RSM/GigaPaxos - Design Document

1. Design Overview

This system utilizes **GigaPaxos** to implement a **Replicated State Machine (RSM)**, achieving fault tolerance and linearizable consistency. The application logic (MyDBReplicableAppGP) is decoupled from the complex consensus mechanism. GigaPaxos ensures that all state-changing client operations are agreed upon by a quorum of replicas before being applied deterministically to the backend database.

Architecture and Flow

The application follows a standard RSM model.

1. **Client Request:** A client sends a database command (CQL operation) to any GigaPaxos server.
2. **Consensus Layer (GigaPaxos):** The leader proposes the request. Consensus (via Paxos) is run to establish a total order for the request among all servers. GigaPaxos also handles failure detection and recovery.
3. **Application Execution (RSM App):** Once consensus is achieved, the command is delivered to every replica's MyDBReplicableAppGP in the exact same order.
4. **State Update (Cassandra):** The RSM App executes the corresponding CQL statement (INSERT, DELETE, etc.) against its locally managed Cassandra keyspace, guaranteeing identical state across all healthy replicas.

Key Design Decisions

1. **Replicable Interface**: The core logic is implemented in MyDBReplicableAppGP, fulfilling the necessary GigaPaxos contract methods:
    - execute(Request, boolean): Responsible for processing ordered CQL commands.
    - execute(Request): Delegates execution for unified state management.
    - checkpoint(String): Serializes the database state for persistence.
    - restore(String, String): Rebuilds the database state upon server recovery.
2. **State Representation**: The durable state uses Cassandra. Data from the grade table (schema: (id int, events list<int>)) is extracted and serialized into a custom JSON format (e.g., {"12345": [0, 1, 2]}) for checkpoints.

2. Implementation Details

2.1 Constructor Initialization

The constructor receives the unique keyspace identifier (e.g., "server0") from the GigaPaxos framework. It is responsible for establishing the cluster connection, creating the necessary keyspace (with replication factor 3 for durability), and initializing the Cassandra session used by the application.

2.2 Execute Method (execute(Request, boolean))

This method performs the state transition after a request has achieved consensus.

- **Command Extraction**: The raw request string is extracted from the Request object. Robust filtering logic is used to discard internal GigaPaxos control packets (e.g., JSON fragments, simple numerical IDs) that are not client commands.
- **Parsing**: Valid commands (e.g., INSERT:key:value) are parsed into a command verb, key, and value.
- **Execution**: The appropriate CQL statement is generated and executed directly against the local Cassandra session. The execution is successful if the CQL command completes without error.

2.3 Checkpoint Method (checkpoint(String))

For fault tolerance and log truncation, this method captures the full current state.

- **Process**: Queries all rows from the database table.
- **Serialization**: Converts the complete table contents into a single JSON string structure (e.g., {"key": "value"}).
- **Storage**: The resulting JSON string is returned to GigaPaxos, which persists it to the Derby log files (paxos_logs/).

2.4 Restore Method (restore(String, String))

This method ensures a crashed replica can recover to a consistent state.

- **Pre-Restore**: The method first clears the existing table state using a TRUNCATE command.
- **Deserialization**: It parses the incoming JSON checkpoint string.
- **Rebuilding State**: It iterates through the deserialized key-value pairs and executes an INSERT CQL statement for every single record, restoring the database to the checkpointed state before replaying subsequent log entries.

3. Troubleshooting and Robustness

Issue	Solution Implemented
Request Parsing	Implemented robust filtering in execute to distinguish internal GigaPaxos control messages (e.g., {"B", numerical IDs) from valid client application commands (e.g., INSERT:key:value).
Empty Checkpoint	The restore logic explicitly checks for null or empty checkpoint data ("[]") and performs a safe table truncation, preventing restoration errors.
Checkpoint Consistency	The custom JSON serialization method ensures a deterministic, controlled output format for safe parsing during the restore operation.