DESIGN DOCUMENT PA 4.1 (GIGAPAXOS)

Team Members:

Sri Abhijit Chunduru (SPIRE ID: 35247123)
Shri Venkatakrishnan Vasudevan (SPIRE ID: 35232787)

System Architecture

The architecture follows a standard RSM pattern:

- **Client:** Sends requests to the GigaPaxos service.
- **GigaPaxos Layer:** Handles leader election, failure detection, and consensus. It ensures that every live replica receives the same stream of requests in the exact same order.
- **Application Layer (MyDBReplicableAppGP):** Receives "committed" requests from GigaPaxos via the execute() interface and applies them to the local Cassandra instance.
- **Storage Layer:** A local Cassandra instance acting as the persistent state for that specific replica.

Component Interaction

- **Request Arrival:** A client sends a request (CQL string) to the GigaPaxos service.
- **Consensus:** GigaPaxos ensures the request is replicated to a quorum of nodes.
- **Commit & Execute:** Once committed, GigaPaxos calls execute(Request) on the MyDBReplicableAppGP instance.
- **Persistence:** The application executes the CQL query against the local Cassandra keyspace.
- **Response:** The result of the query is captured and returned to GigaPaxos, which routes it back to the client.

Implementation Details

Request Execution (execute)

The execute method is the core entry point for the state machine.

- **Input:** Receives a RequestPacket containing the command (typically a Cassandra CQL string).
- **Sanity Checks:** The code verifies that the request is targeting the allowed table (grade) to prevent accidental corruption of system tables.
- **Execution:** The raw string is submitted to the Cassandra session.
- **Idempotency & Determinism:** Since GigaPaxos guarantees the order of delivery, the application assumes that executing these commands in order results in a deterministic state.
- **Response Handling:** The ResultSet from Cassandra is converted to a string and attached to the RequestPacket for the client response.

Checkpointing (checkpoint)

To prevent the command log from growing infinitely, the system implements state checkpointing. When triggered by GigaPaxos:

1. **Snapshot:** The app queries SELECT * FROM table.
2. **Serialization:** The entire table state is serialized into a generic JSON format.
   - **Key:** The Primary Key columns (serialized as ColName|Value).
   - **Value:** The data columns (serialized as ColName|Value|ColName|Value...).
3. **Output:** A JSON string representing the exact state of the database at that moment is returned to GigaPaxos.

State Restoration (restore)

When a node recovers after a crash or falls too far behind, GigaPaxos calls restore with a previously saved checkpoint string.

1. **Parsing:** The JSON checkpoint string is parsed into a Map.
2. **Reconstruction:** The app iterates through the map. For every entry, it constructs a CQL UPDATE statement.
   - The map **Value** becomes the SET clause.
   - The map **Key** becomes the WHERE clause.
3. **Execution:** These UPDATE statements are executed against Cassandra, bringing the local node up to the state of the checkpoint.
4. **Log Replay:** After restore completes, GigaPaxos automatically replays any requests that occurred *after* the checkpoint was taken, bringing the node to the most current state.

Fault Tolerance Strategy

Crash Recovery

Because the system is an RSM, crash recovery is handled largely by the GigaPaxos protocol.

- **If the Leader fails:** GigaPaxos elects a new leader automatically.
- **If a Replica fails:** When it restarts, MyDBReplicableAppGP re-initializes the Cassandra connection. GigaPaxos detects the node's return and initiates the restore() process (if logs were truncated) or simply replays missing requests via execute().

Consistency

The system guarantees **Linearizability**.

- All updates go through the consensus log.
- Reads are implicitly ordered if they are sent through the same GigaPaxos pipeline.
- The synchronized block in execute ensures that a single node processes requests serially, matching the sequential nature of the Paxos log.

DESIGN DOCUMENT PA 4.2 (ZOOKEEPER)

Team Members:

Sri Abhijit Chunduru (SPIRE ID: 35247123)
Shri Venkatakrishnan Vasudevan (SPIRE ID: 35232787)

Implementation details

- **Leader Election**
  Leader election is implemented using ZooKeeper's **Ephemeral Sequential** nodes under the /election path.

1. On startup, every server creates a znode: /election/n_.
2. Servers fetch all children of /election and sort them.
3. **The Leader:** The node with the numerically smallest sequence number.
4. **The Watch:** Non-leader nodes set a ZooKeeper Watch. If the Leader crashes (ephemeral node disappears), the election logic re-runs immediately.

- **Request Handling (The Write Path)**

To ensure **Linearizability**, all write operations must be ordered globally.

1. **Client Request:** A client sends a request (SQL string or JSON) to any random replica.
2. **Forwarding:**
   - If the receiver is the **Leader**: It proceeds to step 3.
   - If the receiver is a **Follower**: It forwards the raw bytes to the current Leader using serverMessenger.
3. **Ordering (Consensus):** The Leader wraps the request payload with the original Sender ID (e.g., server.1::INSERT...) and creates a **Persistent Sequential** znode in /log/entry-.
4. **Execution:**
   - All nodes (Leader and Followers) Watch the /log path.
   - When NodeChildrenChanged triggers, nodes fetch the new log entries.
   - Entries are processed in strict sequence order (tracked by lastExecutedSeq).
5. **Response:** After executing the query against Cassandra, the node extracts the Sender ID from the payload and sends the acknowledgment/response back to that specific server (or directly to the client if the node is the origin).

- **Checkpointing and Log Compaction**

To satisfy the constraint MAX_LOG_SIZE = 400 and ensure fast recovery, the system implements a "Snapshot in ZooKeeper" checkpointing strategy.

- **Checkpoint Creation:**
  - The Leader monitors the log size. Every CHECKPOINT_INTERVAL (200 requests), it queries Cassandra for the full state of the grade table.
  - The state is serialized into a string format: SeqNum|ID:Events;ID:Events.

        - This snapshot is written to a persistent znode: /checkpoint/cp-{SeqNum}.
- **Log Pruning:**
- After a successful checkpoint, the Leader deletes old nodes in /log that have sequence numbers lower than the checkpoint, ensuring the ZK log does not grow unboundedly.
- **Failure Recovery**

The system handles crash-stop failures using a "Restore and Replay" mechanism.

- **Server Crash**

When a server process dies:

1. Its connection to ZK is lost.
2. Its Ephemeral node in /election is deleted automatically by ZK.
3. If it was the Leader, the next node in the sequence detects the deletion and promotes itself.

- **Server Recovery (Restart)**

When a server restarts, it has lost its in-memory state (lastExecutedSeq resets to -1). To prevent data corruption or duplicates:

1. **Restore Phase:**
    - The server queries /checkpoint for the latest snapshot.
    - It executes a TRUNCATE command on the local Cassandra table to clear any potentially stale or partial state.
    - It parses the snapshot string and re-inserts the data into Cassandra.
    - It updates its lastExecutedSeq to the sequence number of the checkpoint.
2. **Replay Phase:**
    - The server begins processing /log.
    - It strictly ignores any log entries where entrySeq <= lastExecutedSeq.
    - It executes any new entries that occurred after the checkpoint, bringing the node up to date with the cluster.