CS 578 - PA3.2

Gigapaxos:

No of test cases passing: 13/13

Design:

■ GigaPaxos has a simple Replicable wrapper API. This API requires three methods to be implemented:

```
boolean execute(Request request)
String checkpoint(String name)
boolean restore(String name, String state)
```

In addition, the state stored during checkpointing: A handle containing the filename of the file containing the entire state of the cassandra table retrieved using “select* from keyspace.grade;

■ The following outlines the design for every method:

1. **boolean execute(Request request):**

○ The query to be executed is extracted from the Request object received.
○ The query is then executed using session.execute(query)

2. **String checkpoint(String name):**

○ A file is created which is named “checkpoint+myID” for each server.
○ If the file already exists, then it is truncated. (Done to ensure that we do not have partial state saved, when we are trying to create a new checkpoint).
○ All the data from the Cassandra table (select*) is added into the checkpoint file.
○ The event data is retrieved from the rows of the ResultSet after executing select* on the table.
○ If the events are null or empty, an empty event string (“[]”) is added to the checkpoint file for the id.
○ The final checkpoint is created by calling the createCheckpointHandle method from the LargeCheckpointer from the utils in Gigapaxos library.
○ Finally, the final checkpoint handle is returned as a string.
○ If an exception is raised during the process, we re-attempt to checkpoint our state until successful.

3. **boolean restore(String name, String state):**

○ A check is made to see if the current checkpointed state in the restore function parameters is null or empty.

■ If state is null or empty, we attempt to reinitialize cassandra table by creating an empty table “grade” in the keyspace for that server, if such a table does not already exist.
● If the table exists, then it is truncated. (This is done to ensure a clean slate so that we do not have partial state stored as we need to execute all requests from the start since no checkpoint is available).
■ If state is not null or empty, we restore the checkpoint by calling the “restoreCheckpointHandle” method from the LargeCheckpointer from the utils in Gigapaxos.
○ We then check if a table “grade” already exists.
■ If it does not exist, we create a new empty table.
■ If it does exist, we truncate it.
○ We then read from the checkpoint file and insert records into the table “grade”.

Zookeeper:

No of test cases passing: 3/13

Design:
Our design involves our replicated servers and a single zookeeper server. We use the existing implementation of the leader based consistent database (with very few modifications) for ensuring total order. In addition to this, we use zookeeper for group membership and leader election in case of faults to make our consistent database fault tolerant.

Each replicated server maintains and stores group information in its memory. This group information tells the server about active servers, current leader of the group and all the nodes that are part of the group.

Using Zookeeper to ensure Fault Tolerance:

Group Membership:
We need to check for group membership of each node for two scenarios:

1) Whenever a follower crashes,the leader needs to know that the follower is not up. In that case, the leader does not send the request to the crashed node while at the same time checkpointing the state of the table after executing the request. When the follower is back up, it can ask the leader for a checkpoint to restore its table state.
2) If the leader is the node that crashed, then the leader election protocol needs to be executed.

Algorithm: Create a parent znode /all_nodes, this znode will be used to store any server that connects to the group. Create a parent znode /live_nodes, this znode will be used to store only the live nodes in the group and will store ephemeral child znodes. If any server crashes or goes down, the respective child ephemeral znode will be deleted. Any server connecting to the group will create a new persistent znode under /all_nodes. Any server connecting to the group will create a new ephemeral znode under /live_nodes. Add a watch for any change in children of /all_nodes. If any server is added or deleted to/from the group, all servers in the group needs to

be notified and their group info needs to be updated. Add a watch for any change in children of /live_nodes so that servers can update their group info as and when required.

Leader Election:
We need to perform the leader election protocol whenever the leader crashes and other nodes can assume the position of the leader.
Algorithm: Create a persistent znode /election. Each server joining the cluster will try to create an ephemeral sequential znode /leader-<seq_number> under node /election with data as hostname. Znode with the least sequence number will be automatically considered as a leader. Each server in the cluster sets a watch on the child znode with one less sequence number. This avoids the herd effect whenever a leader goes down as only the next candidate that would become a leader should get the notification. Once all servers gets the leader's znode-delete notification, they again fetch all children under the persistent znode and get the data associated with the child znode that has the least sequence number and store that as the new leader in its own memory.

Why the test cases are not passing:

The test cases are not passing as the design has not been implemented completely.