4.1 – GigaPaxos RSM Design Document

Ashi Sinha (34743339)
Sameeksha Bhatia (35243160)

Our goal in this assignment (PA4.1) is to build a fault-tolerant key–value store using the GigaPaxos Replicated State Machine (RSM) option. Instead of implementing our own consensus algorithm, we rely on GigaPaxos to order, replicate, and deliver client commands. Our responsibility is to implement an application class, MyDBReplicableAppGP, that applies these ordered commands to a Cassandra backend. In simple words, GigaPaxos ensures that all replicas receive the same sequence of requests, and our code defines what each request does and how the state is recovered after a crash.

Each replica uses its own Cassandra keyspace, which is passed into the constructor through args[0]. For example, one replica might be server0, another server1, and so on. Inside each keyspace, we create the same table named “grade” with columns id (the primary key) and events (a list of integers). The tests send SQL commands such as:
insert into grade (id, events) values (123, []);
and
update grade set events = events + [5] where id = 123;
The events list stores the history of updates for each record.

Because the test SQL statements refer to the table name simply as “grade,” but Cassandra requires a fully-qualified name like <keyspace>.grade, we added a helper function called **maybeQualifyKeyspace**. This function checks whether the SQL already contains <keyspace>.grade. If not, it replaces the bare table name “grade” with the fully-qualified version. For example, if the keyspace is server2, then a statement like “insert into grade …” becomes “insert into server2.grade …”. This ensures that each replica writes to its correct local keyspace and that all operations are applied to the proper table.

The main GigaPaxos callback we implement is **execute**(Request request). Here, we extract the SQL string from the incoming request. When the request is a RequestPacket, we first read rp.requestValue. If that field is null, we fall back to reading the raw JSON produced by request.toString() and manually extract the “QV” field. If both attempts fail, we log an error and return false. When we do find a valid SQL string, we pass it through maybeQualifyKeyspace, log the final SQL, and execute it on Cassandra. If the update succeeds, we return true; if an exception occurs, we catch it and return false. Returning true signals to GigaPaxos that this replica has successfully applied the request and can move forward to the next one.

To support fault tolerance, we implement both **checkpoint**() and restore(). In checkpoint(), we read all rows from <keyspace>.grade. For each row, we extract the id and its events list. Since the events list can sometimes be null, we convert null into an empty list before storing it. We then create a JSON object where each key is an id and the value is its list of integers. The string

form of this JSON is returned as the checkpoint, which represents the complete logical state of the table at that moment. GigaPaxos calls checkpoint(name) every CHECKPOINT_INTERVAL = 10 requests.

The **restore**() method performs the opposite task. First, we truncate the table to remove any old or inconsistent state. If the checkpoint string is empty or “{}”, then there is no state to restore and we finish immediately. Otherwise, we parse the JSON string back into key–value pairs. For each id, we rebuild its events list and insert it into <keyspace>.grade using a prepared statement. After restoration, the replica should contain exactly the same logical state it had before the crash, allowing it to rejoin the system correctly.

The design applies commands deterministically, maintains Cassandra-backed state, and restores it successfully after crashes, meeting the requirements of PA4.1.