Problem Statement

In this project, the primary objective is to integrate a client-centric consistency model, specifically the read-your-writes model, onto Project 3's replicated data system within the banking domain. To achieve this, key responsibilities include developing functions to meticulously monitor read and write events attributed to the same customer across diverse branch processes.

   Major Tasks:

1. Read and verify your writes[3]:

    (a) Implement read-your-writes logic across multiple branches processes by the same customer.

    (b) Implement read your writes across multiple branches.

2. Output customer events

3. The message response from request are to be stored in a JSON format

Goal

The goal of this project is to understand **Client-Centric Consistency Model**. Customer has a single bank account and can access multiple branches instead of just 1, This will be lead of improved synchronization and consistent replication across branches in the system.

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 and activate 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. Unzip the zip file named 'Ipsit Sahoo_Client-Centric Consistency_Code.zip' and extract the contents.

Run

1. Run the command

```
cd Ipsit Sahoo_Client-Centric Consistency/Project 3
```

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

```
python3 -m grpc_tools.protoc -I . .\BankService3.proto --
    python_out=. --grpc_python_out=.
```

3. Run the main.py file in the folder. By default the input file taken is input.json. You can pass the file name as shown below for any other input files. All the events are stored in the output.json. You can also track the process in the logging when the program runs.

```
python main.py input.json
```

4. To run the checker scripts. Note that the file output.json needs to be in the same path as the checker script.

```
python checker.py output.json
```

Result

The results will be printed on the console output,after you run the third command in the run section. The required output gets listed in the file output.json.

Implementation Processes

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

   In this updated communication protocol, two distinct Remote Procedure Calls (RPCs) have been introduced: **MsgDelivery** and **MsgPropagate**. These RPCs are carefully designed to handle events pertaining to customer and branch interactions, respectively. This nuanced division ensures a more granular and tailored approach to managing communication events.

   The parameters of MsgRequest and MsgResponse have been thoughtfully extended to accommodate the demands of a more sophisticated system. I have changed some of the variables from Project 1 to accommodate the new requirements.

   (a) **branch**: This variable captures the branch ID included in the customer request, enabling the code to find the exact process for routing within GRPC. This ensures the request is directed to the appropriate branch, optimizing processing efficiency and system coherence.

(b) **writeset:** This specialized GRPC-native data structure is designed for handling repeated integer data, storing a sequence of events specific to write operations, particularly in the context of branch-specific deposit and withdrawal transactions. The primary objective is to consistently maintain the write set across various branch processes, ensuring that subsequent reads are effectively processed and synchronized, thereby enhancing the overall integrity and coherence of the system.

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) **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.

   (d) **Worker Process Joining**: Once all customer and branch processes have completed their tasks, We ensure to wait for all customer processes to finish their process before moving on to the next step. This step is crucial for maintaining the order and integrity of the banking system's operations.

   (e) **Storing Event Orders** : After the branches and customer processes complete, the implementation collects the customer events and puts them into a json format to save in output.json file.

3. **Branch class**

   In the branch implementation, we have specific functionalities: create_stubs, MsgDelivery, MsgPropagation, process_message, propagate_request. Here's a breakdown of how these components work:

   (a) **process_message**: This method manage customer interactions with the branch and branch-to-branch communication. Whether a customer initiates a query, withdrawal, or deposit request, or the branch propagates a specific customer request, the corresponding event is executed at the branch. This method manages the task of ensuring the new writes are continuously updated in this branch's set before being sent to for propagation.

   (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) **MsgPropagation**: This function acts as a receiver for branch propagation requests. It listens for incoming requests from branches and forwards them to the appropriate methods for processing.

(d) **propagate_request**: 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.

(e) **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.

(f) **add_new_write, verify_writeset**: These functions are used verify the write set in each of the request and receiving entities is the same before processing the incoming request. This ensures, we are not processing a new write before the corresponding reads to the previous are not complete.

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 execute_events 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) **execute_events 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.

There is another step which discerns between QUERY and DEPOSIT / WITHDRAW event and decides parameter to be added to the messages list for the final output in the output.json file.

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 10 branches and 1 customer, we observe that the events are executed with the read-your-writes logic and maintains the order of execution. I have also added the screen-shot of the execution for reference.

Figure	Description
| ```json
Consistent balance between branch 1 and branch 2. Balance=400.0
1 out of 1 cross-branch query events are correct.
``` | Fig. 1: Results of customer checker on small input |
| ```
Consistent balance between branch 6 and branch 7. Balance=40.0
Consistent balance between branch 7 and branch 8. Balance=30.0
Consistent balance between branch 8 and branch 9. Balance=20.0
Consistent balance between branch 9 and branch 10. Balance=10.0
19 out of 19 cross-branch query events are correct.
``` | Fig. 2: Results of branch checker |

Table 1: Description of Figures

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/`

[3] Cloud Bigtable. *Read-Your-Write Consistency*. `https://cloud.google.com/bigtable/docs/writes`.