

Problem Statement

In the context of Project 1, the goal is to enhance the existing system by incorporating Lamport's logical clock algorithm. The envisioned architecture, illustrated in Diagram A, outlines the utilization of logical clocks across customer and branch processes. Lamport's algorithm will be employed for efficient clock coordination among these processes.

Major Tasks:

1. Logical Clock Implementation:
 - (a) Extend the existing Project 1 by integrating logical clocks into every customer and branch process.
 - (b) Logical clocks will serve as a crucial mechanism for tracking and synchronizing events across various processes.
2. Lamport's Clock Coordination:
 - (a) Implement Lamport's logical clock algorithm to facilitate coordination among the processes.
 - (b) Ensure that the logical clocks are updated and maintained in accordance with Lamport's algorithm to establish a consistent and ordered event sequence.

Goal

The goal of this project is to understand **Coordination**. We assume concurrent requests are being made by the customers. Customers as usual share one bank account each linked to a specific branch by a unique ID. This will result in improved synchronization and ordering of events across customer and branch processes, contributing to the overall efficiency and reliability of 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_Logical_Clock_Code.zip' and extract the contents.

Run

1. Run the command

```
cd Ipsit Sahoo_Logical_Clock_Code/Project 2
```

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

```
python3 -m grpc_tools.protoc -I . .\BankService2.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 branch events are output to **branch_output.json**. All customer events are output to **customer_output.json**. All the events are now ordered all stored in the file **output.json**

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

4. To run the checker scripts. Note there are 3 files customer_output.json, branch_output.json and output.json. These files need to be present in the folder before the script is run.

```
python checker_part_1.py customer_output.json
python checker_part_2.py branch_output.json
python checker_part_3.py output.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**

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. In addition to the existing variables carried over from Project 1, two crucial elements have been introduced:

(a) **clock:**

This variable serves as a representation of the Lamport clock, offering a systematic means of ordering events. The inclusion of Lamport clock functionality contributes to the precision and synchronization of event sequencing.

(b) **message_request_id:** Acting as a unique identifier for message requests, this variable facilitates the distinct identification of each communication instance. The introduction of this identifier enhances the traceability and management of communication events, providing a more comprehensive view of system interactions.

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.

(e) **Concurrent Customer Execution:** To maintain order and ensure proper event execution, customer processes run concurrently. This adds the extra overhead of maintaining the execution order

(f) **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.

(g) **Storing Event Orders :** After the branches and customer processes complete, the implementation collects all the events and groups them to customer and branch types and saves the data into corresponding json formatted files.

3. Branch class

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

- (a) **process_message**: These methods 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 execution includes updating the branch balance and Lamport clock. The clock is meticulously updated using a mechanism that ensures atomicity, contributing to a synchronized and coherent system. This approach not only handles diverse transactions seamlessly but also maintains the temporal integrity of the system through the strategic use of Lamport clocks.
- (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_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.
- (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) **cust_request_rcv, branch_request_rcv, branch_request_sent**: These functions serve the purpose of updating lamport clocks and adding events to the defined event list for branches.

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) **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.

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 10 customers, we observe that as the events execute the clock updates in an increasing fashion ensuring the lamport logical sequence maintained for each ordered pair of requests. I was also able to order for each branch and customer after each transaction denoting the establishment of concurrency as was the goal of the distributed systems application.

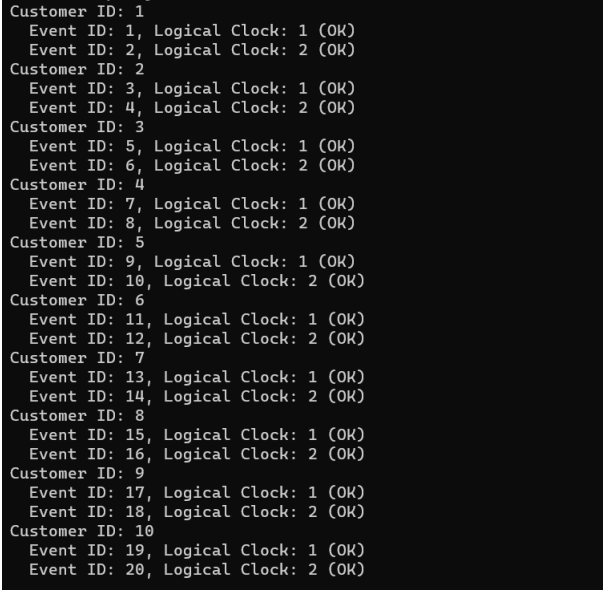
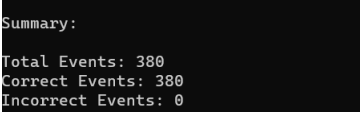
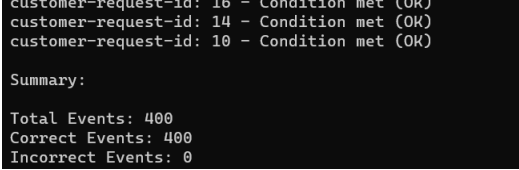
Figure	Description
Fig. 1	 <p>Results of customer checker</p>
Fig. 2	 <p>Results of branch checker</p>
Fig. 3	 <p>Results of overall checker</p>

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/>