CSE 531: Logical Clock Written Report

Shachi Shah
School of Computing Augmented Intelligence
Arizona State University
Azusa, California, United States of America
spshah22@asu.edu

I. PROBLEM STATEMENT

The core problem addressed in this project is maintaining distributed consistency in a banking system where multiple branches handle transactions concurrently. Each customer may deposit or withdraw funds from a branch, and this update must propagate across all other branches to ensure the bank's records remain synchronized. Since operations at different branches may happen simultaneously, achieving a consistent, correct state across all branches requires a mechanism to order these events accurately. This synchronization is challenging in a distributed system due to the lack of a global clock, network latency, and concurrency issues. Without a consistent ordering of events, branches could end up with conflicting information, leading to an inaccurate representation of account balances.

The problem is tackled using Lamport's logical clock algorithm, which introduces an ordering mechanism to ensure that the "happens-before" relationship between events is correctly maintained across all branches. In this system, every customer transaction, whether a deposit or withdrawal, generates an event that is propagated to every other branch. Each branch independently maintains a logical clock to order these events locally, simulating a global order and enabling reliable synchronization across distributed processes. This approach ensures that any transaction at one branch is reflected in all other branches, maintaining a consistent account state across the entire distributed network.

II. GOAL

The goal of the project is to implement a distributed consistency mechanism for the bank's branches using Lamport's logical clock algorithm within a gRPC-based framework. This solution should enforce client-centric consistency models, specifically the Monotonic Writes policy and the Read Your Writes policy, ensuring that each transaction by a customer reflects immediately in all branches. With Monotonic Writes, if a customer deposits at one branch, any subsequent withdrawals at another branch will occur after that deposit, even if requests arrive out of order due to network latency. The Read Your Writes policy ensures that if a customer deposits an amount, this balance update will be reflected immediately upon reading the account balance, even from a different branch.

Implementing this logical clock setup with gRPC for inter-branch communication aims to guarantee that all branches handle events in a specific, consistent order. Through this setup, each customer request (whether deposit or withdrawal) is processed, ordered, and propagated to other branches. By following Lamport's algorithm, the system enforces a logical, causal order for events across branches, regardless of the actual times at which events are processed by different branches.

III. SETUP

To implement the project, the following technologies and libraries are employed:

1. Python: Used for implementing client and server logic.
2. gRPC: Facilitates remote procedure calls (RPCs) between different processes, enabling communication across branches. The gRPC setup consists of defining service calls in a .proto file, which are then compiled into Python files for handling requests between branches.
3. Protocol Buffers (proto3): Used for defining the gRPC message structure, allowing for

efficient serialization of messages. The file banks.proto defines the messages and services for deposit, withdraw, and propagate requests.

4. Lamport's Logical Clocks: Implemented within the client and server logic to enforce ordering of events in a distributed manner.
5. JSON: Used to output data in a structured format for the three parts of the project, allowing verification against expected results.

All libraries used are compatible with Python 3.x, and gRPC is initialized using Python's grpcio library. For version control and collaboration, GitHub can be used to manage the project files and iterations.

IV. IMPLEMENTATION PROCESSES

The implementation can be broken down into several key steps:

1. Defining Protobuf:
    a. We start by defining the service and message types in banks.proto. This file contains service definitions for Deposit, Withdraw, and Propagate, each accepting request messages that include the customer_id, amount, and a logical clock.
    b. Here's an example of the DepositRequest message and the Deposit RPC service:

```
message DepositRequest {
    int32 customer_id = 1;
    int32 amount = 2;
    int32 logical_clock = 3;
}
service BranchService {
    rpc
Deposit(DepositRequest)
returns (Response);
}
```

    c. After defining the .proto file, it's compiled to generate banks_pb2.py and banks_pb2_grpc.py, which are used in the Python server and client code.
2. Server Setup:
    a. Each branch runs as a server, listening on a distinct port. The branch maintains an initial balance, processes requests for deposits and withdrawals, and propagates updates to other branches.
    b. A logical clock is implemented in each branch. For every incoming or outgoing event (such as a transaction request or a propagation message), the branch increments its logical clock, ensuring an incremented ordering for each event.
    c. Each branch has a method Propagate that sends an update to the other branches whenever it receives a new transaction. This method helps maintain consistency across branches by broadcasting state changes caused by deposits and withdrawals.
3. Client Setup:
    a. The client reads the input file input.json, containing requests for each customer and sends these requests to a designated branch.
    b. After initiating a deposit or withdrawal, the client records the event, including the customer-request-id and logical_clock, to a file (test_1.json).
    c. Each transaction request from a customer triggers propagation messages to other branches to reflect the update across all branches.
4. Logical Clock Synchronization:
    a. Each branch maintains a logical clock for ordering events. When a branch receives a request, it updates its clock based on the incoming logical clock and increments it further before processing. This ensures Lamport's happens-before relationship is respected.
    b. Example for clock synchronization:

```
def Deposit(self, request,
context):
    self.branch.logical_clock
=
max(self.branch.logical_clock,
```

```
request.logical_clock) + 1

self.log_event(request.customer_id,
self.branch.logical_clock,
"deposit", "event_recv")
    return
banks_pb2.Response(success=True,
logical_clock=self.branch.logical_clock)
```

c. This clock synchronization mechanism ensures that events are consistently ordered across branches.

5. Output Generation and Validation:
   a. The client and server generated outputs for verification. test_1.json logs all customer events, test_2.json logs branch events, and test_3.json consolidates all events for validation.
   b. Checker scripts (parts 1, 2, and 3) validated that each output maintains the correct logical order and met the expected structure.
   c. Then, combined the outputs for all test scripts into output.json for submission.

V. RESULTS

The implementation results in three main output files, each capturing a different aspect of event consistency:

1. test_1.json: Logs each customer's deposit and withdrawal events, detailing logical times and comments for each transaction. This file validates that each customer's requests follow a sequential and monotonic logical order.

```
Summary: 8 out of 8 customer answers are correct.
```

2. test_2.json: Captures branch events, including receiving deposits and withdrawals, along with propagation events across branches. The logical clocks in this file confirm that each branch handles events in a causally consistent order, where each propagate event correctly follows a transaction event.

```
Summary:
Branch Total Events: 56
Branch Correct Events: 56
Branch Incorrect Events: 0
```

3. test_3.json: Combines all customer and branch events, showing a comprehensive sequence of all transactions and propagate events. This file's consistency ensures the entire system adheres to the expected event order.

```
Summary:
Branch Total Events: 64
Branch Correct Events: 64
Branch Incorrect Events: 0
```

The results confirm that Lamport's logical clock implementation effectively orders events across distributed branches, allowing consistent updates for customer transactions. The logical clock values provide a means to verify the happens-before relationship, ensuring that if one event causally precedes another, its logical timestamp reflects this order. The system thus meets the goals of maintaining distributed consistency with client-centric policies, demonstrating the robustness and correctness of Lamport's logical clocks in enforcing causal ordering across distributed processes.

REFERENCES

[1] gRPC Quick Start, gRPC, https://grpc.io/docs/languages/cpp/quickstart/. Accessed 1 November 2024.

[2] Sadri, M. 2024. *CSE 531_Logical Clock Project Overview Document*, Ira A. Fulton Schools of Engineering, Arizona State University. Accessed 2 November 2024.