CSE 531: Distributed and Multiprocessor Operating Systems

Logical Clock Written Report

Problem Statement

The problem statement for the "Logical Clock" project is to implement Lamport's logical clock algorithm on top of Project 1. The objective is to enable processes (customers and branches) to use logical clocks and follow Lamport's algorithm for clock coordination, ensuring proper sequencing of events and maintaining logical time relationships based on the happens-before relationship between requests. The project involves implementing logical clocks in customer and branch processes, as well as integrating Lamport's algorithm for clock coordination among these processes.

Goal

The goal of the "Logical Clock" project is to implement Lamport's logical clock algorithm on top of the existing Project 1. Specifically, the project aims to achieve the following objectives:

1. **Implement Logical Clocks:** Integrate logical clocks into every customer and branch process. Logical clocks are crucial for tracking the ordering of events and establishing a logical time framework.
2. **Apply Lamport's Algorithm:** Implement Lamport's algorithm for clock coordination among the processes. Lamport's algorithm helps in maintaining a consistent and ordered view of events across distributed processes.
3. **Coordinate Processes:** Ensure that processes, both customers and branches, coordinate effectively using logical clocks and Lamport's algorithm. This coordination is essential for maintaining the logical ordering of events and propagating updates accurately.
4. **Input and Output Handling:** Develop mechanisms to handle input containing lists of customers and branch processes. Generate output that includes a detailed account of events, their logical times, and the relationships between customer requests and their subsequent events.
5. **Enforce Happens-Before Relationship:** Implement logic to enforce the happens-before relationship between events, both within the same process and between the send and receive events of the same request.

The ultimate goal is to create a distributed system where logical clocks and Lamport's algorithm work in tandem, providing a reliable and consistent method for tracking and coordinating events across different processes in the system.

Setup

The goal of the "Logical Clock" project is to implement Lamport's logical clock algorithm on top of the existing Project 1. Specifically, the project aims to achieve the following objectives:

1. **Implement Logical Clocks:** Integrate logical clocks into every customer and branch process. Logical clocks are crucial for tracking the ordering of events and establishing a logical time framework.
2. **Apply Lamport's Algorithm:** Implement Lamport's algorithm for clock coordination among the processes. Lamport's algorithm helps in maintaining a consistent and ordered view of events across distributed processes.
3. **Coordinate Processes:** Ensure that processes, both customers and branches, coordinate effectively using logical clocks and Lamport's algorithm. This coordination is essential for maintaining the logical ordering of events and propagating updates accurately.
4. **Input and Output Handling:** Develop mechanisms to handle input containing lists of customers and branch processes. Generate output that includes a detailed account of events, their logical times, and the relationships between customer requests and their subsequent events.
5. **Enforce Happens-Before Relationship:** Implement logic to enforce the happens-before relationship between events, both within the same process and between the send and receive events of the same request.

The ultimate goal is to create a distributed system where logical clocks and Lamport's algorithm work in tandem, providing a reliable and consistent method for tracking and coordinating events across different processes in the system.

To run the code submitted:

Please note the directory structure below

- Project Directory/
    - protos/
        - banking_system.proto
    - Customer.py
    - Branch.py
    - input.json
    - checker_part_1.py
    - checker_part_2.py
    - checker_part_3.py

Instructions:

1. Download the provided code archive from the canvas.
2. Extract all files to a single directory on your system.

3. Open your terminal and navigate to the root directory containing Branch.py and Customer.py. Execute the following command: `python3 -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=. ./protos/banking_system.proto`
4. Your setup is now complete. You can run Branch.py and Customer.py using python3

The relevant technologies for setting up this project are:

1. **Python**: This project is implemented using the Python programming language.
    - Python 3.9.6
2. **gRPC**: The remote procedure call (RPC) framework gRPC is fast and independent of programming language. In this project, it is utilized for communication between several processes.
    - Name: grpcio
    - Version: 1.59.0
3. **Protocol Buffers (protobuf)**: Protocol Buffers is a method for serializing structured data. It is used in combination with gRPC to define the service methods and message formats.
4. **Multiprocessing**: Python's multiprocessing module is used to create separate processes for both customer and branch tasks. This allows for parallel execution of these tasks. Multiprocessing is used to provide enhanced serialization, and this context to provide a sense of distributed system.
5. **JSON**: JSON (JavaScript Object Notation) is used for loading the initial data (customers and branches) into the program. This is important for setting up the

Implementation Processes

Below is the approach that I followed for this project:

Protobuf Definition (`banking_system.proto`):

- The protocol buffer file defines a service called `exchange_messages` with a single RPC method called `MsgDelivery`.
- Two messages, `Request` and `Response`, are defined to structure the communication between clients and the server. Both messages include a `logical_timestamp` field, which is crucial for implementing logical clocks.

```
syntax = "proto3";

package banking_system;

service exchange_messages {
    rpc MsgDelivery (Request) returns (Response);
}

message Request {
    int32 id = 1;
    string interface = 2;
    int32 money = 3;
    int64 logical_timestamp = 4; // Added logical timestamp field
}

message Response {
    string interface = 1;
    int32 balance = 2;
    string result = 3;
    int64 logical_timestamp = 4; // Added logical timestamp field
}
```

Customer Class (`Customer.py`):

- The `Customer` class represents a customer in your banking system.
- Each customer has a unique `id`, a list of `events` representing the operations they perform, a list to store received messages (`recvMsg`), and a gRPC `stub` to communicate with the server.
- The `createStub` method initializes the gRPC stub for communication.
- The `update_logical_clock` method updates the logical clock based on the received logical clock value.
- The `executeEvents` method processes each event, updates the logical clock, sends gRPC requests to the server, and updates the received messages.
- The `load_json` function is a utility function to load input data from a JSON file.
- The script runs multiple customer processes in parallel.

Branch Class (`Branch.py`):

- The `Branch` class represents a branch in your banking system.
- Each branch has a unique `id`, a `balance`, a list of other branch `ids` for communication, a list of gRPC `stubs` for communication, a list to store branch events (`branch_events`), and a logical clock.
- The `init_stub` method initializes gRPC stubs for communication with other branches.
- The `update_logical_clock` method updates the logical clock based on the received logical clock value.
- The `track_event` method records branch events.
- The `construct_res` method constructs a response containing the branch's id, type, and events.
- The `MsgDelivery` method handles incoming gRPC requests, updates the balance, propagates deposit and withdrawal operations to other branches, and records events.
- The `get_branch_events` method retrieves branch events.

- The `load_json` function is a utility function to load input data from a JSON file.
- The script runs multiple branch processes in parallel.

Main Execution (`__main__` section):

- The script initializes customer and branch processes and runs them in parallel.
- Customer and branch events are recorded in separate files (`customer_file.txt` and `branch_file.txt`).
- After process completion, the script reads event data from files, merges customer and branch events, sorts the merged data, and writes it to an output file (`event_file.json`).

Results

The input file is a JSON configuration that describes a system with both customer and branch processes. It consists of an array containing objects representing individual customer and branch processes. Each customer process is defined by its unique identifier ("id"), type ("customer"), and a list of events it will execute. Similarly, each branch process is defined by its unique identifier, type ("branch"), and an initial balance.

Here's a breakdown of the key components in the input:

Customer Process:

- **id:** Unique identifier for the customer.
- **type:** Type of entity (customer).
- **events:** List of events the customer will perform. Each event has:
  - **interface:** Type of operation (e.g., "deposit," "withdraw," "query").
  - **money:** (Optional) Amount of money involved in deposit or withdrawal operations.
  - **customer-request-id:** Unique identifier of the customer request.

Branch Process:

- **id:** Unique identifier for the branch.
- **type:** Type of entity (branch).
- **balance:** Initial amount of money stored in the branch.

Sample Input:

```
[
  {
    "id": 1,
    "type": "customer",
    "events": [
      {"interface": "deposit", "money": 10, "customer-request-id": 1},
      {"interface": "withdraw", "money": 10, "customer-request-id": 2}
    ]
  },
  {
    "id": 2,
    "type": "customer",
    "events": [
      {"interface": "deposit", "money": 10, "customer-request-id": 3},
      {"interface": "withdraw", "money": 10, "customer-request-id": 4}
    ]
  },
  {
    "id": 3,
    "type": "customer",
    "events": [
      {"interface": "deposit", "money": 10, "customer-request-id": 5},
      {"interface": "withdraw", "money": 10, "customer-request-id": 6}
    ]
  },
  {
    "id": 1,
    "type": "branch",
    "balance": 400
  },
  {
    "id": 2,
    "type": "branch",
    "balance": 400
  },
  {
    "id": 3,
    "type": "branch",
    "balance": 400
  }
]
```

Output Explanation:

The output consists of three parts, each containing an array of events with their logical times:

Part 1: List of Events on Each Customer

- Each customer process has an array of events with details such as customer-request-id, logical_clock (timestamp from the logical clock), interface type, and a comment describing the event.

```
[
  {
    "id": 1,
    "type": "customer",
    "events": [
      { "customer-request-id": 1, "logical_clock": 1, "interface": "deposit",
"comment": "event_sent from customer 1" },
      { "customer-request-id": 2, "logical_clock": 4, "interface": "withdraw",
"comment": "event_sent from customer 1" }
    ]
  },
  {
    "id": 2,
    "type": "customer",
    "events": [
      { "customer-request-id": 3, "logical_clock": 1, "interface": "deposit",
"comment": "event_sent from customer 2" },
      { "customer-request-id": 4, "logical_clock": 10, "interface": "withdraw",
"comment": "event_sent from customer 2" }
    ]
  },
  {
    "id": 3,
    "type": "customer",
    "events": [
      { "customer-request-id": 5, "logical_clock": 1, "interface": "deposit",
"comment": "event_sent from customer 3" },
      { "customer-request-id": 6, "logical_clock": 17, "interface": "withdraw",
"comment": "event_sent from customer 3" }
    ]
  }
]
```

Part 2: List of Events on Each Branch

- Each branch process has an array of events with similar details as Part 1. Events include information about customer requests received, propagated deposits, and propagated withdrawals.

```
[
  {
    "id": 1,
    "type": "branch",
    "events": [
      { "customer-request-id": 1, "logical_clock": 2, "interface": "deposit",
"comment": "event_recv from customer 1" },
      { "customer-request-id": 1, "logical_clock": 3, "interface":
"propagate_deposit", "comment": "event_sent to branch 2" },
      // ... (other events)
    ]
  },
  // ... (other branches)
]
```

Part 3: List of Events Triggered by Each Customer Deposit/Withdraw Request

- A separate array includes events triggered by each customer's deposit or withdrawal request. It provides details such as customer-request-id, logical_clock, interface type, and comments indicating whether the event was sent or received.

```
[
  {
    "id": 1,
    "type": "branch",
    "events": [
      { "customer-request-id": 1, "logical_clock": 2, "interface": "deposit",
"comment": "event_recv from customer 1" },
      { "customer-request-id": 1, "logical_clock": 3, "interface":
"propagate_deposit", "comment": "event_sent to branch 2" },
      // ... (other events)
    ]
  },
  // ... (other branches)
]
```

Output Justification:

The output is structured to provide a clear and organized representation of the events, their logical times, and the interactions between customers and branches. It ensures clarity by ordering events based on their logical times and categorizing them by customers, branches, and across all processes. This format facilitates understanding and verification of the happens-before relationships, as mentioned in the input explanation. The logical times help establish a chronological order of events in the system.