CS578
DISTRIBUTED COMPUTING AND SYSTEMS

PROGRAMMING ASSIGNMENT 3 PART 2

Design Document

Submitted by:
Sai Prathik Ravipalli (Spire ID: 34059746)
Prateek Bhindwar (Spire ID: 33979050)

USING CONSENSUS TO BUILD DISTRIBUTED SYSTEMS: Part 2

Replicated state machine (GigaPaxos): GigaPaxos to encapsulate the application as an RSM

Key Components

- **TableQueryList**: An inner class representing a list of queries associated with a specific table.
- **Fields**:
  - session: A `Session` object for interacting with the Cassandra database.
  - cluster: A `Cluster` object representing the Cassandra cluster.
  - keyspace: A `String` representing the Cassandra keyspace to which the server connects.
  - bufferQueries: A `Queue<String>` that buffers queries to be executed.
- **Core Methods**:
  - execute(Request request, boolean b):: Processes a `Request` object with a boolean flag 'doNotReplyToClient'.
  - execute(Request request): A method to execute a given request by sending it to the data store.
  - checkpoint(String s): Creates a checkpoint in the database, returning a JSON string representation of the database state.
  - restore(String s, String s1): Restores the database state from a given JSON string.
  - getRequest(String s) and getRequestTypes(): Methods related to handling request types, as required by the `Replicable` interface.

Functionality

- **Database Interaction**
  - The class primarily interacts with a Cassandra database, executing queries, and managing data states through checkpoints and restore operations.
- **Gigapaxos Integration**
  - As a `Replicable` implementation, this class can be used in Gigapaxos-based distributed systems, enabling replication and consensus management.
- **Request Handling**
  - It handles requests packaged in `RequestPacket` objects, allowing it to process and execute various database operations.
- **Checkpoint and Restore**
  - The checkpoint mechanism captures the current state of the database in a JSON format. The restore functionality uses this JSON to revert the database to a previous state

Performing Checkpoints and Restore in the application:

Checkpoint() method: Everytime the execute() method is invoked in the server, the request query is inserted into the queue **bufferQueries**. This queue serves as a buffer to store queries between a pair of checkpoint and a restore operation. When the checkpoint method is called, it first retrieves the list of tables in the keyspace and then for each table, retrieves all the rows present in the table at the moment. It then segregates the column names and corresponding values for each row, collects them into two different lists and finally creates an INSERT query which can insert the same row. In this way, we have *n* INSERT queries against given *n* rows for each table. We create an object of type **TableQueryList** to store each such pair of a table and its rows and form a list of such objects. We finally JSONify this list into a JSONArray and return it as a string, thus enabling any server to restore to exactly this state of the database (with the given *n* rows). Since we have already saved the current state of the database, we can empty the bufferQueries as we need to start capturing the upcoming requests until a restore() is called.

Restore() method: Once the restore() method is called with a checkpoint string, it first parses it as a JSONArray and extracts out the pairs of tables and queries. For each such pair (i.e. each table), the method executes the set of queries provided in the checkpoint. Before executing these INSERT queries, we first TRUNCATE the table to remove any previous data. Finally we empty the bufferQueries queue and execute all the requests present in it, to ensure we have not missed any requests which came during the interval between the last checkpoint and current restore.

Coordination server (Zookeeper): A coordination protocol using Zookeeper as a logically centralized service accessible to all replicas

MyDBFaultTolerantServerZK is a Java class designed to implement a replicated, fault-tolerant database server using Zookeeper for consensus and coordination. It integrates with Apache Cassandra for data storage and Zookeeper for managing distributed state and leader election

Key Components

- **Fields**
  - session: A `Session` object for interacting with the Cassandra database.
  - cluster: A `Cluster` object representing the Cassandra cluster.
  - myID: The identifier of the server.
  - serverMessenger: Handles network communication between servers.
  - leader: Tracks the current leader in the cluster.
  - queue: A `ConcurrentHashMap` to store pending requests.
  - notAcked: A `CopyOnWriteArrayList` tracking acknowledgments.
  - zookeeper: An instance of `ZooKeeper` for interacting with the Zookeeper service.

    - electionPath, leaderPath, logDirectoryPath, stateLogFilePath: Various paths used for Zookeeper nodes and logging.
- **Constructor**
    - Initializes connections to Cassandra and Zookeeper.
    - Sets up leader election and server node in Zookeeper.
    - Initializes the server messenger for inter-server communication.

Functionality

- **Distributed Coordination**: Leverages Zookeeper for distributed state management, leader election, and handling server node crashes and recoveries.
- **Fault Tolerance**: Implements mechanisms to detect leader crashes and re-elect a new leader. Maintains server state to handle server recoveries and ensure consistent states across replicas.
- **Client and Server Communication**: Handles communication with clients and other server nodes, ensuring proper forwarding and processing of requests and acknowledgments.
- **State Management**: Manages server states and logs actions for potential recovery scenarios.

Leader Election and Crash Detection

The algorithm uses Zookeeper to manage the leader election process.

- **Ephemeral Nodes for Leadership Declaration**: When a server starts, it attempts to create an ephemeral node at a predefined path in the Zookeeper namespace (e.g., `/leader`). Ephemeral nodes in Zookeeper exist as long as the session that created them is active. This mechanism is used to signify leadership - the node that successfully creates this ephemeral node is considered the leader.
- **Leader Determination Based on Node ID**: The server with the lowest ID (as determined from the sorted list of active node IDs) is selected as the leader. This is done by having each server check the list of active nodes and compare their IDs.
- **Handling Leader Failure**: The algorithm sets a watch on the leader node. If the leader node fails (e.g., due to server crash or network partition), the ephemeral node in Zookeeper will be automatically removed. This removal triggers watches set by other servers, informing them of the leader’s failure.
- **Re-election on Leader Failure**: Upon detecting the leader's failure, remaining servers will try to create the ephemeral `/leader` node themselves, effectively starting a new round of leader election. The server with the lowest ID among the currently active nodes will become the new leader.

- The list of alive nodes are stored in *myNodeConfig* where the **servers are dynamically removed and added upon crash and recovery leveraging ephemeral node feature of Zookeeper**

Server State Preservation and Recovery

- We have implemented state preservation recovery mechanism using local file system to enable the nodes to save their states at each stage below of the transaction processing
    - On Receipt of REQUEST Type
    - On Receipt of PROPOSAL Type
    - On Receipt of ACKNOWLEDGEMENT Type
- This facilitates the functionality, if a node crashes at any stage and recovers, it first checks its logs and handles different message types according to the already implemented consistency logic.
- On server startup, myNodeConfig is loaded from the file, providing the server with the latest alive nodes. By persisting myNodeConfig on the file system, the server ensures that it has a consistent view of the cluster configuration across restarts