CISC 5597 Distributed Systems

Lab 2: Consensus and Consistency— Basic Paxos Algorithm

Name: Swoichha Adhikari

Introduction

In this lab, I have implemented the Basic Paxos consensus algorithm in a distributed environment. Paxos is a protocol for achieving consensus among a distributed set of nodes (or servers) where a group of nodes must agree on a single value, even in the presence of failures. This implementation utilizes XML-RPC for communication between nodes and simulates the decision-making process based on proposals, votes, and value commitments.

Objectives

- Implement the Paxos consensus protocol using a distributed approach

- Create a client-server architecture where multiple nodes communicate using XML-RPC

- Ensure that nodes can propose, agree/promise, and eventually commit to a value

- Handle potential failures and restarts of nodes, ensuring that consensus is still reached

System Overview

Components:

1. Client:

- The client interacts with the nodes and sends commands to trigger different phases of the Paxos protocol (such as proposals and commitments).

- It supports commands like:

  - Set value <vale> on <port_for_node>: Simulates a scenario for single proposer
  - A_wins: Simulates a scenario for two proposers where the first node (A) proposes a value and wins(slide 22)
  - B_wins: Simulates a scenario for two proposers where the second node (B) wins a proposal(slide 23)
  - Restart: Resets all nodes to their initial states.

2. Server:

- Each server represents a single node in the Paxos protocol.

- Nodes communicate with each other to propose values and reach a consensus.

- Each server handles three key steps:

- Prepare: A node prepares to propose a value by checking if it can accept it based on previous proposals.

- Accept: If the proposal is accepted, it will be committed to the system.

- Commit: The node will update its local state (in a file) once a value is committed.

Architecture:

The system consists of a set of servers (nodes) running on different ports. Each server is responsible for handling proposals, responses, and committing values. The client sends commands to initiate actions on the servers, such as proposing values, accepting values, or restarting nodes.

The communication between the client and the servers is done using XML-RPC. The protocol's flow involves broadcasting proposals, handling responses, and committing agreed-upon values across all nodes in the system.

Technologies Used:

- XML-RPC: A remote procedure call (RPC) protocol that uses XML to encode the messages.

- Threading and Concurrency: Python’s threading and concurrent.futures libraries are used to handle multiple operations concurrently, such as sending requests to multiple nodes simultaneously.

Methodology:

Steps in the Paxos Algorithm:

1. Prepare Phase:

- A proposer (client) sends a prepare request to a majority of nodes, indicating a proposal number.

- Each node responds with its highest accepted proposal number and value, if any.

2. Propose Phase:

- After receiving a prepare response from a majority of nodes, the proposer sends a propose request to the same nodes, asking them to accept a value.

3. Commit Phase:

- Once a majority of nodes accept the proposal, the value is committed, and all nodes update their local state.

Communication Between Nodes:

1. The client sends a command to a set of nodes, triggering the prepare phase.
2. Each server processes the proposal and responds with its decision (accept or reject).
3. Once the majority of servers accept the proposal, the client triggers the commit phase, and all servers update their state to reflect the consensus.

Handling Failures:

- The servers are designed to restart and clear their state (file reset) on command “restart”, simulating a clean scenario to run and test different cases without closing the terminal and rerunning the code.
- The nodes can recover from crashes and still reach consensus by re-executing the Paxos protocol after recovery.

Concurrency:

- The client uses threading to simulate multiple proposals happening concurrently.
- The servers use thread locks to ensure that shared variables (such as accepted proposals and values) are accessed safely across different threads.

Key Functions:

1. Client-side Functions:
    - send_propose(): Sends a proposal to a server.
    - restart_all_nodes(): Restarts all nodes to their initial state.
    - main(): Processes user inputs and triggers commands like A_wins, B_wins, and others.

2. Server-side Functions:

- prepare(): Handles the prepare phase of the Paxos protocol.
- accept(): Accepts a proposal if it meets the criteria.
- broadcast_commit(): Broadcasts a commit message to other nodes.
- update_file(): Updates the local file to reflect the committed value.

Experimental Setup:

- Five servers were simulated, each running on a different port (8000 to 8004).

- The client interacted with these servers by sending commands to initiate the Paxos protocol and test different scenarios (such as A_wins, B_wins, and set value).

- The system was tested for its ability to recover from failures by using the restart command and ensuring that consensus was still maintained after recovery.

Sample Commands and Outputs:

1. Command: A_wins 8000 8004 123 456

- Output: Node 8000 responded: Value '123' has been updated and committed.

- Output: Node 8004 responded: Value '123' has been updated and committed.

2. Command: B_wins 8000 8004 456 123

- Output:

2024-11-10 12:13:29,516 - INFO - Node 8004 responded: Value '456' has been updated and committed.

2024-11-10 12:13:30,410 - INFO - Node 8000 responded: Value '456' has been updated and committed.

2024-11-10 12:13:30,410 - INFO - Final response from Node 8000: Value '456' has been updated and committed.

2024-11-10 12:13:30,410 - INFO - Final response from Node 8004: Value '456' has been updated and committed.

3. Command: restart

- Output: All nodes are restarted and their states reset.

Challenges

- Concurrency Issues: Handling multiple concurrent requests and ensuring thread safety in the client and server was a challenge. This was resolved using thread locks to manage access to shared variables.

- Network Delays: Simulating network delays between nodes introduced complexity, but it was successfully handled by adding sleep intervals between operations.

How to Execute:

First change the IP and port number in each file.

1. Start the Server:
    - Run the server-side script: python3 server.py.
    - The server will start listening for request from client to set value on IP:port.
2. Start the Client:
    - Run the client-side script: python3 client.py.
3. The client now can send value to be updated on the distributed file using available commands.
4. Available Commands:

    All the commands are case in-sensative.

    - set value <value> on <port_of_node>
    - A_wins <port_of_node1> <port_of_ node1> <value_for_node1> <value_for_node2>
    - B_wins <port_of_node1> <port_of_ node1> <value_for_node1> <value_for_node2>
    - BB_wins <port_of_node1> <port_of_ node1> <value_for_node1> <value_for_node2>
    - restart