grouping ensures that customers and branches are aligned correctly during event processing.

(d) **Customer and Branch Processes**: We create distinct processes for customers and branches. The branch processes are exceptional because they act as servers, continuously running to serve customer requests. In contrast, customer processes are created to execute events sequentially.

(e) **Sequential Customer Execution**: To maintain order and ensure proper event execution, customer processes run sequentially. This ensures that the interactions between customers and branches are accurately preserved.

(f) **Worker Process Joining**: Once all customer and branch processes have completed their tasks, we employ process joining to guarantee that they are synchronized and executed in the intended sequence. This step is crucial for maintaining the order and integrity of the banking system's operations.

In summary, the process involves parsing, categorizing, and grouping events, creating distinct customer and branch processes, and ensuring sequential execution and synchronization through process joining. This approach ensures the accurate handling of customer interactions and branch operations in the distributed banking system.

3. **Branch class**

In the branch implementation, we have specific functionalities: QUERY, WITHDRAW, DEPOSIT, PROPAGATE_DEPOSIT, and PROPAGATE_WITHDRAW. Additionally, there is a MsgDelivery function responsible for receiving customer requests. Here's a breakdown of how these components work:

(a) **Query, Withdraw, Deposit**: These methods handle customer interactions with the branch. When a customer issues a query, withdrawal, or deposit request, the respective event is executed on the branch. The branch updates its balance as needed, and the result is returned to the customer.

(b) **MsgDelivery**: This function acts as a receiver for customer requests. It listens for incoming requests from customers and forwards them to the appropriate methods for processing.

(c) **Propagate_Deposit and Propagate_Withdraw**: These methods are responsible for updating other branches after a deposit or withdrawal event occurs. When a deposit or withdrawal operation is completed, the updated amount is disseminated to all other branches. This is achieved by creating a communication stub (likely a gRPC client) for each branch other than the primary branch. These stubs are used to send the updated information to all branches, ensuring that all branches remain synchronized.

(d) **Branch Communication**: Each branch maintains a list of known branches and their IP addresses. This information is vital for inter-branch communication. It simplifies the process of sending updates to other branches and ensures that all branches are kept informed about changes in the account balance.

In summary, the branch implementation consists of customer interaction methods (QUERY, WITHDRAW, DEPOSIT), a message reception function (MsgDelivery) to handle customer requests, and separate methods (PROPAGATE_DEPOSIT and PROPAGATE_WITHDRAW) for updating other branches with the latest account balance information. Effective communication between branches is facilitated by maintaining a list of known branches and their IP addresses.

4. **Customer Class**

In the customer class, operations like QUERY, WITHDRAW, and DEPOSIT are invoked for interactions with the branch in each Customer process. Additionally, there is an executeEvents function, which iterates through a list of requests and sends each request to the branch for a specific event. Furthermore, it's mentioned that the processed events and the final amounts are stored in a JSON file to be reflected in the final output. Here's a summary:

(a) **Customer-Branch Interactions**: In each Customer process, operations like QUERY, WITHDRAW, and DEPOSIT are executed, enabling customers to interact with the branch for various financial transactions.

(b) **executeEvents Function**: This function plays a critical role in handling customer events. It processes a list of customer requests, sending each request to the branch for the specified event (e.g., QUERY, WITHDRAW, DEPOSIT). This function is likely used to automate the handling of multiple customer events.

(c) **Data Dumping into JSON**: After each customer event is processed and the final amount is determined, the information, including the events and final balances, is recorded in a JSON file. This process ensures that the results of customer interactions are captured and can be reflected in the final output or for further analysis.

The combination of these components allows the customer class to effectively engage with the branch, automate event processing, and maintain a record of the interactions, enhancing the functionality and traceability of the distributed banking system.

RESULTS

On executing the given large input test case having 50 branches and 50 customers, we observe that as the events execute the branch balance increase from 400 to 900 and then decrease back to 400. I was also able to see the balance in each branch after each transaction remain the same denoting the establishment of concurrency as was the goal of the distributed systems application.

References

[1] Dejanovic, V. (2018). *gRPC Bank Example*. GitHub Repository. https://github.com/vladimir-dejanovic/grpc-bank-example/blob/master/src/main/proto/bank.proto

[2] gRPC Authors (2023). *gRPC Quick start*. Documentation. https://grpc.io/docs/languages/python/quickstart/