## Report on Basic Paxos Algorithm

**Note**: My original version of this program ran all three nodes on one terminal. If you would like to view that code, you can at this github link. It should produce the same results as the submitted version.

### Introduction

In this lab, we explored the implementation of the Basic Paxos consensus algorithm within a distributed file system consisting of three computing nodes. The Paxos algorithm allows nodes in a distributed network to agree on a single proposed value, achieving consensus despite potential network asynchrony and failures. This setup serves as a fundamental model of how consensus can be reliably achieved in distributed systems, forming the basis for data replication and fault-tolerant storage in real-world applications.

### System Design Outline

The system was designed to handle the following key elements:

1. **Node Representation**: Each of the three computing nodes is represented by a `Node` class that maintains a replica of the distributed file `CISC5597`. These nodes are connected through Remote Procedure Calls (RPCs) to allow inter-node communication.
2. **Client Interaction**: A client application submits a value to the nodes, triggering the Paxos protocol to reach consensus on the value.
3. **Basic Paxos Protocol**: The Paxos protocol implemented here is based on the two-phase approach. Each node can propose values to others and also act as an acceptor to incoming proposals, achieving consistency through majority rule.

This design ensures that even in cases of concurrent proposals, the Paxos protocol will enforce a single accepted value among the nodes.

## Node Class

The `Node` class is the core of this implementation, encapsulating both the logic of the Paxos algorithm and the necessary mechanisms for communication and concurrency in a distributed environment. Each `Node` instance represents one of the three computing nodes in the network. Here's a breakdown of the primary functionalities encapsulated within the `Node` class:

1. **Paxos Algorithm Logic**: The `Node` class implements the two-phase Paxos algorithm, with functions for proposing values, preparing responses, accepting proposals, and committing values. Each node can act as both a proposer and an acceptor, allowing for a flexible and robust consensus protocol.
2. **RPC Communication**: Each `Node` communicates with other nodes through Remote Procedure Calls (XML-RPC). This network communication allows nodes to share

proposals, acceptances, and final values, simulating the distributed nature of the system. RPCs are handled with a lightweight communication protocol, allowing each node to invoke methods on its peers.

3. **Concurrency Management with Threads**: To simulate asynchronous behavior and enable parallel processing, each node runs its RPC server and a separate monitoring process in dedicated threads:
    a. **Server Thread**: The RPC server runs in a background thread, continuously listening for incoming requests and processing them concurrently. This allows nodes to handle multiple requests simultaneously.
    b. **Inactivity Monitor Thread**: A separate monitoring thread periodically checks for node inactivity, enhancing fault tolerance and system stability. This thread can identify inactive nodes or network issues, supporting robust distributed consensus.

```
# Initialize the server and threads
self.rpc_server = ThreadedXMLRPCServer(("localhost", port), allow_none=True)
self.rpc_server.register_instance(self)
self.rpc_server.logRequests = False  # Disable request logging
self.rpc_server.allow_reuse_address = True


self.server_thread = threading.Thread(target=self.rpc_server.serve_forever, daemon=True)
self.server_thread.start()
self.monitor_thread = threading.Thread(target=self.monitor_inactivity, daemon=True)
self.monitor_thread.start()
```

## Paxos Consensus Process

The Paxos consensus process in this implementation consists of three primary phases managed within the `Node` class: **submission**, **prepare**, and **accept**. These phases work together to ensure consensus is achieved across the distributed nodes, even under conditions of network delay or concurrent proposals.

1. **Submit Value**: The `submit_value` method initiates the consensus process when a client submits a value. If consensus has not yet been reached, the node sets a unique proposal ID and initiates the prepare phase. It uses the `simulation_case` parameter to introduce controlled network delays, simulating different Paxos scenarios like livelock and concurrency. The client can also submit its own `proposed_id` for testing purposes.
2. **Promise**: The `promise` method allows a node to respond to a prepare request, agreeing to a proposal if its proposal ID is higher than any previous promise or accepted ID. This helps maintain protocol integrity, as only the highest proposal ID is guaranteed consideration. If a lower ID is proposed, the promise is rejected, effectively aborting the proposal.
3. **Prepare Phase**: In this phase, the node sends a "prepare" request to other nodes, asking them to "promise" not to accept proposals with lower IDs. Nodes return a promise if the proposal ID is higher than any previously accepted proposal. During this process, nodes may adopt previously accepted values with higher IDs, ensuring that any partially

accepted proposal gains priority. This phase ends with either gathering a majority of promises or failing if enough promises aren't received after multiple attempts.

4. **Acceptance**: The `accept` method finalizes a node's acceptance of a value. Nodes will only accept a value if it has a proposal ID equal to or greater than any previously promised or accepted ID, thus maintaining the protocol's strict hierarchy.
5. **Accept Phase**: If a majority of promises is secured, the node proceeds to the accept phase, proposing that all nodes accept the proposed value. During this phase, each node will either accept or reject the value, depending on whether it conflicts with a higher proposal ID. The node tracks acceptance counts and considers consensus reached when a majority of nodes accept the value.
6. **Consensus and Broadcast**: Once a majority of nodes have accepted a proposal, consensus is reached, and the node broadcasts the consensus value to all nodes. This ensures all nodes replicate the agreed-upon value, achieving consistency in the distributed file system. Each node writes the agree-upon value to their file ("`CISC5597_node[#].txt`").

Through controlled delays and concurrency management, this design handles scenarios like network latency, conflicting proposals, and livelock, showcasing the resilience and efficiency of Paxos in distributed consensus.

## Error Handling

The Paxos implementation includes robust error handling mechanisms on both the server and client sides to ensure the system remains stable and responsive under various network and operational challenges. These mechanisms address issues such as timeouts, connection errors, and node inactivity.

**Server-Side Error Handling**

1. **Timeouts in Node Communication**:
    - During the consensus process, each node communicates with its peers through RPC calls. However, network latency or node unavailability can lead to timeouts. If a timeout occurs when attempting to connect to a peer, the server logs the issue and continues with the process rather than crashing, allowing the consensus process to proceed with available nodes.
2. **Graceful Shutdown on Keyboard Interrupt (Ctrl+C)**:
    - For testing or maintenance, a user may need to interrupt the node processes manually. The server includes handling for `KeyboardInterrupt` (Ctrl+C), which allows the node to shut down gracefully.
3. **Inactivity Monitoring and Node Shutdown**:
    - Each node monitors its activity level to detect prolonged inactivity. If a node becomes inactive (e.g., no requests received or no actions performed for a specified period), it can automatically initiate a shutdown.
    - This inactivity shutdown is managed by a dedicated monitoring thread within each node, periodically checking the last activity timestamp. If the threshold is

exceeded, the node shuts down to free up resources and prevent the system from holding on to unresponsive nodes.

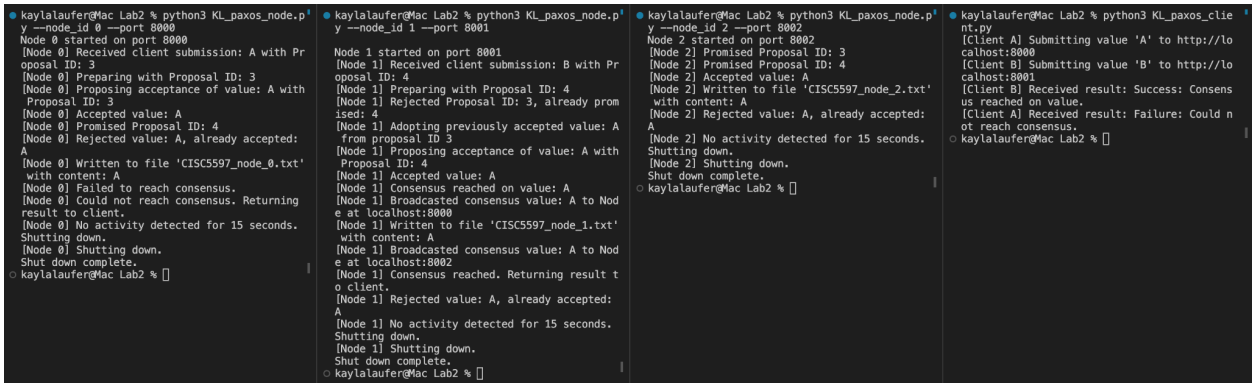
**Client-Side Error Handling**

1. **Connection Refusals**:
    - When a client attempts to connect to a node that is unavailable or down, it may encounter a connection refusal error. The client is equipped to handle such refusals by capturing the exception and alerting the user.

## Simulation Scenarios

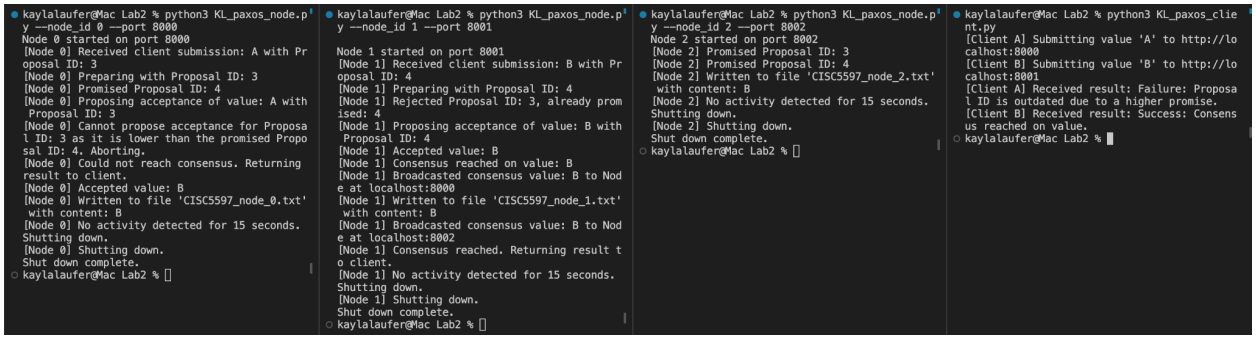
The Paxos implementation includes several simulation scenarios to test and demonstrate the algorithm's behavior under different conditions, mirroring classic Paxos cases and real-world challenges.

1. **Single Client Submission**: In this scenario, a single client submits a value to the server, and the Paxos algorithm reaches consensus on this value. This serves as a baseline case, demonstrating the protocol's normal function with one proposer.
2. **Two Clients with Random Proposal IDs**: Two clients submit values with randomly assigned proposal IDs. The protocol handles the competing proposals by establishing consensus on one of the values based on the highest proposal ID, demonstrating Paxos's ability to handle concurrent submissions.
3. **Consensus Reached, Second Client Attempts Submission**: Here, Client A reaches consensus on a value. When Client B attempts to submit a new value, it is informed that consensus has already been achieved, and no new values can be accepted. This scenario illustrates Paxos's ability to enforce consensus once reached, preventing additional submissions.
4. **Slide 24 Scenario**: Client A submits a proposal with ID 3 (to node 0), receives majority promises, and moves to the accept phase. Client B then submits (to node 1) a proposal with a higher ID (4), adopting the majority-agreed value (from A) but proceeding with ID 4. Both proposals succeed, with Client B reaching consensus on Client A's value.

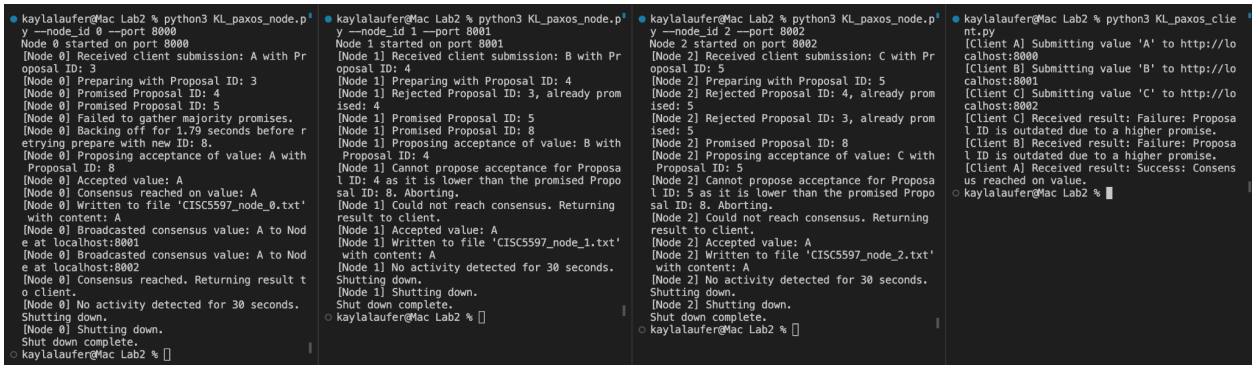



5. **Slide 25 Scenario**: Client A submits a proposal (ID 3 to node 0) and reaches the prepare phase with majority promises. Before acceptance, Client B submits with a higher

ID (4 to node 1), gathers majority promises, and proceeds independently. Consensus is reached on Client B's value, blocking Client A's completion. This illustrates Paxos's prioritization of the highest proposal ID, ensuring system convergence on a single value.




6. **Livelock Scenario with Randomized Backoff**: Multiple nodes (Node 0, Node 1, and Node 2) submit nearly simultaneously with different proposal IDs, leading to repeated interference where no proposal can reach acceptance. By introducing a randomized backoff delay, nodes are given time to reattempt with adjusted proposal IDs. This breaks the livelock, allowing one node to proceed to consensus, demonstrating the effectiveness of backoff in resolving livelock in Paxos.




Note, for scenarios 4-6, controlled network delays are implemented to help reach the state needed to see these effects. Due to actual network delays, these results may vary across multiple attempts. Screenshots are provided to illustrate the depicted cases.

**Challenges**

Implementing this Paxos protocol required overcoming several challenges:

1. **Concurrency**: Ensuring nodes could handle multiple requests simultaneously was crucial for simulating a distributed environment. This required implementing a threaded XML-RPC server and carefully managing shared resources, like the proposal ID and accepted value, to avoid race conditions and ensure correct protocol execution across threads.
2. **Creating Simulations**: Testing different scenarios—such as proposal conflicts, livelock, and timeout conditions—was essential to validate the protocol's behavior. This involved

introducing configurable delays and randomized backoff strategies to simulate network latency and contention, helping to observe how nodes react in competitive and unstable network environments.

3. **Graceful Shutdown**: Since each node operates independently and can be idle for extended periods, implementing a clean shutdown process was challenging. This required setting up inactivity monitoring and handling shutdown signals to ensure each node could terminate gracefully without leaving processes hanging, especially during network delays or unresponsive peers.

### Imports and How to Run

```
import xmlrpc.server
import xmlrpc.client
import threading
import time
import random
import sys
import socket
import signal
import argparse
from socketserver import ThreadingMixIn
```

To run the server, you must include the node ID and the port. You need to have a separate terminal open for each of the three nodes. Here are the terminal commands:

# On terminal 1
`python3 KL_paxos_node.py --node_id 0 --port 8000`

# On terminal 2
`python3 KL_paxos_node.py --node_id 1 --port 8001`

# On terminal 3
`python3 KL_paxos_node.py --node_id 2 --port 8002`

Then in the fourth terminal, run the client code:

`python3 KL_paxos_client.py`

To run each scenario, comment out the corresponding test in the `KL_paxos_client.py` file. Ensure it is the only case commented out.

If running on the cloud, ensure that `'localhost'` is substituted with the internal IP of the corresponding server.

Server:

```
# Configuration for 3 nodes with different ports
NODES = [("localhost", 8000), ("localhost", 8001), ("localhost", 8002)]
```

```
# Initialize the server and threads
self.rpc_server = ThreadedXMLRPCServer(("localhost", port), allow_none=True)
```

Client:

```
NODE_URL_A = "http://localhost:8000"  # Node for Client A
NODE_URL_B = "http://localhost:8001"  # Node for Client B
NODE_URL_C = "http://localhost:8002"  # Node for Client B
```

**Conclusion**

This lab has demonstrated the implementation of a Basic Paxos protocol for achieving distributed consensus. The system successfully handled both single and multiple proposer scenarios, utilizing a two-phase approach to ensure consistency across nodes. By handling concurrency and using randomized delays, the implementation mirrors the challenges of real-world distributed systems. This project highlights the effectiveness of the Paxos algorithm in creating fault-tolerant systems, especially where consensus must be reached in asynchronous networks.