# CS590CC Programming Assignment 3

## [PA3 Part 2]: USING CONSENSUS TO BUILD DISTRIBUTED SYSTEMS

### Group members:


- Pragya Sarda
- Priyanka Singla


### Algorithm:

The goal of this assignment is to use consensus to build a fault-tolerant replicated datastore application. The datastore involved is Cassandra and the target is to get a consensus on the order of queries being executed in Cassandra given system failures can happen, using one of the following three options:

1. Coordination server (Zookeeper)
2. Replicated state machine (GigaPaxos)
3. Custom protocol (Custom)

We are choosing **method 2, Gigapaxos** which is a replicated state machine to implement a fault tolerant replicated datastore.

GigaPaxos handles the total ordering of the requests so that every request is executed in the same order at all servers even when node failures occur. To implement this consensus algorithm in a fault tolerant manner, we are implementing MyDBReplicableAppGP, an application which is an implementation of Replicable interface. It has the following three methods to implement:

**Execute:**

There are two implementations of execute, where one execute(Request) is called to execute the query at the cassandra instance, and execute(request, b) is called when the result needs to be sent to the client. If b is true, the result is returned to the client, otherwise not. For execution of queries, we are setting up the Cassandra instance and connecting to the keyspace which is being passed in the args. Once the connection is set up and connected to the keyspaces successfully, we are retrieving the request package. Then the request is typecasted to the RequestPacket and the request value is retrieved. Deserialization of the packet is done once the request value is converted into a JSON object, where we are fetching the query string out of

it using key “QV”. Once we have a query string, we execute it using the instantiated session as Gigapaxos would maintain the total ordering of the requests coming in. Each of these queries are being logged to aid the fault tolerant cases. But as more and more of these requests are being logged, the size of the logs will keep on increasing. To keep a bound on the size of the logs, we are checkpointing the current state of the servers after every 400 requests and clear out all the logged requests. Whenever a crashed service would recover from a failure, it would fetch the last checkpointed state and implement the logged requests on top of this checkpointed state to recover to the final state.

**Checkpoint:**

For checkpointing, we are taking the state of the table being updated and are storing this state. To fetch the state of the table in cassandra, we are using Copy To command where it exports data from a table into a CSV file. Each row is written to a line in the target file with fields separated by the delimiter. We are maintaining a keyspace variable tracking the keyspace for each of the servers. After every 400 requests, checkpoint function is called, which executes the following query to checkpoint the entire state of the table for a particular instance of the server.

```
COPY <TableName> TO <FileName> WITH HEADER = TRUE ;
```

**Restore:**

Restore function is called every time any server comes live. For restoring, we are reading the checkpointed file and loading the state from that. Once the state is loaded, GigaPaxos would perform the logged requests so that the latest state can be restored and the new requests can be executed. To load the checkpointed state, following query is being used:

```
COPY <TableName> FROM <FileName> WITH HEADER = TRUE ;
```

To execute both of these COPY TO and FROM requests, we are using runtime execution to execute a cqlsh command after starting a cqlsh shell from the java environment.

## Observation:

A peculiar thing we noticed while executing the test failures is the randomness. A lot of times random test cases would abruptly fail. We have not been able to point to the exact reason behind this. We will dig deeper into the test cases and flow of execution to understand this random behaviour.

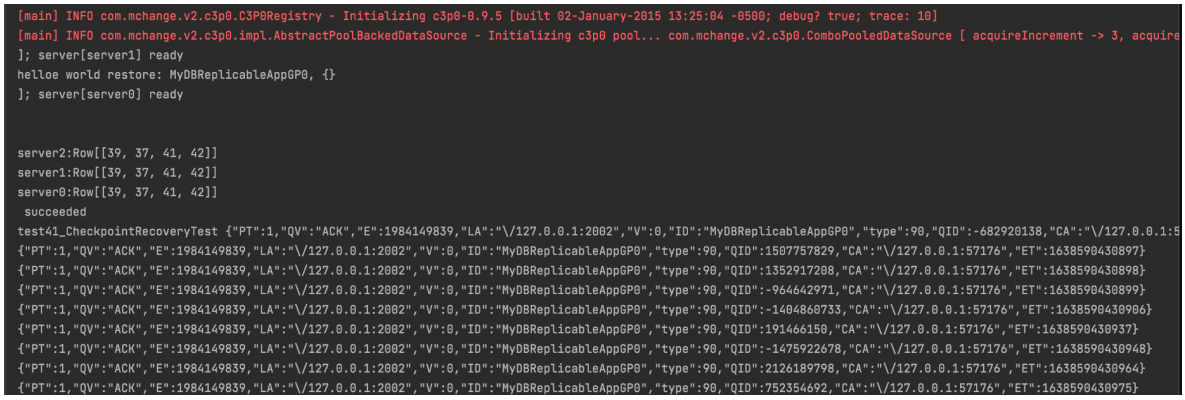
## Output:

After implementing the above, following are the results of grader test files:

1. GraderConsistency: All test cases ran successfully and passed.
2. GraderFaultTolerance: All Test cases passed.

Following are the screenshots of the test cases of Grader Fault Tolerance:
Initial tests (31,32,33) succeeded:


Test 40 succeeded, 41 started:




Test 41 succeeded: