Ira A. Fulton Schools of Engineering
Arizona State University

CSE 531: Distributed and Multiprocessor Operating Systems

Client-Centric Consistency Written Report

Problem Statement

The core objective of this project is to develop and implement a client-centric banking system using gRPC in Python, addressing specific challenges in distributed systems: Monotonic Writes and Read-Your-Writes consistency. The system must ensure that client transactions are handled reliably and consistently across multiple branches of the bank.

Goal

The primary goal of this project is to design and implement a robust, client-centric banking system utilizing gRPC and Python, which adheres to specific consistency models: Monotonic Writes and Read-Your-Writes.

- **Implementation of Monotonic Writes Consistency**:

To ensure that write operations performed each client are executed and propagated in the correct sequential order across all bank branches. This involves creating a mechanism where a write operation, such as a deposit made at one branch, is recognized and accounted for in subsequent operations, like withdrawals at another branch, maintaining the write events integrity and consistency for the same client.

- **Implementation of Read-Your-Writes Consistency**:

To guarantee that each client can immediately see the effects of their write operations in subsequent read operations. For instance, after a client makes a deposit, any immediate query for their balance should reflect this recent transaction, regardless of which branch is handling the request. This objective focuses on providing a seamless and intuitive experience for clients interacting with different branches of the bank.

Setup

In this project, we use gRPC with Python to create a distributed, client-server architecture. his architecture enables efficient communication between clients and bank branches. The process involves defining service methods and message types in proto file, generating corresponding Python files, and implementing the server and client logic in Python.

1. Defining Services and Messages in Proto File

We start by creating a proto file, which contains the definitions for the services and message types that our application will use. Based on Project 1, we need modify the following Message definitions:

CustomerRequest: This message type now includes a “repeated int32 write_set” field, enabling the tracking of write operations' IDs to support Monotonic Writes consistency.

BranchRequest: Similar to CustomerRequest, this message also includes a “repeated int32 write_set” field, used for maintaining consistency in inter-branch communications.

ResponseMessage: Added an “int32 write_id” field, this message type facilitates the tracking of individual write operations, crucial for implementing both Monotonic Writes and Read-Your-Writes consistency.

```
message CustomerRequest {
    int32 customer_id = 1;
    string event = 2;
    int32 amount = 3;
    repeated int32 write_set = 4; // Track write operations' IDs

message BranchRequest {
    int32 branch_id = 1;
    int32 amount = 2;
    repeated int32 write_set = 3; // Added to maintain consistency in inter-branch communications
}

message ResponseMessage {
    string message = 1;
    bool success = 2;
    int32 write_id = 3; // Track individual write operations
}
```

2. Generating Python Files

After updating the proto file, we use the protocol buffer compiler (protoc) to generate the corresponding Python files: bankService_pb2.py and bankService_pb2_grpc.py. These files include classes for each service and message type, as well as stubs for client-side code and servicers for server-side code.

Implementation Processes

The implementation of the banking system is spread across Customer.py, Branch.py and main.py files, each handling specific functionalities and leveraging gRPC for inter-process communication as below:

1. Customer class

_ _**init**_ _**:** first initializes the customer with a unique ID and a list of events (transactions) to be executed. Based on Project 1, we added “writeSet” parameter for Customer to track write event ids.

createStub: Creates a gRPC stub, enabling communication with the bank (branch) server.

addWriteSet: Add a new write ID to the customer's write set, used for maintaining consistency.

```
def addWriteSet(self, write_id):
    # Add a new write ID to the write set
    if write_id not in self.writeSet:
        self.writeSet.append(write_id)
```

executeEvents:

Processes each event (e.g., deposit, withdraw, query) by sending appropriate gRPC requests to the bank server and handling responses. To achieve client-centric consistency goals, we include the customer's write set in the outgoing request. The write set is a collection of identifiers for write operations (like deposits or withdrawals) that the customer has previously performed. This is crucial for maintaining Monotonic Writes consistency, ensuring that the branch server knows about all previous writes done by this customer.

```
    # Include the write set in the request
    request.write_set.extend(self.writeSet)
```

Also, we need to update the Write Set upon successful operations. If the operation was successful (response.success is True) and the response contains a write_id (checked using hasattr(response, 'write_id')), the write_id is added to the customer's write set.

Another update is to update the “balance” field in recvMsg only if the event is “query”, which records only the most recent read result from the server by the customer.

```
if response.success and hasattr(response, 'write_id'):
    self.addWriteSet(response.write_id)  # Update write set only for successful operations

if event['interface'] == 'query':
    self.recvMsg["balance"] = int(response.message)
```

2. Branch class

_ _**init**_ _**:** we added self.writeSet, which is a list intended to store the identifiers (IDs) of all write operations (like deposits or withdrawals) that have occurred at this branch. It helps the branch keep track of all write operations in the order they were received and processed. Also, we added “write_id_counter “ to assign a unique ID to each write operation processed by the branch. When a new write operation is performed, the counter is incremented, and the new value is used as the “write_id”.

```
    # a list to store all write events
    self.writeSet = []
    # Initialize the write ID counter
    self.write_id_counter = 0
```

generate_write_id: This method generates a unique identifier for each write operation (like a deposit or withdrawal) performed at the branch. The method returns the current value of write_id_counter, modulo 2147483647. This modulo operation ensures that the value of write_id stays within the range of a 32-bit integer (int32) to avoid errors.

```
# Generate a unique write ID
def generate_write_id(self):
    self.write_id_counter += 1
    return self.write_id_counter % 2147483647  # Ensuring it doesn't exceed int32 range
```

updateWriteSet: This method takes a write_set (a list of write IDs) as an input.
It updates the branch's writeSet by adding the new write IDs. This is done by union operation which ensures that all unique write IDs are included in the branch's writeSet without duplication.

```
def updateWriteSet(self, write_set):
    # Update the branch's WriteSet with new write IDs
    self.writeSet = list(set(self.writeSet).union(set(write_set)))
```

checkWriteSet: The method iterates through each item in the customer_write_set and checks if it is present in the branch's writeSet. It returns True if all items in the customer's write set are found in the branch's write set, indicating that the branch is aware of all the customer's write operations. Otherwise, it returns False.

```
def checkWriteSet(self, customer_write_set):
    # Check if all customer's writes are reflected in the branch's WriteSet
    return all(item in self.writeSet for item in customer_write_set)
```

In Withdraw and Deposit methods, we need to update WriteSet first, and then need to use checkWriteSet to check and assign “Success” or “False” based on if all write items in the customer’s write set are found in the branch’s write set.

```
# Update and check WriteSet before processing
self.updateWriteSet(request.write_set)
if not self.checkWriteSet(request.write_set):
    return bankService_pb2.ResponseMessage(success=False, message="Write operations not yet propagated")
```

3. Main

This file keeps the same structure from Project 1, and modified to take “mono_input.json” and “read_your_write_input.json”.

Results

The output of the banking system demonstrates the successful implementation and validation of the Monotonic Writes and Read-Your-Writes consistency models.

1. Monotonic Write Implementation

Output: {'id': 1, 'balance': 0}

The output shows that after the series of events, the balance of the customer is 0.
This indicates that the deposit was successfully recorded at Branch 1 and was acknowledged by Branch 2 before the withdrawal, demonstrating the successful implementation of Monotonic Writes.

2. Read-Your-Writes Implementation

Output: {'id': 1, 'balance': 400}

The output demonstrates that the customer's query at Branch 2 reflects the deposit made at Branch 1, showing a balance of 400.
This confirms that the Read-Your-Writes consistency model is effectively implemented.

```
PS C:\Users\Fangjie\OneDrive\Study\ASU\CSE 531 Multiprocess Oper Sys\Project3_Client-Centric Project\bank>  python main.
py
Monotonic Write Output:
{'id': 1, 'balance': 0}
Read Your Write Output:
{'id': 1, 'balance': 400}
```