CISC 5597 Distributed Systems

Lab 2: Consensus and Consistency— Basic Paxos Algorithm

Name: Swoichha Adhikari

Overview:

This report outlines the code development and workflow of a Basic Paxos Algorithm using XMLRPC library and multihreading in Python's programming. The server can accept multiple request to update the value. It also accepts different commands like set value, A_wins, B_wins, refresh, and exit from clients and will handle them accordingly. Once a client is connected to a server, clients can send and receive messages from server to update a distributed value. The code also handles errors for scenarios like sending a message from a client to itself, trying to use incorrect commands, using an invalid client ID to perform some invalid actions, etc.

Methodology:

Technologies Used in the codebase:

1. Python 3.x
2. Socket programming
3. Multithreading and ThreadPoolExecutor

System Architecture:

The application consists of two main components: the server and the client. The server-side code is written in the file ***server.py*** and the client-side code is written in the file named ***client.py.***

Client interacts with the nodes by sending commands to set values, propose a winner, and restart nodes whereas Server Implements the Paxos protocol for value proposals, prepares, accepts, and commits values to ensure consensus is reached across the nodes.

A. Client:

The client connects to multiple servers running on different ports and sends various commands to manage the system. It provides commands such as:

- ` A single proposer.`, `A_wins`, `B_wins (first scenario) `, `B_wins (second scenario) ` for simulating different proposal values across nodes.

- `set value` to propose a value to a specific node and its peers.

- `restart` to reset all nodes to their initial state.

- `exit` to terminate the program.

B. Server

Each server in the system manages its state using the Paxos protocol. Servers handle:

- Proposal phase: The node accepts or rejects proposals from other nodes.

- Prepare phase: Nodes broadcast the proposal to peers and respond based on their state.

- Accept phase: Nodes try to reach a majority consensus on a value and broadcast the decision.

- Commit phase: Once a consensus is reached, the node commits the value and updates its local file.

Key Challenges and Solutions:

1. Delay

When handling multiple activities at once, it might be challenging to add a delay to mimic network lag or processing time. Because of the operating system's scheduling, threads may encounter additional, unpredictable delays, therefore a delay does not ensure that they will finish in a particular sequence.

2. Race Conditions and Shared State
    - Race situations may arise when several threads alter common resources.
    - Simultaneous attempts by threads to read or write these variables could produce inconsistent or inaccurate results.

3. Debugging and Logging Concurrency Issues
    - Because the order of execution is non-deterministic in a concurrent setting, debugging problems becomes more challenging. Certain states may be difficult to replicate, and logs may appear out of order.

4. Testing and Verification of Concurrency Logic

- Because timing-dependent behavior may produce inconsistent results, testing concurrent programs with delays might be unreliable. It is difficult to replicate mistakes and confirm accuracy because of this discrepancy.

How to Execute:

5. Client-side (Main Functionality): python client.py

The client interacts with the servers by sending various commands. Key functions include:

- send_propose: Sends a proposal to a server and logs the response.

- restart_all_nodes: Sends a restart command to all nodes, resetting their state.

- main: The entry point of the client program where the user interacts with the system via commands. Based on the command input, different operations such as value proposals, node restarts, or value settings are triggered.

6. Server-side (Paxos Algorithm Implementation)

The server implements the Paxos protocol to ensure consensus:

- MyServer: This class defines the behavior of each server node, including handling proposals, accepting or rejecting values, and broadcasting decisions.

- State Management: It uses locks (`acceptedProposal_lock`, `acceptedValue_lock`) to safely manage shared state among multiple threads.

7. **Key Features**

- Concurrency: The use of `ThreadPoolExecutor` allows multiple proposals to be sent to different servers in parallel, improving the system's responsiveness.

- Error Handling: Comprehensive error handling is in place for all server interactions, ensuring that any failure in communication or processing is logged and can be diagnosed.

- Logging: Extensive logging is used throughout the system, making it easier to track the state of the system, proposals, and decisions.

- State Management: Each node maintains its own state using `acceptedProposal` and `acceptedValue`, ensuring that proposals are accepted based on the Paxos algorithm's rules.

7. **Conclusion**

This project successfully implements a distributed consensus system using the Paxos protocol, allowing multiple nodes to agree on a value despite failures or concurrent proposals. The system's

total performance is enhanced by the use of threading and concurrency, while the client-server design guarantees seamless communication between nodes. The implementation offers a strong starting point for learning about and experimenting with consensus methods and distributed systems.