System Architecture

The system is built to implement the Paxos consensus algorithm, which is a protocol for achieving distributed consensus across multiple nodes. This implementation separates nodes into three roles:

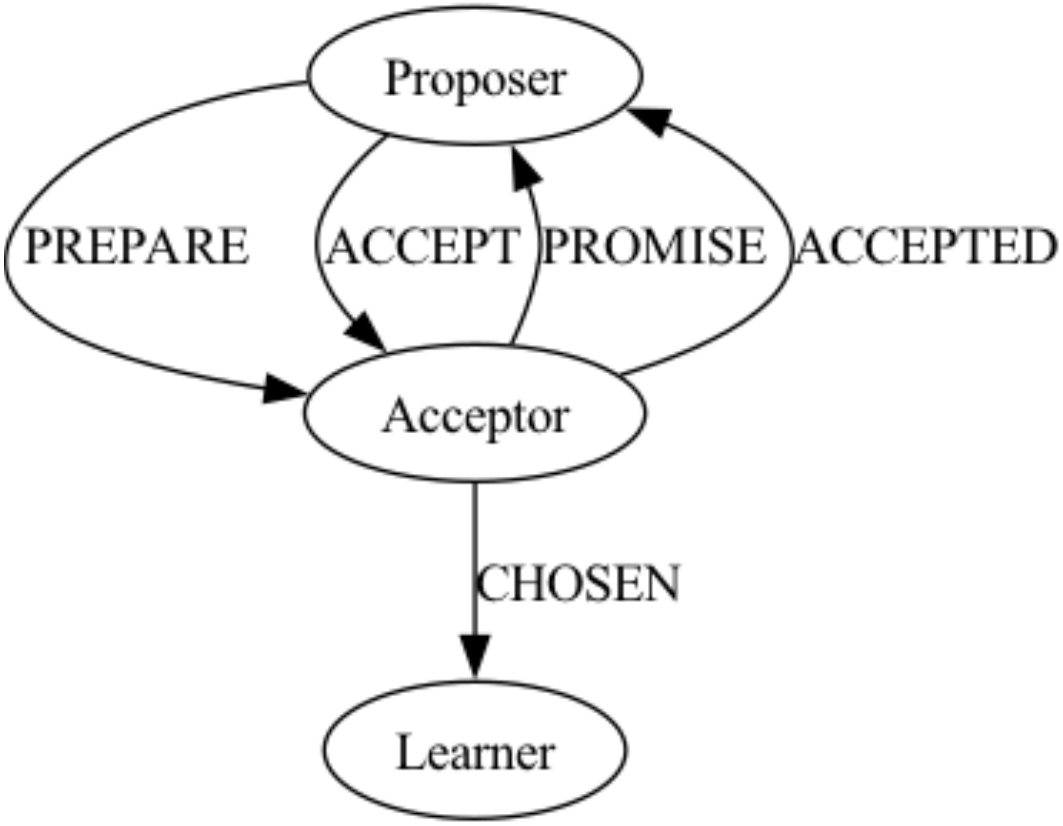
- **Proposer**: Initiates proposals and attempts to reach consensus by coordinating with Acceptors.
- **Acceptor**: Receives and decides on proposals, helping to enforce consensus through responses.
- **Learner**: Receives the final decision after consensus is reached.

Each node maintains several key parameters, including identifiers, proposal numbers, and accepted values. Communication between nodes occurs over TCP sockets, and JSON is used to structure and log messages.

State Diagram

The Paxos algorithm involves the following states and message flows:

1. **Proposer**:
    - Begins by sending a `PREPARE` message to Acceptors.
    - If a quorum (majority) of `PROMISE` responses are received, it proceeds by sending an `ACCEPT` message with the proposal value.



Caption

2. **Acceptor**:

- Responds with a `PROMISE` if it can accept the proposal number, otherwise sends `NACK`.
- Upon receiving an `ACCEPT` request from a Proposer, it may respond with `ACCEPTED` if the proposal is valid.

3. **Learner**:
   - Receives a final `CHOSEN` message indicating that consensus is reached.

Design Decisions

1. **Modular Classes and Enums**:
   - The code is structured around the Paxos roles: `PROPOSER`, `ACCEPTOR`, and `LEARNER`.
   - Messages are defined with an enum `MessageType`, which keeps the message types centralized, reducing potential errors.
2. **Network Communication and Synchronization**:
   - Socket communication is used to allow distributed deployment across a network.
   - Semaphores (`sem_t`) and mutex locks ensure synchronized access to shared resources, such as `prepareResponses` and `acceptResponses`.
3. **Consensus Handling**:
   - Each Proposer's proposal includes a unique proposal number generated to avoid conflicts.
   - Acceptors use `lastPromisedProposalNum` to track the highest proposal they've promised to respond to, ensuring ordered and reliable proposal handling.
   - The Proposer only proceeds if a quorum of Acceptors responds positively to its `PREPARE`.
4. **Concurrency**:
   - Threads handle concurrent connections to support communication with multiple nodes simultaneously.
   - The `listenForMessages` function enables each node to handle incoming messages asynchronously.

Implementation Issues

1. **Concurrency Management**:
   - Achieving smooth concurrency in a distributed environment is complex. Synchronization mechanisms (e.g., mutexes, semaphores) are essential but can lead to deadlock if not carefully managed.
2. **Socket Communication**:
   - Sockets must be correctly initialized, bound, and managed across different nodes.
   - Error handling during socket creation and connection is crucial, as network disruptions can prevent messages from being delivered reliably.
3. **JSON Serialization for Logging**:
   - JSON is used to structure message data for easy parsing and readability. However, concurrent output could still interleave due to shared stdout usage. A

`safePrintJson` function was implemented to manage this, although further optimization may be needed under heavy loads.

4. **Proposal and Consensus Validity**:
    - Ensuring proposal numbers remain unique and ordered is critical to the Paxos algorithm. Thus, the `generateProposalNumber` method provides unique proposal IDs for each proposer.